The 37th Symposium on Implementation and Application of Functional Languages

October 1-3, 2025

Montevideo, Uruguay

Pre-proceedings

Contents

Regular papers

Recipe-Based Errors

Methodology, Implementation, and Evaluation

Marco T. Morazán
Seton Hall University
South Orange, NJ, USA
morazanm@shu.edu

Shamil Dzhatdoyev
Axoni, USA
New York, NY, USA
shamil.d11@gmail.com

Josephine A. Des Rosiers
Penguin Random House
New York, NY, USA
josieadesrosiers@gmail.com

Rose Bohrer
Worcester Polytechnic Institute
Worcester, Massachusetts, USA
rbohrer@wpi.edu

Andrés M. Garced
Seton Hall University
South Orange, NJ, USA
maldona2@shu.edu

David Anthony K. Fields
Seton Hall University
South Orange, NJ, USA
fieldsda@shu.edu

Abstract

The design and implementation of error messaging systems is a long-lasting challenge for programming language developers. The challenges become more pronounced when students are exposed to a programming language for the first time. Poor error messages make it challenging for students to understand and correct programming bugs. In part, the problem is that the search space for a solution is large, and students lack the experience to navigate this space. This article puts forth the thesis that in courses following a design-based methodology recipe-based errors help students resolve bugs. That is, the error messaging system provides scaffolding to reduce the size of the search space by focusing the student's attention on detected design errors. To explore this hypothesis, recipe-based errors are implemented as part of, FSM, a domain-specific functional language for the Formal Languages and Automata Theory classroom. Each error message is enhanced with the step of the design recipe that has not been successfully completed. In addition, the error messaging system integrates testing–a fundamental step in program design–as a criteria for generating error messages. Empirical data collected at two US-based universities is presented that suggests students find recipe-based errors clear, succinct, and a factor in reducing debugging time. In addition, the data suggests that the association of design recipe steps with error messages is useful.

CCS Concepts

• **Software and its engineering** → **Functional languages**; **Domain specific languages**; **General programming languages**; • **Theory of computation** → *Formal languages and automata theory*; • **Applied computing** → **Education**.

Keywords

Error Messages, Design Recipes, Formal Languages and Automata Theory Education, Functional Programming

IFL 2025, Montevideo, Uruguay

ACM Reference Format:
Marco T. Morazán, Shamil Dzhatdoyev, Josephine A. Des Rosiers, Rose Bohrer, Andrés M. Garced, and David Anthony K. Fields. 2025. Recipe-Based Errors: Methodology, Implementation, and Evaluation. In *Proceedings of the 37th Symposium on Implementation and Application of Functional Languages (IFL 2025), October 1–3, 2025, Montevideo, Uruguay*. ACM, New York, NY, USA, 13 pages.

1 Introduction

Developing error messages that are useful and accurate has been a challenge for the programming languages and the human-computer interaction communities for at least 60 years since FORTRAN [54] and ALGOL 60 [60] developers first took steps to generate them. Useful and accurate error messages are important, because poor error messages have been found to make it difficult to learn how to program [29, 37] and to learn a new language [18, 26, 66]. These difficulties are attributed to error messages being, for example, cryptic [1, 2, 4, 5, 28, 35, 66], inadequate [1, 3, 9, 31, 54], frustrating [6, 23], and a barrier to progress [5, 6, 37]. This all means that the interaction with poor error messages is inefficient and costly [4].

These problems are accentuated in students learning the abstractions offered by a new programming language. This occurs for multiple reasons such as the use of technical jargon in the message [4, 15, 35, 38, 57], students not having assimilated the heuristic to correct the first error first (when multiple errors are reported) [6, 55], students interpreting the location highlighted for where an error is detected as a signal for where to edit the program to make corrections [38, 68], and the lack of precise information in the message [26, 38]. Efforts to address these problems and create good error messages include suggesting successful solutions by others [25], eliminating jargon vocabulary in error messages [4, 15, 35], using familiar classroom vocabulary in error messages [15, 38], enhancing error messages with more information [5, 26, 31, 35, 58], making error messages more succinct [15, 26], and reducing redundant and repeated information [4, 27].

Good error messages alone, however, are only part of the solution to help students learn new abstractions. Students also need a systematic approach for problem solving and program implementation. For example, an arguably successful approach to help programming beginners uses a program-by-design methodology [7, 21]. This approach utilizes a collection of domain-specific languages (DSLs) embedded in Racket [22] (commonly referred to as

the Racket student languages). These DSLs restrict the amount of syntax a student may use to implement solutions to problems. In addition, they have tailored-made student-friendly error-messaging systems and are tightly-coupled with textbooks for beginners in programming [19, 44, 45]. At the heart of the approach, first pioneered by Felleisen et al. [19] and later expanded to a two-semester course by Morazán [44, 45], is the design recipe–a series of concrete steps, each with a specific outcome, that help students go from a problem statement to a documented and validated solution expressed as a program. Interestingly enough, the error messages in the Racket student languages do not refer to the steps of the design recipe. In the experience of the authors, this leads students to repeatedly wonder what step of the design recipe they have incorrectly performed when an error is thrown. Thus, suggesting that the program-by-design methodology may benefit from error messages that refer to the steps of the design recipe not successfully completed. We coin such error messages as *recipe-based errors* (RBEs). The goal is to encourage students to think, when debugging, about design recipe steps to determine the problem that led to an error.

To explore the idea of coupling the pedagogic pillars of the program-by-design methodology (i.e., domain-specific languages, design recipes, error messages, textbooks, and instructors), the error-messaging system for FSM (Functional State Machines) [53] is redesigned to produce RBEs. FSM is a functional domain-specific language embedded in Racket [22] developed for the Formal Languages and Automata Theory (FLAT) classroom. The worked described is part of a larger effort to vertically integrate the program-by-design methodology into the Computer Science curriculum. The described contract-based error messaging system complements Racket's error messaging system. That is, error messages for Racket syntax or misuse of Racket functions are generated by Racket's error messaging system. FSM's novel error-messaging system addresses errors detected when using FSM abstractions. This is important, because generating a Racket error message referencing a Racket function in an internal FSM implementation file breaks the abstraction barrier expected by the programmer. That is, the programmer expects an FSM-relevant error message when FSM abstractions are used. In addition, the described system also integrates unit testing–an essential design step–to reduce the amount of syntax students must write for unit tests and to encourage the construction of validated machines and grammars. Every error message generated includes a reference to an unsuccessfully completed design recipe step. We note that FSM's new error messaging system is the first to ever integrate design recipes and unit testing.

To evaluate student experiences with RBEs, we explore the following research questions:

- **RQ1:** Are RBEs useful in debugging?
- **RQ2:** Why are RBEs useful?
- **RQ3:** Is a design recipe step useful in an error message?

RQ1 focuses on how students use RBEs as a debugging tool. In contrast, if RBEs are useful in debugging, **RQ2** and **RQ3** focus on message characteristics that help explain why they are useful. These questions are explored through the administration of anonymous surveys at two US-based universities teaching an undergraduate FLAT course using the methodology and the RBEs described in this article as well as the same textbook (i.e., [46]).

The article is organized as follows. Section 2 provides an overview of designing and programming using FSM. Section 3 describes RBEs. Section 4 discusses FSM contracts for machine and grammar constructors to produce RBEs. Section 5 presents how testing and contracts are incorporated for RBEs. Section 6 presents the empirical data collected. Section 7 contrasts with related work. Section 8 discusses the implications and the limitations of the presented work. Finally, concluding remarks and directions for future work are presented in Section 9.

2 Designing and Programming in FSM

FSM is dynamically typed and, among other features, allows programmers to easily define deterministic finite-state automata (dfa), nondeterministic finite-state automata (ndfa), pushdown automata (pda), Turing machines (tms), composed Turing machines (ctms) [43], regular grammars (rgs), context-free grammars (cfgs), and unrestricted grammars (urgs). In addition, programmers can also implement the construction algorithms they design as part of their constructive proofs. Integrated into FSM is a powerful suite of static and dynamic visualization tools to help students understand FLAT principles [47–51].

This section presents an overview of designing and implementing state machines and grammars in FSM. The design recipes for state machines and grammars are outlined and an illustrative example for each is discussed. In addition, the arguments for machine and grammar constructors are specified to make clear what is tested by the novel error-messaging system. In the interest of brevity, we use a single regular language to illustrate the design process for both machines and grammars:

L={w|w has an odd number of as and an odd number of bs}

Although only the development of a dfa and of a rg are presented, we note that the development for other types and machines and grammars follows in the same manner.

2.1 State Machines

2.1.1 State Machine Constructors. The constructor signatures for a dfa, an ndfa, a pda, and a tm are:

make-dfa:	K	Σ	S	F	δ	['no-dead]	→	dfa
make-ndfa:	K	Σ	S	F	Δ		→	ndfa
make-ndpda:	K	Σ	Γ	S	F	Δ	→	pda
make-tm:	K	Σ	Δ	S	F	[Y]	→	tm

K denotes a set of states. Σ denotes the input alphabet. S∈K denotes the starting state. F⊆K denotes the set of final states. Γ denotes a stack alphabet. A transition function and a transition relation are denoted, respectively, by δ and Δ. Y∈F is an optional argument for the Turing machine constructor denoting the accept state (omitting this argument signals that the tm computes a function instead of deciding or semideciding a language). Finally, the optional argument for the dfa constructor, 'no-dead, signals that the transition function is fully specified (omitting this argument has the constructor add a dead state and transitions into it to fully specify the transition function).

(1) Name the machine and specify alphabets
(2) Write unit tests
(3) Associate a state with tracked conditions, and identify the start and final states.
(4) Formulate the transition relation
(5) Implement the machine
(6) Run the tests and, if necessary, redesign

Figure 1: The design recipe for state machines.

2.1.2 State Machine Argument Types. The arguments to construct state machines are partitioned into two sets: those that do not have dependencies and those that have dependencies. The arguments that do not have dependencies are K, Σ, and Γ. The arguments that have dependencies are S, F, δ, Δ, and Y. For instance, S and Y depend on K given that both must be members of K.

The transition rules depend on K, Σ, and Γ and are represented as lists:

tm: ((K {Σ ∪ {@ _}}) (K {Σ ∪ {L R _}})) **dfa**: (K Σ K)
pda: ((K {Σ ∪ {ϵ}} Γ^*) (K Γ^*)) **ndfa**: (K {Σ ∪ {ϵ}} K)

Each transition is a list that has a source state, an element to read (which may be empty), and a destination state. In addition, pda-rules specify the elements to pop and push and tm-rules specify the action to take on the tape[1].

2.1.3 The State Machine Design Recipe. The abbreviated design recipe for state machines is displayed in Figure 1[2]. To illustrate the result of following the steps of the design recipe, consider the dfa developed by a student displayed in Figure 2 for L (defined at the beginning of Section 2). The FSM implementation is displayed in Figure 2a. The results for step 1 of the design recipe are the name odd-a-odd-b (on line 8) and the input alphabet (on line 10). The unit tests to satisfy step 2 are displayed on lines 17–24. The tests use RackUnit's check-equal? [67] and use FSM's sm-apply to apply a given machine to a given word. Observe that students are writing a fair amount of repeated code for testing. The conditions the states represent to satisfy step 3 are displayed in lines 2–7. The transition function developed for step 4 is captured on lines 13–14. The defined dfa satisfies step 5 and running the tests for step 6 reveals that they all pass. The transition diagram generated using FSM's sm-graph is displayed in Figure 2b. The start state, S, is highlighted in green and the final state, C, is rendered using a double circle.

2.2 Grammars

2.2.1 Grammar Constructors. The constructor signatures for a regular (rg), context-free (cfg), and unrestricted (urg) grammars are:

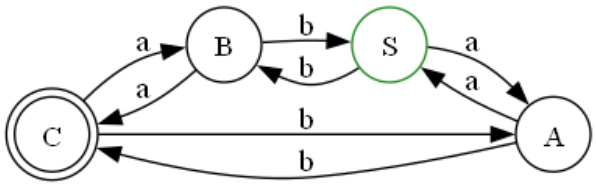
make-rg: V Σ δ S → rg
make-cfg: V Σ δ S → cfg
make-grammar: V Σ δ S → urg

From left to right, V denotes the nonterminals; one for each syntactic category needed. Σ denotes the input alphabet. The production rules are denoted by δ. The starting nonterminal is denoted by S.

[1]The symbols @, _, L, and R, respectively, denote the left-end marker, a blank, moving the head left, and moving the head right.
[2]The full design recipe has two extra steps for machine verification [46], which are omitted here given that they are not relevant to the error-messaging system.

```
1  #lang fsm
2  ;; L = {w|w has an odd number of as and an odd number of bs}
3  ;; State Documentation
4  ;;  S: even a and even b, starting state
5  ;;  A: odd a and even b
6  ;;  B: even a and odd b
7  ;;  C: odd a and odd b, final state
8  (define odd-a-odd-b
9    (make-dfa '(S A B C)
10             '(a b)
11             'S
12             '(C)
13             '((S a A)  (S b B)  (A a S)  (A b C)
14               (B a C)  (B b S)  (C a B)  (C b A))
15             'no-dead))
16
17 (check-equal? (sm-apply odd-a-odd-b '()) 'reject)
18 (check-equal? (sm-apply odd-a-odd-b '(a b b a b))
19                'reject)
20 (check-equal? (sm-apply odd-a-odd-b '(a a b a a b))
21                'reject)
22 (check-equal? (sm-apply odd-a-odd-b '(a b)) 'accept)
23 (check-equal? (sm-apply odd-a-odd-b '(a b b b a a))
24                'accept)
```

(a) The dfa for L in FSM.



(b) The transition diagram.

Figure 2: L={words with odd a and odd b}.

2.2.2 Grammar Argument Types. The types needed to construct grammars are also partitioned into those that do not have dependencies and those that have dependencies. The following descriptions summarize the properties tested by the error messaging system. The arguments without dependencies are described as follows:

V: A set of symbols denoted the same way as machine states.
Σ: Denoted the same way as for state machines.

The arguments with dependencies are described as follows:

δ A set of production rules.
S: S∈V.

The production rules, δ, are represented as lists whose values depend on V and Σ:

rg: (V → {ϵ | Σ | ΣV})
cfg: (V → {ϵ | {Σ | V}$^+$})
urg: ({Σ | V}*V{Σ | V}* → {ϵ | {Σ | V}$^+$})

Each production rule must have at least one nonterminal on the left hand side and the right hand side is an aggregate symbol. For rg, only S may produce, ϵ, the empty word.

(1) Name for the grammar and specify the alphabet
(2) Associate a nonterminal with each with a nonterminal and specify the starting nonterminal
(3) Develop the production rules
(4) Write unit tests
(5) Implement the grammar
(6) Run the tests and redesign if necessary

Figure 3: The design recipe for grammars.

2.2.3 The Grammar Design Recipe. The abbreviated design recipe for grammars in displayed in Figure 3 [46, 52][3]. As with machine design, students are taught to systematically follow the steps.

To illustrate the results for the steps of the design recipe in action, consider the `rg` displayed in Figure 4 to generate the same language decided by the state machine developed in Section 2.1.3. The results for Step 1 are displayed on lines 8 (the grammar's name is `odd-a-odd-b-rg`) and 10 (the alphabet is `'(a b)`). The documentation for the nonterminals to satisfy Step 2 is displayed on lines 3–7. Observe that it clearly indicates the starting nonterminal. The production rules for Step 3 are displayed in lines 11–14. The unit tests developed for Step 4 are on lines 17–29. Once again, observe the amount of repeated code written for unit tests. The defined `rg` satisfies Step 5 and running the tests for Step 6 reveals that they all pass. Figure 4b displays the final derivation tree generated by `FSM`'s dynamic visualization tool, `grammar-viz`, to illustrate word derivation.

3 RBE Design

The error messaging system's design follows principles that we categorize over three dimensions: composition, content, and reporting time. In terms of composition, error messages use: jargon-free vocabulary that is familiar to students from both lectures and the textbook, a positive non-accusatory tone, and simple sentences. The goals fostered by these characteristics include not intimidating students, making it easier for students to comprehend the content of messages, and making messages more easily understandable for non-native English speakers [24].

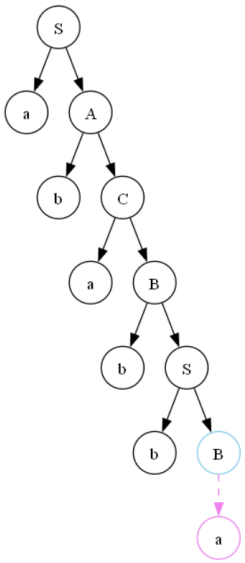
In terms of content, error messages must logically and clearly explain why an argument provided for a component has an error, identify the design recipe step not successfully completed, and only contain information relevant to the error without offering prescriptive solutions. The goals fostered by these characteristics include helping students focus on the reason for the error in terms of their design efforts and on not leading students down an unfruitful resolution path.

In terms of reporting time, error messages for arguments that do not depend on other arguments for correctness are reported first, error messages for optional arguments are reported only after no error messages for required arguments are detected, and testing errors are reported last. The goal fostered by these characteristics is not burdening students with derivative error messages[4]. For instance, an error in the non-dependent argument for the states of

[3]The full design recipe has two extra steps for grammar verification [52], which are omitted here given that they are not relevant to the error-messaging system.
[4]Errors detected given the existence of previous errors.

```
1  #lang fsm
2  ;; L = {w|w has an odd number of as and an odd number of bs}
3  ;; Nonterminal Documentation
4  ;;   S: generates words in L, starting nonterminal
5  ;;   A: generates an even number of as and an odd number of bs
6  ;;   B: generates an odd number of as and an even number of bs
7  ;;   C: generates an even number of as and an even number of bs
8  (define odd-a-odd-b-rg
9    (make-rg '(S A B C)
10            '(a b)
11            '((S → aA)  (S → bB)
12              (A → b)   (A → aB)   (A → bC)
13              (B → a)   (B → aC)   (B → bS)
14              (C → aB)  (C → bA))
15            'S))
16
17 (check-equal? (grammar-derive odd-a-odd-b-rg '())
18                "() is not in L(G).")
19 (check-equal?
20   (grammar-derive odd-a-odd-b-rg '(a b b a b))
21   "(a b b a b) is not in L(G).")
22 (check-equal?
23   (grammar-derive odd-a-odd-b-rg '(a a b a a b))
24   "(a a b a a b) is not in L(G).")
25 (check-equal?
26   (last (grammar-derive odd-a-odd-b-rg '(a b))) '(ab))
27 (check-equal?
28   (last (grammar-derive odd-a-odd-b-rg '(a b b b a a)))
29   '(abbbaa))
```

(a) `FSM` implementation.



(b) A derivation tree using L.

Figure 4: The regular grammar for L.

a machine may also cause errors in the argument for the transition relation. Fixing the error in the argument for states may also fix the error in the transition relation and, therefore, the student is not burdened with the latter error.

FSM's new error messaging system is the first to integrate design and unit testing. Syntax is provided for the programmer to optionally provide a list of words that ought to be accepted/generated and/or a list of words that ought to be rejected/not-generated by a proposed machine/grammar. If used, an error, indicating the design recipe step not successfully completed, is thrown when the expected behavior is not obtained and, as a consequence, the machine/grammar is not constructed. In addition, the integration of testing addresses a common student complaint regarding the amount of repeated code needed to write unit tests (e.g., using testing libraries such as RackUnit [67]). Furthermore, this feature also allows instructors to easily run a test suite of their choice as they grade assignments.

Finally, machine RBEs refer to one of the six steps in Figure 1. Every machine RBE reports a single step unsuccessfully completed, which may include multiple errors. For instance, if several words that ought to be accepted are rejected then a single error message referencing Step 2 and all such words is generated. Similarly, grammar RBEs refer to one of the six in Figure 3 and report a single unsuccessfully completed step, which may include multiple errors.

4 FSM Contracts

This section outlines machine- and grammar-constructor contracts. First, the general outline for contract design is presented. Second, contract sharing to avoid code duplication is described. Third, contract implementation for arguments that do not have dependencies is outlined. Fourth, contract implementation for arguments that have dependencies is presented.

4.1 General Design

In general, machine- and grammar-constructor contracts combine contracts for each of their arguments. The `->i` contract combinator is used for both, given that some arguments depend on other arguments.

4.1.1 Machine-Constructor Contract Outline. The general structure of a machine-constructor contract is outlined in Figure 5. For each constructor argument, there is a stanza that names the argument and associates a contract with it. Any given machine constructor does not contain all the stanzas outlined in Figure 5. For instance, the constructor for `ndfa`s does not contain contracts for `gamma` or `accept`.

The first set of stanzas are for required arguments (lines 2–8). Within this set, the stanzas for arguments without dependencies are listed first (lines 2–4) followed by the stanzas for arguments with dependencies (lines 5–8). Each stanza for an argument with dependencies includes the dependencies in a sublist after the argument's name. This ordering is done by design to first report errors for arguments without dependencies. As mentioned above, this is important, for example, because fixing a bug in an argument with no dependencies may resolve a bug in an argument with a dependency. The reverse is not true. For instance, consider `(A b C D)` being the argument for `states` and `(B a D)` being part of the

```
1  (→i (;; required arguments
2       [states <contract>]
3       [sigma  <contract>]
4       [gamma  <contract>]   ;; only for make-ndpda
5       [start  (states) <contract>]
6       [finals (states) <contract>]
7       [rules  (states sigma <other dependencies>)
8               <contract>]
9       ;; optional arguments
10      ([add-dead <contract>] ;; only for make-dfa
11       [accept (finals) <contract>] ;; only for make-tm
12       #:accepts
13         [accepts (states sigma start finals rules
14                    <other dependencies>)
15                  <contract>]
16       #:rejects
17         [rejects (states sigma start finals rules
18                    <other dependencies>)
19                  <contract>])
20       ;; returned value
21       [result <contract>])
```

Figure 5: Template for machine-constructor contracts.

argument for `rules`. We can observe that `b` is a bug in the first and that (with the given argument for `states`) `B` is a bug in the second. Resolving the bug in `states` by changing `b` to `B` resolves the bug in the transition rule. In contrast, changing `B` to `b` in the transition rule does not resolve the bug in the argument for `states`. By validating the argument with no dependency first, the student only sees the first error. Thus, better focusing the student's attention on the reason for the error. Observe that this means that our error messaging system has the accepted practice of fixing the first error first built-in [55].

Following the stanzas for required arguments, the stanzas for the optional arguments are listed (lines 10–19). Within these, the lists for words to accept and reject are listed last. This is purposely done, because these arguments cannot be tested before all other arguments have satisfied their contracts. This follows by observing that testing requires the machine to be constructed. Finally, the last stanza is for the returned value as required by Racket contracts.

To outline using the syntax, consider the header for the `ndfa` constructor:

```
(define/contract
  (make-dfa states sigma start finals rules
            #:accepts [accepts '()]
            #:rejects [rejects '()])
```

A required argument without dependencies is `states`. Its stanza is outlined as follows:

```
[states (and/c ...)]
```

The ... contain a contract for each property tested and they are combined using `and/c` given that all must hold for the argument to be valid. For an argument with dependencies, like `finals`, its stanza is outlined as follows:

```
(→i (;; required arguments
     [nts   <contract>]
     [sigma <contract>]
     [delta (nts sigma) <contract>]
     [start (nts) <contract>])
     ;; optional arguments
     (#:accepts [accepts (nts sigma delta start)
    <contract>]
      #:rejects [rejects (nts sigma delta start)
    <contract>])
     ;; returned value
     [result <contract>])
```

Figure 6: Template for grammar-constructor contracts.

```
[finals (states) (and/c ...)]
```

This stanza lists `states` as the only value it depends on. The contracts provided to `and/c` may refer to `states`. Finally for keyword parameters, the keyword must appear before its contract stanza. For instance, the stanza for a list of words that ought to be accepted by an `ndfa` is outlined as follows:

```
#:accepts [accepts (states sigma start finals rules)
                   (and/c ...)]
```

The list of words that ought to be accepted depends on all arguments needed for the machine under construction. This follows, as mentioned above, by observing that the machine must be built to apply it to each word.

4.1.2 Grammar-Constructor Contract Outline. The general structure of a grammar-constructor contract is outlined in Figure 6. The structure is similar to that for a machine-constructor. All grammar constructors require the same four inputs. The nonterminals, `nts`, and the alphabet, `sigma`, do not have dependencies and are listed first. The remaining two required arguments depend on these. Similarly to machine constructors, the optional word lists that must be generated and that must not be generated depend on all the required arguments given that the grammar must be constructed. As machine contracts, grammar contracts also have the accepted practice of fixing the first error first built-in.

To outline using the syntax, consider the header for the `rg` constructor:

```
(define/contract (make-rg nts sigma delta start
                          #:accepts [accepts '()]
                          #:rejects [rejects '()])
```

The contract stanzas for nonterminals, production rules, and list of words not in the language may be outlined as follows:

```
[nts (and/c ...)]
[delta (nts sigma start) (and/c ...)]
#:rejects
  [rejects (states sigma delta start) (and/c ...)]
```

As done for machines, `and/c` combines contracts for each property tested. Recall that `delta` depends on `start` given that only `start` may generate ϵ in a `rg` and that `#:rejects` depends on all required arguments because the grammar must be constructed to test words.

4.2 Ordering of Errors and Design Recipe Steps

Observe that the ordering of the stanzas in contracts for machine and grammar constructors guarantee that errors for required arguments are thrown before optional arguments. For the required arguments, further observe, that errors for arguments without dependencies are thrown before errors for arguments with dependencies and that errors for arguments with dependencies are only thrown if no errors are detected with arguments that do not have dependencies. Finally, the last error thrown is for testing after no errors are detected in all other arguments.

This framework does not always report errors in the same order that corresponds to the order in which the steps of the design recipe are listed. For example, testing errors for machines, related to step 2 of the design recipe for state machines, are thrown after errors for arguments with dependencies that are always associated with a latter design recipe step. Unfortunately, this is unavoidable given that it is impossible to report testing failures until the machine is successfully constructed. To the extent possible, however, an effort is made to report errors detected following the order of the arguments expected by the constructors.

4.3 Contract Sharing

The contracts for several constructor arguments have checks in common. For instance, machine-constructor contracts need to determine that the arguments given for the states, the alphabet, the final states, the rules, and the stack alphabet have no duplicates. Similarly, grammar-constructor contracts need to determine this for the arguments given for the nonterminals, the input alphabet, and the production rules. In addition, the step of the design recipe that is not successfully completed depends on the component tested:

states	Step 3	**machine alphabet**	Step 1
final states	Step 3	**machine rules**	Step 4
nonterminals	Step 2	**grammar alphabet**	Step 1
stack alphabet	Step 1	**grammar rules**	Step 3

Instead of writing a tailor-made contract to test for no duplicates for each of these arguments, contracts may be shared by abstracting over the argument tested and the step of the design recipe not successfully completed.

To illustrate the abstraction, consider the implementation displayed in Figure 7 to test for no duplicates. The contract takes as input the component tested, `type`, and a design recipe step, `step`, that is unsuccessfully completed upon failure. The error message is formatted using `step`, the values duplicated (i.e., the value returned by `return-duplicates`), and `type`. We note that `Racket`'s contract system automatically highlights the code where the error occurs.

4.4 Contracts without Dependencies

These contracts test properties of the given argument without regard for other arguments. For instance, the argument provided for the nonterminals of a grammar is tested to determine that the argument is a list, that each element is valid (as defined in Section 2.2.2), and that it contains no duplicates. If any of these conditions are not met then the programmer is informed that step 2 of the design recipe for grammars has not been successfully completed. The contract stanza for nonterminals in a grammar contract is implemented using `and/c` as follows:

```
(define (no-duplicates/c type step)
 (make-flat-contract
  #:name
   (string->symbol (format "distinct-list-of-~a" type))
  #:first-order (λ (vals) (not (check-duplicates vals)))
  #:projection
   (λ (blame)
    (λ (vals)
     (current-blame-format format-error)
     (if (not (check-duplicates vals))
         vals
         (raise-blame-error
          blame
          vals
          (format "Step ~a of the design recipe has not
                   been successfully completed. The
                   following values, ~a, are duplicated
                   in the given ~a"
                   step
                   (return-duplicates vals)
                   type))))))
```

Figure 7: The shared contract to determine no duplicates.

```
[nts (and/c (is-a-list/c "nonterminals" "two")
            (valid-listof/c
              valid-nt?
              "nonterminal"
              "list of nonterminals"
              #:rule "two")
            (no-duplicates/c "nonterminals" "two"))]
```

Three contracts are combined. The first determines if the given argument is a list and on failure constructs a message indicating that the argument for the nonterminals must be a list and that step 2 of the design recipe has not been successfully completed. The second determines that the argument is a valid list of nonterminals. The predicate `valid-nt?` is used to determine if a given list element is a valid nonterminal. Finally, the third, as described in Section 4.3, determines if there are no duplicates. The choice to use a string to indicate the unsuccessfully completed design recipe step upon failure comes from our classroom experience. Students consistently expressed that the use of a string, instead of a number, makes the error message clearer[5].

To illustrate a debugging session related to arguments without dependencies, consider the misuse of `make-rg`:

```
(define abstar (make-rg '(S a S) '(a B)
                        '((S -> ϵ) (S -> aA) (A -> bS))
                        'S)
```

Evaluation yields the following error message:

```
Step two of the design recipe has not been successfully
completed. The following: (a) are not valid nonterminals
in the given list of nonterminals: (S a S).
```

[5] Admittedly, we are not sure why. We speculate that since the design recipe is English prose, using words brings a student's mind back to thinking about the text in the error message instead of code.

The reader can appreciate that the message is succinct, includes the unsuccessfully completed design recipe step, is not prescriptive, and reports a single error. Upon reading the message, the student changes the lowercase a in the list of nonterminals to uppercase. Reevaluation generates the following error message:

```
Step two of the design recipe has not been successfully
completed. The following values, (S), are duplicated in
the given nonterminals: (S A S).
```

Once again, the message focuses the student on a single error. The student deletes the duplicate S and upon reevaluation gets:

```
Step one of the design recipe has not been successfully
completed. The following: (B) are not valid lowercase
alphabet letters in the given input alphabet: (a B).
```

Upon changing B to lowercase in the alphabet argument, the grammar is successfully constructed.

4.5 Contracts with Dependencies

These contracts test properties of the given argument using arguments with no dependencies that have been vetted for their required properties. For instance, the argument provided for pda transition rules is tested to determine that it is a list, that each rule has the right structure, that each rule only refers to elements in the arguments provided for the states, the input alphabet, and the stack alphabet, and that there are no duplicated rules. If any of these conditions fail then step 4 of the design recipe for state machines has not been successfully completed. The contract stanza for pda rules is:

```
[rules (states sigma gamma)
  (and/c (is-a-list/c "machine rules" "four")
         correct-ndpda-rule-structures/c
         (correct-ndpda-rules/c states sigma gamma)
         (no-duplicates/c "rules" "four"))]
```

The first contract determines if the argument is a list and is given two strings for the component and the recipe step not successfully completed. The second checks if the rule structure is correct (i.e., a list containing a sublist of length 3 and a sublist of length two). This contract is only used to test pda rules and is not shared and, therefore, has no inputs. The third determines if each rule properly refers only to the states, the input alphabet, and the stack alphabet. This contract is only used for pda rules and, therefore, does not require input to identify the component nor the design recipe step not successfully completed upon failure. Finally, the fourth contract determines if there are no duplicated rules.

To illustrate a student debugging session consider:

```
(define wcw^r
  (make-ndpda '(S P Q F) '(a b c) '(a b) 's '(F)
              '(((S ϵ ϵ) (P ϵ))   ((P a ϵ) (P (a)))
                ((P b ϵ) (P (b))) ((P c) (Q ϵ))
                ((Q a (a)) (Q ϵ)) ((Q v (b)) (q ϵ))
                ((Q ϵ ϵ) (F ϵ)))))
```

Evaluation yields the following error message:

```
Step three of the design recipe has not been
successfully completed. The given starting state:
s is not a valid state.
```

Upon capitalizing `s` and reevaluating, the generated error message is:

```
Step four of the design recipe has not been
successfully  completed. The following rules have
structural errors:
   Rule ((P c) (Q ϵ)):
    The first part of the rule, (P c), does not have
    the correct structure. It  must be a list with
    three elements.
```

The student has omitted the list of elements to pop from the stack. In this instance, the student does not want to pop anything off the stack. Upon changing `(P c)` to `(P c ())` and reevaluating, the following error message is obtained:

```
Step four of the design recipe has not been
successfully completed. The following rules have
structural  errors:
  Rule ((Q v (b)) (q ϵ)):
   The first element in the second part of the rule, q,
   is not a valid state.
```

Reading the error message makes the student realize that `q` must be capitalized. Reevaluating produces the following error:

```
Step four of the design recipe has not been
successfully completed. The following rules have errors,
which make them invalid: Rule ((Q v (b)) (Q ϵ)): v is
not in the given input alphabet.
```

The student realizes the typo: `v` instead of `b`. Upon fixing it, the pda is successfully constructed.

5 Incorporating Testing into `FSM` Contracts

To assist students and instructors to more easily validate machines and grammars, contracts are leveraged to easily test the expected result for given words. In addition, when these testing facilities are used the result is a validated machine/grammar. The syntax for machine/grammar constructors allows for two optional keyword parameters: a list of words that ought to be rejected/not-generated (the rejects list) and a list of words that ought to be accepted/generated (the accepts list). The contracts for these arguments are listed last and, therefore, only report an error, if any, after the contracts for all other machine/grammar components are satisfied and the machine/grammar may safely be built. This approach results in three advantages. First, it cuts down on the amount of coding required by the programmers as they do not have to write boilerplate testing code for each word (but still may if they so desire). Second, the user gets one error message listing the words that do not produce the expected result instead of multiple test failure reports (i.e., one for each word that causes a test to fail). Third, it simplifies how instructors may use their own test suites during grading by simply plugging in their own accepts and rejects lists.

The contracts for these optional arguments follow the same general design based on three (sub)contracts. The first tests that the argument is a list of words and takes as input a string identifying if the tests are for `accepts` or for `rejects`. The second tests that the given words only contains elements in `sigma`. It is given the input alphabet argument and, to build the error message upon failure, a symbol to identify the list tested. In this manner, students are explicitly warned about invalid testing words. The third takes as input all the arguments needed to build the machine/grammar and a Boolean to identify if the words ought to be accepted/generated or not. This contract builds the machine/grammar and accumulates words that do not produce the expected result to build the error message. For example, the `#accepts` and `#rejects` contract for `make-ndfa` are implemented as follows:

```
(define ALOM
  (make-ndfa
    '(S A B C) '(a b c) 'S '(A B C)
    '((S ϵ A) (S ϵ B)  (S ϵ C)  (A b A)  (A c A)
      (B a B) (B c B)  (C a C)  (C b C))
    #:accepts '((a a c) (a b c) (b a b b c) (b x))
    #:rejects '((c a a) (s b))))
```

Figure 8: An `ndfa` for words missing at least on of `(a b c)`.

```
(#:accepts [accepts (states sigma start finals rules)
  (and/c
    (listof-words/c "accepts")
    (words-in-sigma/c sigma 'accepts)
    (ndfa-input/c states sigma start finals rules #t))]
#:rejects [rejects (states sigma start finals rules)
  (and/c
    (listof-words/c "rejects")
    (words-in-sigma/c sigma 'rejects)
    (ndfa-input/c states sigma start finals rules #f))]
```

To illustrate a debugging session around testing lists, let Σ=`(a b c)` and consider the `ndfa` displayed in Figure 8 to decide the language containing all words that have at least one input alphabet letter missing. The first error message generated is:

```
Step two of the design recipe has not been
successfully completed. The following words
in the accepts list contain symbols not
included in sigma: ((b x)).
```

The student corrects the typo by changing `x` to `c`. The next error message generated is:

```
Step six of the design recipe has not been
successfully completed. The constructed machine
does not accept the following words: ((a b c)
(b a b b c)).
```

Upon moving the two words that ought to be rejected to the rejects list the next error generated is:

```
Step two of the design recipe has not been
successfully completed. The following words in
the reject list contain symbols not included
in sigma: ((s b)).
```

Here the student mistyped `s` instead of `a`. Upon correcting the typo, the error message generated is:

```
Step six of the design recipe has not been
successfully completed. The constructed machine
does not reject the following words: ((c a a)
(a b)).
```

	Q1	Q2	Q3	Q4	Q5	Q6
μ	3.8	3.5	3.5	3.7	3.9	3.9
eta	4	4	4	4	4	4
Mo	4	4	4	4	4	4

Table 1: Descriptive Statistics.

The student now sees that the two identified words clearly ought to be accepted and moves them to the accepts list. Upon reevaluation the machine is successfully constructed validated.

6 Empirical Data

This section presents empirical data collected from students at Seton Hall University and Worcester Polytechnic Institute over three offerings of the course: 2 at Seton Hall University (Spring 2024 and Spring 2025) and 1 at Worcester Polytechnic Institute (Spring 2024). The courses enrolled 117 students and 63 volunteered to participate in an anonymous survey. Seventeen respondents identified as female, forty-three as male, one as nonbinary, one as agender, and one did not respond this question. The age of all respondents is between 18 and 23. All respondents were taking their first FLAT course and, with the exception of one pursuing a Mathematics degree, are majoring in Computer Science. None of the respondents received any benefit or compensation for participating in the study.

The survey includes questions measuring student perceptions about the debugging usefulness and properties of RBEs. Students indicate agreement with a statement using a Likert scale [36]: `[1] Strongly diasgree...Strongly agree [5]`, including, `[3]`, a neutral response option. The survey includes the following six questions:

(1) Q1 Recipe-based errors are useful in debugging
(2) Q2 Recipe-based errors are clear
(3) Q3 Recipe-based errors are informative
(4) Q4 Recipe-based errors cut down my debugging time
(5) Q5 The length of recipe-based errors is appropriate
(6) Q6 Associating a step of the design recipe with an error message is useful

In the presentation of the data, we describe responses 4 and 5 as "tend to agree" and responses 1 and 2 and "tend to disagree."

The distribution of responses is displayed in Figure 9 and summary descriptive statistics, mean (μ), median (η), and mode (Mo), for each question are displayed in Table 1. For all questions, we observe a fairly normal distribution (i.e., $\mu \approx \eta$=Mo) indicating that responses around the mean are more frequent that responses away from the mean. All distributions are slightly negatively skewed indicating that the majority of responses are above the mean (i.e., responses 4 and 5).

For Q1, we observe that a majority of respondents, 63%, tend to agree that RBEs are useful in debugging. A small minority, 10%, tends to disagree, but not strongly. This suggests that the primary goal of RBEs, to be useful in the debugging process, has been achieved. A little over a quarter of the respondents (27%), however, feel neutral about this statement. A cohort of neutral students is always expected for many different reasons. For example, there are students that ignore the steps of the design recipe, students that rather consult the documentation, and students that make no effort to debug their designs. Nonetheless, we feel that there is likely more that may be done to reduce the neutrality cohort. For example, having one or more lectures whose sole purpose is to illustrate how to debug using RBEs may help make RBEs useful for an even larger cohort of students.

The responses for Q2–Q5 shed some light on why RBEs are useful in debugging. A majority tends to agree that RBEs are clear (Q2, 55%) and informative (Q3, 51%)). Thus, suggesting that students understand the error messages. A majority finding that RBEs cut down on debugging time (Q4, 59%) suggests that RBEs effectively hone students onto the cause of a bug. Finally, a majority responding that the length of RBEs is appropriate suggests that the messages are succinct, which encourages students to read them.

For Q6, we observe that a majority of respondents, 66%, tend to agree that associating a design recipe step with an error is useful. We also observe that a very small minority, 13%, tends to disagree. This is a very encouraging result, because it suggests that the design recipe step included in an RBE helps students in their debugging efforts. It is also noteworthy that at 21% this question exhibits the smallest neutral cohort among all survey questions. Thus, suggesting that students in general value the information that a design recipe step conveys.

7 Related Work

There is evidence in the literature that difficult-to-read error messages impact a programmer's performance [2]. Despite decades of research on error messages systems, a 2019 ITiCSE working group report found that error messages continued to be described as inadequate, useless, frustrating, cryptic, confusing, and a barrier to progress [4]. They recommended a set of guidelines for developers of error messaging systems that include making error messages comprehensible using a positive tone and providing relevant context to help understand the error. FSM's error messaging system has adopted these recommendations and introduces a new dimension in the context to understand the cause of an error: a failed design recipe step. A two-fold approach is taken to make error messages comprehensible using a polite tone. On the one hand, FSM error messages match the language found in the textbook used for instruction and purposely avoid faulting, blaming, or demeaning the programmer as suggested by the literature [10, 14, 42, 66]. On the other hand, instructors are encouraged to use in the classroom the vocabulary found in the textbook for instruction and in FSM error messages. To provide further relevant context to help understand the error, FSM error messages highlight the location of the error and provide information about the values and the types related to the error as suggested by the literature [20].

The negative consequences on novices stemming from error messages that are not easily comprehended are well-documented [28, 32–34, 40]. Messages that are not easily comprehended also have consequences for instructors that must invest time and effort explaining them to enable students to correct their mistakes [12, 23, 62]. To address this problem several error-messaging systems offer more informative and/or enhanced error messages [12, 17, 31, 59, 65]. For example, Rust [30, 39] offers extended error explanations

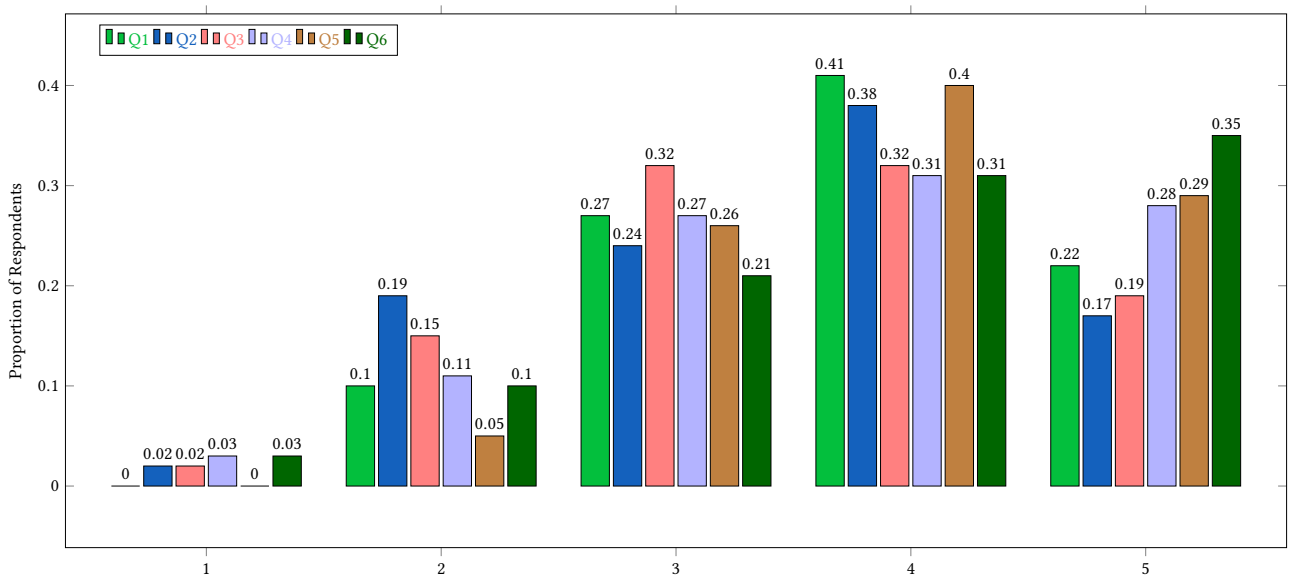


Figure 9: Data distribution for survey questions.

by including unique *error codes* in error messages that may be consulted by the programmer [64]. On the other hand, `Elm` [13, 61] and `Clang` [63] offer helpful hints to fix errors. In contrast, `FSM` enhances error messages by associating them with a design recipe step.

Marceau et al. first suggested that making error messages comprehensible goes beyond what is generated by a compiler or an interpreter [38]. They suggest to treat error messages as an integral part of course design, to make error messages nonprescriptive, and to phrase error messages using the vocabulary employed in the classroom. The work done with the `FSM` error messaging system expands on this idea by putting forth that the language employed in error messages and in the classroom ought to match the language in the textbook used for instruction and by including the step of the design recipe not successfully completed.

8 Discussion

We discuss the work's implications and lessons learned. The first two subsections specifically address our research questions. The remaining subsections address openness towards non-mainstream programming languages and study limitations.

8.1 RBEs and the Debugging Interaction Loop

We answer **RQ1**, *Are* RBEs *useful in debugging?*, based on the results obtained for Q1–Q6. Majorities tend to agree with all questions and these majorities are substantially larger than the dissenting minorities. Thus, our answer to **RQ1** is a strong affirmative answer supported by the context provided by the results obtained by Q2–Q6.

Though further research is needed to confirm *how* RBEs are useful. We hypothesize that RBEs are useful through the debugging interaction loop:

```
while (code throws an RBE)
  read RBE
  revise code
```

Code revisions present students with a huge range of potential actions to choose from. That is, if an unguided code revision process is followed then there is an arbitrary number of code editions a student may perform. We observe this every semester. Students get stuck when confronted with a large action space. This is especially true of students that ignore following the steps of the design recipe. Such students expend long periods of time on undirected experimentation (a.k.a. hacking or tinkering). Reading the RBE is a *feedback* step, software's opportunity to assess the user's past actions in an effort to guide future user actions during code revision. Thus, we explain the key intended benefit of RBEs as a feedback mechanism: **RBEs scaffold the broad action space of programming around a much narrower action space, the steps of the design recipe**. When students are stuck, software and instructor alike can encourage them to revisit a specific design recipe step and address a specific issue within it, simplifying their search space for a correct solution. The ability of RBEs to provide this encouragement in real time is critical to its success: a study using a previous-generation design-recipe approach reported that students typically did not return to earlier design recipe steps during debugging in the absence of such encouragement [11]. RBEs directly offer such encouragement by highlighting a design recipe step unsuccessfully completed. Furthermore, RBEs are a specific instance of a broader strategy within error message research: automatically hinting programmers toward potential solutions [4].

8.2 RBEs are a Key Quality Factor

RQ2, *Why are RBEs useful?*, is directly addressed by Q2–Q6. Majorities agree with all questions, substantially more than those who

disagree. Neutral answers are substantial, but less common than agreeing answers. Notably, the most agreed survey question is Q5 addressing RBEs' length appropriateness. This result is noteworthy because message length has been consistently identified as an important challenge in the literature [4, 5, 15, 26, 27, 31, 35, 58]. The second most agreed survey question is Q6 assessing the usefulness of design recipe steps in RBEs. Thus, indicating that design recipe step numbers are critical to the debugging interaction loop. Our answer to **RQ2** is that RBEs are useful, because students find them clear, informative, and of the proper length as well as a factor in reducing debugging time.

We further reflect on perceived usefulness to answer, "*Is a design recipe step useful in an error message?*" **RQ3**. In our context, usefulness is fundamentally tied to pedagogy. The importance of message vocabulary choice is well-cited [4], including alignment with classroom vocabulary. We ensure our classroom vocabulary is consistent with FSM, which requires instructor familiarity with FSM. Marceau argues that error message design is essential to curriculum design [38]. In this light, we see RBEs as a first step toward *strengthening integration between the debugging loop and the program-by-design methodology as a whole.* When a student sees a design recipe step number in a message, this number has meaning in their textbooks, lectures, interactions with course staff, and mental models. Our goal is that once a student identifies which step has an issue, they can review classroom examples to understand how the same step was approached for different applications. RBEs aim to encourage their reader to retrieve their knowledge about a step, which they learned from the textbook or from the lectures. Therefore, we offer a strong affirmative answer for **RQ3**.

8.3 Openness to Less-Used Languages

The work described is part of a larger programming-based effort to develop a new curriculum approach for FLAT courses. Throughout this larger project, language choice has been a common topic, with potential instructors asking whether students are discouraged by the use of a domain-specific language embedded in Racket. Such questions are appropriate, as students in one Racket-based introductory programming class cited the language choice as a primary source of discontent [8].

Our survey results suggest high approval of the Racket-based FSM language in the context of a FLAT course (e.g., error message quality is rated favorably). We hypothesize that this approval is explained by differences in context. At both institutions, introductory-level Racket courses are prerequisite to the FLAT course, meaning that students have familiarity with Racket as a language. Familiarity with existing tools is known to be a decision-making factor for programmers [41], so we designed our questions with familiarity bias in mind. For example, we asked students about a specific FSM feature which is not present in vanilla Racket. Familiarity may still play a subtler role: because our students have already become familiar with Racket, they need not repeat the growing pains of first-time Racket programmers and, thus, have greater opportunity to engage with FSM's specifics as intended. During instructor-student interactions, several students proposed another theme: their languages preferences differed between courses. Though these specific students perceived Racket to be detached from programming tasks in industry, they also perceived FLAT courses as theory-focused. When their course goal was to strengthen theoretical understanding instead of building software, they were open to the use of a programming language which they also viewed as abstract.

8.4 Limitations

A notable limitation is that our evaluation is not a controlled trial (i.e., we do not divide into control and test groups). Though this limitation is nontrivial, it is relatively common in education-related research. Classrooms cannot be controlled like laboratory environments, varying greatly depending on instructor, students, and environment. Furthermore, most universities, including ours, are unlikely to approve offering differing quality of instruction among sections of a course for the purpose of conducting a study. Moreover, our questions were part of a larger survey studying the FSM teaching approach overall for the first time. In this context, where RBEs are tightly integrated with FSM pedagogy, creating control and test groups would effectively require running two parallel, substantially different courses.

Tight integration between course software and pedagogy is itself a limitation. Software needs maintenance over time. We reduce this concern by building on the Racket language, whose DrRacket [20] teaching environment is historically well-maintained. The effects of RBEs are unclear outside classroom environments, but we argue this limitation is appropriate. Scaffolding builds expertise; as expertise matures, programmers become more effective at debugging without scaffolds [56]. Likewise, a study of programmer sense of belonging found scaffolded errors mattered most at the novice level [16].

Our evaluation is exclusively quantitative. Though this approach scales well to large class sizes, it leaves out the details of individual experiences. In our classroom experience, students are often able to vocalize their experiences in narrative form, highlighting specific moments of success or struggle. Future work can use qualitative methods to elicit narrative student experiences, drawing out details beyond our quantitative data.

Our evaluation considers specific student populations at specific moments and places, leading to diversity limitations. Our two institutions provide a diversity of academic background. One is a technical university where Computer Science is the largest major. The other is a liberal arts university where Computer Science is a medium-sized major. However, both institutions are located in the same geographic region and Computer Science students are mostly male European Americans. Thus, our sample population may not be representative of the broader and diverse Computer Science student population.

9 Concluding Remarks

This article puts forth a novel error messaging system that encourages thinking of errors in terms of design. The error messaging system is implemented as part of FSM–a functional DSL developed for the Formal Languages and Automata Theory classroom. It integrates the steps of the design recipe with which students learn to design and implement state machines and grammars, and integrates unit tests–an essential step in the design process–to generate error messages. Empirical data collected from two US-based universities suggests that students find that the generated messages

clear, useful, succinct, and successfully help reduce debugging time. Furthermore, the data suggests that students find the association of design recipe steps with error messages useful. In summary, the novel error messaging system is well-received by students. In part, we attribute the success of the system to using the same language in the classroom, in the textbook of instruction, and in the error messages.

Future work includes performing more empirical studies on the impact of including testing as part of the error messaging system and on the impact the error messaging system may have on female and on minority students. Future work also includes extending the work presented to the construction of multitape Turing machines.

References

[1] Sanaa M. Algaraibeh. 2022. Techniques for Enhancing Compiler Error Messages. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2* (Lugano and Virtual Event, Switzerland) *(ICER '22)*. Association for Computing Machinery, New York, NY, USA, 1–2. doi:10.1145/3501709.3544292

[2] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE, New York City, USA, 575–585. doi:10.1109/ICSE.2017.59

[3] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 126–131. doi:10.1145/2839509.2844584

[4] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. doi:10.1145/3344429.3372508

[5] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective Compiler Error Message Enhancement for Novice Programming Students. *Computer Science Education* 26, 2-3 (2016), 148–175. doi:10.1080/08993408.2016.1225464

[6] Brett A. Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. 2018. Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 634–639. doi:10.1145/3159450.3159453

[7] Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Michael Sperber, Marcus Crestani, Herbert Klaeren, and Eric Knauel. 2008. HtDP and DMDA in the Battlefield: A Case Study in First-Year Programming Instruction. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education* (Victoria, BC, Canada) *(FDPE '08)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/1411260.1411262

[8] Rose Bohrer. 2023. Centering Humans in the Programming Languages Classroom: Building a Text for the Next Generation. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E, SPLASH-E 2023, Cascais, Portugal, 25 October 2023*, Molly Q. Feldman and Michael Hilton (Eds.). ACM, 26–37. doi:10.1145/3622780.3623646

[9] P. J. Brown. 1983. Error Messages: The Neglected Area of the Man/Machine Interface. *Commun. ACM* 26, 4 (apr 1983), 246–249. doi:10.1145/2163.358083

[10] Andrew Buxton and Lesley Trenner. 1987. An Experiment to Assess the Friendliness of Error Messages from Interactive Information Retrieval Systems. *J. Inf. Sci.* 13, 4 (jul 1987), 197–209. doi:10.1177/016555158701300403

[11] Francisco Enrique Vicente Castro and Kathi Fisler. 2020. Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE 2020, Portland, OR, USA, March 11-14, 2020*, Jian Zhang, Mark Sherriff, Sarah Heckman, Pamela A. Cutter, and Alvaro E. Monge (Eds.). ACM, 487–493. doi:10.1145/3328778.3366847

[12] Natalie J Coull. 2008. *SNOOPIE: Development of a Learning Support Tool for Novice Programmers within a Conceptual Framework*. Ph.D. University of St Andrews, St Andrews, Scotland. http://hdl.handle.net/10023/522.

[13] Evan Czaplicki. 2015. Compiler Errors for Humans. https://elm-lang.org/news/compiler-errors-for-humans. Last accessed: June 2025.

[14] Carla De Lira. 2017. Improving the Learning Experiences of First-Year Computer Science Students with Empathetic IDEs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 293–294. doi:10.1145/3105726.3105742

[15] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 55, 15 pages. doi:10.1145/3411764.3445696

[16] Maja Dornbusch and Jan Vahrenhold. 2024. "In the Beginning, I Couldn't Necessarily Do Anything With It": Links Between Compiler Error Messages and Sense of Belonging. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1, ICER 2024, Melbourne, VIC, Australia, August 13-15, 2024*, Paul Denny, Leo Porter, Margaret Hamilton, and Briana B. Morrison (Eds.). ACM, 14–26. doi:10.1145/3632620.3671105

[17] Thomas Dy and Ma. Mercedes Rodrigo. 2010. A Detector for Non-Literal Java Errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 118–122. doi:10.1145/1930464.1930485

[18] Joseph Eremondi, Wouter Swierstra, and Jurriaan Hage. 2019. A Framework for Improving Error Messages in Dependently-Typed Languages. *Open Computer Science* 9, 1 (2019), 1–32. doi:10.1515/comp-2019-0001

[19] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (Second ed.). MIT Press, Cambridge, MA, USA.

[20] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A Programming Environment for Scheme. *J. Funct. Program.* 12, 2 (mar 2002), 159–182. doi:10.1017/S0956796801004208

[21] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 35–42. doi:10.1145/2632320.2632346

[22] Matthew Flatt, Robert Bruce Findler, and PLT. 2024. *The Racket Reference*. PLT. https://docs.racket-lang.org/reference/index.html Last accessed: June 2025.

[23] T. Flowers, C.A. Carver, and J. Jackson. 2004. Empowering Students and Building Confidence in Novice Programmers through Gauntlet. In *34th Annual Frontiers in Education, 2004. FIE 2004.* (Savannah, GA, USA). IEEE, New York City, USA, T3H/10–T3H/13 Vol. 1. doi:10.1109/FIE.2004.1408551

[24] Philip J. Guo. 2018. Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3173574.3173970

[25] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1019–1028. doi:10.1145/1753326.1753478

[26] Bart Heemskerk. 2020. *The Error that is the Error Message: Comparing Information Expectations of Novice Programmers Against the Information in Python Error Messages*. Master's thesis. Delft University of Technology, Delft, The Netherlands.

[27] C. D. Hundhausen, D. M. Olivares, and A. S. Carter. 2017. IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda. *ACM Trans. Comput. Educ.* 17, 3, Article 11 (aug 2017), 26 pages. doi:10.1145/3105759

[28] Matthew C. Jadud. 2006. *An Exploration of Novice Compilation Behaviour in BlueJ*. Ph.D. Thesis. University of Kent at Canterbury, Cantebury, UK.

[29] Caitlin Kelleher, Dennis Cosgrove, and David Culyba. 2002. Alice2: Programming Without Syntax Errors. In *User Interface Software and Technology - UIST 2002* (Paris, France), Vol. 2. ACM, ACM, New York, NY, USA, 35–36. https://uist.acm.org/archive/adjunct/2002/pdf/demos/p35-kelleher.pdf.

[30] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

[31] Tobias Kohn. 2019. The Error Behind The Message: Finding the Cause of Error Messages in Python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 524–530. doi:10.1145/3287324.3287381

[32] Sarah K. Kummerfeld and Judy Kay. 2003. The Neglected Battle Fields of Syntax Errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20* (Adelaide, Australia) *(ACE '03)*. Australian Computer Society, Inc., AUS, 105–111.

[33] Thomas E. Kurtz. 1978. BASIC. *SIGPLAN Not.* 13, 8 (aug 1978), 103–118. doi:10.1145/960118.808376

[34] Michael J. Lee and Amy J. Ko. 2011. Personifying Programming Tool Feedback Improves Novice Programmers' Learning. In *Proceedings of the Seventh International Workshop on Computing Education Research* (Providence, Rhode Island, USA) *(ICER '11)*. Association for Computing Machinery, New York, NY, USA, 109–116. doi:10.1145/2016911.2016934

[35] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using Large Language Models to Enhance Programming Error Messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 563–569. doi:10.1145/3545945.3569770

[36] Rensis Likert. 1932. A Technique for the Measurement of Attitudes. *Archives of Psychology* 140 (1932), 1–55.

[37] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106. doi:10.1145/3293881.3295779

[38] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2011)*. Association for Computing Machinery, New York, NY, USA, 3–18. doi:10.1145/2048237.2048241

[39] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. doi:10.1145/2663171.2663188

[40] Davin McCall. 2016. *Novice Programmer Errors - Analysis and Diagnostics*. Ph. D. Dissertation. University of Kent,. https://kar.kent.ac.uk/61340/

[41] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical Analysis of Programming Language Adoption. *SIGPLAN Not.* 48, 10 (oct 2013), 1–18. doi:10.1145/2544173.2509515

[42] Rolf Molich and Jakob Nielsen. 1990. Improving a Human-Computer Dialogue. *Commun. ACM* 33, 3 (mar 1990), 338–348. doi:10.1145/77481.77486

[43] Marco Morazán. 2023. Composing Turing Machines in FSM. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E* (Cascais, Portugal) *(SPLASH-E 2023)*. Association for Computing Machinery, New York, NY, USA, 38–49. doi:10.1145/3622780.3623647

[44] Marco T. Morazán. 2022. *Animated Problem Solving - An Introduction to Program Design Using Video Game Development*. Springer, Cham, Switzerland. doi:10.1007/978-3-030-85091-3

[45] Marco T. Morazán. 2022. *Animated Program Design - Intermediate Program Design Using Video Game Development*. Springer, Cham, Switzerland. doi:10.1007/978-3-031-04317-8

[46] Marco T. Morazán. 2024. *Programming-Based Formal Languages and Automata Theory - Design, Implement, Validate, and Prove*. Springer, Cham, Switzerland. doi:10.1007/978-3-031-43973-5

[47] Marco T. Morazán and Oliwia Kempinski. 2024. Using Computation Graphs to Explain Nondeterminism to Students. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E* (Pasadena, CA, USA) *(SPLASH-E '24)*. Association for Computing Machinery, New York, NY, USA, 23–33. doi:10.1145/3689493.3689978

[48] Marco T. Morazán and Tijana Minić. 2024. Finite-State Automaton To/From Regular Expression Visualization. *Electronic Proceedings in Theoretical Computer Science* 405 (July 2024), 36–55. doi:10.4204/eptcs.405.3

[49] Marco T. Morazán and Tijana Minić. 2024. Nondeterministic to Deterministic Finite-State Machine Visualization: Implementation and Evaluation. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 262–268. doi:10.1145/3649217.3653641

[50] Marco T. Morazán, Tijana Minić, and Oliwia Kempinski. 2024. Visualizing Composed Turing Machines. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E* (Pasadena, CA, USA) *(SPLASH-E '24)*. Association for Computing Machinery, New York, NY, USA, 34–44. doi:10.1145/3689493.3689979

[51] Marco T. Morazán, Joshua M. Schappel, and Sachin Mahashabde. 2020. Visual Designing and Debugging of Deterministic Finite-State Machines in FSM. *Electronic Proceedings in Theoretical Computer Science* 321 (August 2020), 55–77. doi:10.4204/eptcs.321.4

[52] Marco T. Morazán. 2025. Grammar Verification for Students: A Grammar Design Recipe with Verification Steps. In *Proceedings of the 17th International Conference on Computer Supported Education - Volume 2: CSEDU*. INSTICC, SciTePress, 662–669. doi:10.5220/0013216500003932

[53] Marco T. Morazán and Rosario Antunez. 2014. Functional Automata - Formal Languages for Computer Science Students. *Electronic Proceedings in Theoretical Computer Science* 170 (Dec. 2014), 19–32. doi:10.4204/eptcs.170.2

[54] P. G. Moulton and M. E. Muller. 1967. DITRAN—A Compiler Emphasizing Diagnostics. *Commun. ACM* 10, 1 (jan 1967), 45–52. doi:10.1145/363018.363060

[55] Jonathan P. Munson and Elizabeth A. Schilling. 2016. Analyzing Novice Programmers' Response to Compiler Error Messages. *J. Comput. Sci. Coll.* 31, 3 (jan 2016), 53–61. doi:10.5555/2835377.2835386

[56] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2008, Portland, OR, USA, March 12-15, 2008*, J. D. Dougherty, Susan H. Rodger, Sue Fitzgerald, and Mark Guzdial (Eds.). ACM, 163–167. doi:10.1145/1352135.1352191

[57] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler Error Messages: What Can Help Novices?. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 168–172. doi:10.1145/1352135.1352192

[58] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 74–82. doi:10.1145/3105726.3106169

[59] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (oct 2017), 24 pages. doi:10.1145/3077618

[60] Brian Randell and Lawford J. Russell. 1964. *Algol-60 Implementation*. Academic Press, Inc., USA.

[61] R.H. Richens. 1983. *Elm*. Cambridge University Press, Cambridge, UK.

[62] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4, Article 19 (nov 2013), 40 pages. doi:10.1145/2534973

[63] The Clang Team. 2024. Clang: A C Language Family Frontend for LLVM. https://clang.llvm.org/. Last accessed: June 2025.

[64] The Rust Team. 2024. *Rust Compiler Development Guide*. Rust-Lang. Last accessed: June 2025.

[65] Suzanne Marie Thompson. 2006. *An Exploratory Study of Novice Programming Experiences and Errors*. Master. University of Victoria, Victoria, BC, Canada.

[66] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Adv. in Hum.-Comp. Int.* 2010, Article 3 (jan 2010), 26 pages. doi:10.1155/2010/602570

[67] Noel Welsh and Ryan Culpepper. 2024. *RackUnit: Unit Testing* (v8.12 ed.). PLT Racket. Last accessed: June 2025.

[68] John Wrenn and Shriram Krishnamurthi. 2017. Error Messages are Classifiers: A Process to Design and Evaluate Error Messages. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 134–147. doi:10.1145/3133850.3133862

Design and Implementation of DSLs for Unit Testing

Marco T. Morazán
Seton Hall University
South Orange, NJ, USA
morazanm@shu.edu

Andrés M. Garced
Seton Hall University
South Orange, NJ, USA
maldona2@shu.edu

Abstract

More intuitive abstractions for software development are made possible by domain-specific languages. Although the use of domain-specific languages has proliferated, the sound software practice of testing (programs written in a DSL) has not kept up with the pace of development. Many domain-specific languages do not provide syntax to express unit tests nor do they provide the infrastructure for test evaluation. This article presents a set of design principles to guide the development of a unit testing infrastructure within a domain-specific language. These principles embrace domain knowledge and put forth a much neglected aspect: the user evaluation of the offered abstractions. The design of a unit testing domain-specific language using these principles is outlined and its implementation is described. In addition, the results of a small formative study suggest that the developed domain-specific language for unit testing is well-received by its users.

CCS Concepts

• **Software and its engineering → Domain specific languages**;
• **Human-centered computing → Interaction design**;

Keywords

Domain-Specific Languages, Unit Testing, Implementation

ACM Reference Format:
Marco T. Morazán and Andrés M. Garced. 2025. Design and Implementation of DSLs for Unit Testing. In *Proceedings of the 37th Symposium on Implementation and Application of Functional Languages (IFL 2025), October 1–3, 2025, Montevideo, Uruguay*. ACM, New York, NY, USA, 12 pages.

1 Introduction

The beginning of the domain-specific languages (DSLs) era is usually attributed to to Peter J. Landin [37]. In his seminal paper, *The Next 700 Programming Languages* [16], Landin argues for a diversity of languages to service a diversity of application areas. Since Landin, the number of DSLs has mushroomed and they have become a common tool for a myriad of domains. For example, GraphViz's DOT language [10] is a DSL for representing abstract graphs, HTML [33] and CCS [25] are DSLs for web page design, SQL [6] is a DSL for managing data in a relational database, iTask [30] is a DSL for task-oriented programming, funQ [2] is a DSL for functional quantum programming, and FitNesse is a DSL for test-driven development in a .NET environment. In fact, the cited DSLs are only a minuscule part of the DSLs in existence today, which may be in the thousands [20].

Unlike a general-purpose programming language (GPL) that targets problem solving in all application domains, a DSL targets a specific application domain [7, 12, 24, 35–37]. Typically, this leads to gains in expressibility that reduce the distance between a problem as a domain expert understands it and a program [12]. In turn, this enables domain experts, that may not be programmers, to contribute to software customization. This is achieved by raising the abstraction level to capture commonalities in the problem domain that may be reused and added as concepts in a DSL [37] to more clearly specify solutions to problems and communicate the intent of a program [7, 13].

Given the level of abstraction a DSL provides, a great deal of domain logic is hidden in the implementation. Therefore, it is important to test its implementation [37]. Such testing is done mostly in two ways. In the first, unit tests may be written in the host GPL to validate an embedded DSL's implementation [36]. This approach likely excludes domain experts that are not programmers and programmers that are not familiar with the host GPL. In the second, unit tests are written in the DSL [36]. This approach adds some complexity to DSL implementation, because syntax and dynamic semantics need to support, respectively, expressing and executing unit tests. This approach is attractive, because it allows programmers that use the DSL to validate their programs through testing. Providing unit testing to DSL programmers, in part, inspires the work presented in this article.

Surprisingly, it is not uncommon for DSL implementations to fail to include unit testing facilities (e.g., GraphViz's DOT and HTML). Given that testing is an integral part of software development, we argue that DSLs need to provide syntax and dynamic semantics to express and execute unit tests. This must be done, however, in a manner that is meaningful and useful to programmers as well as to domain experts. Otherwise, it is unlikely to de adopted by either stake holders. To this end, this article makes the following contributions:

- Proposes design criteria for unit-testing DSLs
- Illustrates the unit testing interface satisfying the outlined criteria using a sample target DSL
- Outlines the implementation of the unit-testing DSL for the sample target DSL
- Presents the results of a formative human factors study evaluating the unit testing DSL

The article is organized as follows. Section 2 describes the proposed design criteria for unit-testing DSLs. Section 3 briefly describes the target DSL. Section 4 presents the interface for the unit-testing DSL for the target language. Section 5 outlines the implementation of the unit-testing DSL. Section 6 presents the empirical

IFL 2025, Montevideo, Uruguay

data collected from a small formative study to ascertain the usefulness of the unit-testing DSL. Section 7 compares and contrasts with related work. Finally, Section 8 presents concluding remarks and directions for future work.

2 Design Principles for Developing a Unit Testing DSL

Given that DSLs are used to develop software, programmers and domain experts ought to be able to perform testing. Inevitably, this requires designing and implementing a DSL for expressing tests. That is, unit tests have their own syntax and semantics and, thus, may be thought of as a domain-specific programming language.

Every DSL implementation may be divided into 5 core aspects [20, 36, 37]:

Concrete Syntax	Defines what programmers write and is typically described using a context-free grammar
Abstract Syntax	Defines the representation of a program in computer memory
Static Semantics	Defines the static properties of programs
Dynamic Semantics	Defines how programs are evaluated
Design Environment	Defines services provided to assist with program writing and refinement

Of these, the most relevant to domain experts and programmers using a given DSL are the concrete syntax, the static semantics, the dynamic semantics, and the design environment. Rarely, if ever, are the users of a DSL concerned with abstract syntax.

Conspicuously absent from the core aspects of DSL implementation outlined above is *evaluation*. The development of a DSL goes beyond technical achievements that work. Considering that the main purpose of a DSL is to offer expressibility and abstractions that ought to make program development easier for both programmers and non-programmers, it is necessary to conduct human factors studies to ascertain if the offered abstractions meet this goal. Thus, we add a sixth core aspect for DSL implementation:

User Evaluation	Determine user satisfaction and impressions

We further expand on these core aspects relevant to users with a focus on the development of a DSL to express unit tests.

The concrete syntax ought to make writing and reading unit test relatively easy for all DSL users. To this end, the concrete syntax ought to:

- use domain jargon, not GPL or generic jargon
- easily express domain abstractions
- be concise
- clearly communicate the purpose of a test to a domain expert

Such characteristics make a unit tests easy to write, read, and understand. As a consequence stake holders are likely to learn and adopt the language for unit tests. When a variety of unit testing categories are offered, a balance must be struck between language cacophony (i.e., the number of DSLs) and the size of a DSL (i.e., the DSL ought not become a GPL) [7].

The static semantics aid in correctly writing unit tests. In essence, constraints need to be defined that programs must conform to. These constraints may include:

- the correct writing of unit tests (e.g., the correct number of expressions are provided for testing)
- unit tests are properly typed

Such characteristics help ensure that unit testing "vocabulary" is properly used. It assists all stake holders in writing tests that are correct and prevent errors at interpretation/compile time.

The dynamic semantics aid in understanding how unit tests are evaluated. This includes the handling of errors. The characteristics that important for unit testing are:

- test evaluation needs to make sense to domain experts, not just programmers
- the reason for a failed test needs to have meaning to a domain expert
- failed test reports use domain jargon
- failed test reports should not prescribe solutions
- failed test reports are concise and succinct

Such characteristics help prevent DSL programmers from being overwhelmed, exhausted, or discouraged trying to pinpoint the reason a test fails. Like some error messaging systems (e.g., [21, 39]), failed test reports should not prescribe solutions. Prescribing solutions may lead stake holders down unfruitful refinement paths. In addition, careful attention to the length of failed tests reports is important to encourage programmers and domain experts to read them. If such reports are too long, a tendency to ignore them may take root.

The design environment ought to provide services that aid programmers and domain experts to correct failed unit tests. These services ought to present a low extraneous cognitive load to make them easy to understand. Some of the services the design environment may provide related to unit testing are:

- failed test location in the program
- failed test syntax highlighting in the program
- reinforce the failed test report as accurately as possible

Like done by error messaging systems that pinpoint where in the program an error occurs [1], pinpointing the location of failed tests assists programmers to quickly focus on debugging. Reinforcing the failed test report means that, for example, instead of highlighting an entire test, only the values that cause the test to fail are highlighted. This is possible, for instance, when the syntax allows to express multiple test values in a single unit test.

User evaluation is important, because it can help the evolution of a unit testing DSL. Human factors studies can help determine DSL abstractions that are considered useful and ought to be kept as well as abstractions that are difficult to use and need to be refined. Such studies may be performed through the use of surveys, interviews, and observational sessions.

3 Target DSL

To illustrate the implementation of a DSL for unit testing, FSM (Functional State Machines) [28] is used as the target language in which to embed such a DSL. The DSL developed for unit testing is referred to as FSMt (FSM testing language). FSM itself is a domain-specific language, embedded in Racket [5], for the Formal Languages and Automata Theory (FLAT) classroom. It is used to harness CS student interest in programming to make FLAT topics

more palatable. For the purposes of this article, we target the subset of FSM that is used to build state machines (i.e., deterministic finite-state machines (dfas), nondeterministic finite-state machines (ndfas), pushdown automata (pdas), Turing machines (tms), and multitape Turing machines (mttms)), and to build grammars (i.e., regular (rg), context-free (cfg), and unrestricted (urg)). Briefly, state machines are used to decide or semidecide languages[1]. Given a word, a machine either accepts or rejects. Grammars are used to derive words. Given a word, a grammar returns a derivation or indicates that the word is not in the grammar's language. In this article, we endeavor to use examples that are easy to follow. Readers seeking more details on these models on computation are referred to any standard FLAT textbook (e.g., [17, 19, 22, 27, 31, 32]).

Formerly, to write unit tests, FSM programmers needed to use RackUnit [38]–a unit testing framework for Racket. Specifically, tests were written using check-equal?, which requires two expressions to evaluate. By convention, the first is the expression to test and the second expression is for the expected value. If these are equal the tests passes and no failed test report is generated. Otherwise, the test fails and a failed test report is printed to the terminal.

To briefly illustrate programming in FSM, we present an example of building a state machine and an example of building a grammar. For each, unit testing is highlighted.

3.1 FSM State Machine Example

Figure 1a displays the FSM implementation for a dfa to decide the language of all words with an even number of as and an odd number of bs over the alphabet {a, b}. The machine has four states that are described as follows in terms of the read input:

S An even number of as and an even number of bs have been read. This is the starting state.
N An even number of as and an odd number of bs have been read. This is the only final state
M An odd number of as and an odd number of bs have been read.
P An odd number of as and an even number of bs have been read.

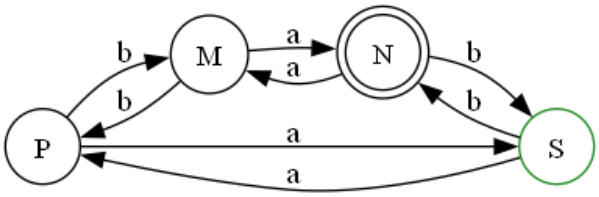
The transition function defines how the machine changes states upon reading the next element in the input word. For instance, consider the transition rule `(M b P)`. This transition states that if the machine is in state M and reads a b then it moves to state P. Observe this transition is correct. When the machine is in M, an odd number of as and an odd number of bs have been read. If a b is read, then an odd number of as and an even number of bs have been read, which is the condition that P represents. The transition diagram for EVEN-A-ODD-B is displayed in Figure 1b.

To test state machines, the FSM primitive sm-apply is used. It takes as input a state machine and a word, applies the machine to the word, and returns a symbol for the result: either `'accept` or `'reject`. In Figure 1a, the unit tests for EVEN-A-ODD-B are displayed on lines 13–34. Each test uses check-equal? to test a single word. Running the code in Figure 1a yields that all tests pass and no test failure reports are printed.

[1]In this article, we do not concern ourselves with tms and mttms that, instead of deciding or semideciding a language, compute a function.

```
1  (define EVEN-A-ODD-B
2    (make-dfa
3      '(S M N P)   ;; the states
4      '(a b)       ;; the alphabet
5      'S           ;; the starting state
6      '(N)         ;; the final states
7      ;; the transition function
8      '((S a P) (S b N) (M a N) (M b P)
9        (N a M) (N b S) (P a S) (P b M))
10     ;; optional argument: transition function is fully specified
11     'no-dead))
12 ;; Unit tests for EVEN-A-ODD-B
13 (check-equal? (sm-apply EVEN-A-ODD-B '())
14               'reject)
15 (check-equal? (sm-apply EVEN-A-ODD-B '(a b b a))
16               'reject)
17 (check-equal? (sm-apply EVEN-A-ODD-B '(b a b b a a))
18               'reject)
19 (check-equal? (sm-apply EVEN-A-ODD-B '(a b))
20               'reject)
21 (check-equal? (sm-apply EVEN-A-ODD-B '(a b b b b))
22               'reject)
23 (check-equal? (sm-apply EVEN-A-ODD-B '(b a b b a a b))
24               'reject)
25 (check-equal? (sm-apply EVEN-A-ODD-B '(b))
26               'accept)
27 (check-equal? (sm-apply EVEN-A-ODD-B '(b b b))
28               'accept)
29 (check-equal? (sm-apply EVEN-A-ODD-B '(a b b a b))
30               'accept)
31 (check-equal? (sm-apply EVEN-A-ODD-B '(a a b))
32               'accept)
33 (check-equal? (sm-apply EVEN-A-ODD-B '(a a a b a b b))
34               'accept)
```

(a) FSM implementation.



(b) Transition diagram.

Figure 1: An FSM dfa for L={w|w has an even number of a and an odd number of b}.

To illustrate test failure reports, consider changing the expected value for each test in Figure 1a from `'accept` to `'reject` and vice versa. A subset of the 11 failed test reports in displayed in Figure 2. Each error report includes the actual and expected values,

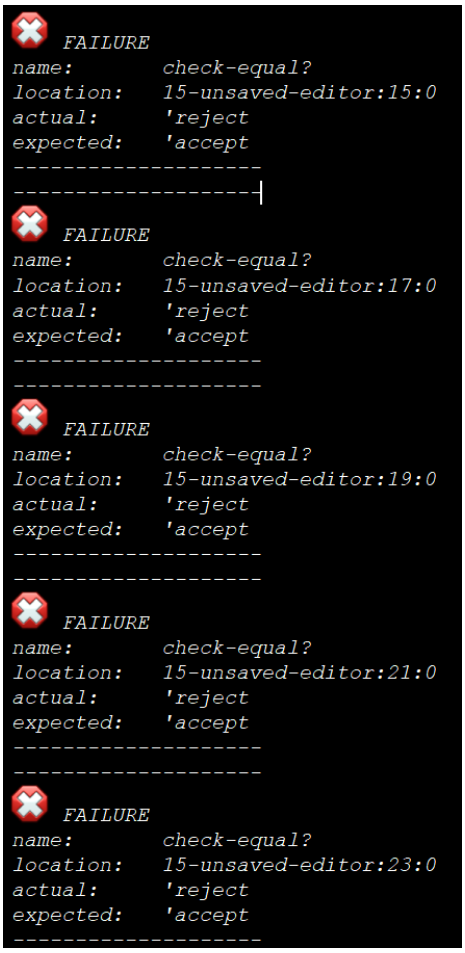


Figure 2: Failed test reports for state machines.

```
1  (define a2nb2n
2    (make-cfg '(S)     ;; nonterminals
3              '(a b)   ;; terminals
4              '((S → ϵ)  ;; production rules
5                (S → aSb))
6              'S))     ;; starting nonterminal
7  ;; Unit tests for a2nb2n
8  (check-equal?
9    (grammar-derive a2nb2n '(a b a))
10   "(a b a) is not in L(G).")
11 (check-equal?
12   (grammar-derive a2nb2n '(b b b))
13   "(b b b) is not in L(G).")
14 (check-equal?
15   (last (grammar-derive a2nb2n '()))
16   'ϵ)
17 (check-equal?
18   (last (grammar-derive a2nb2n '(a b)))
19   'ab)
20 (check-equal?
21   (last (grammar-derive a2nb2n '(a a b b)))
22   'aabb)
23 (check-equal?
24   (last (grammar-derive a2nb2n '(a a a b b b)))
25   'aaabbb)
26 (check-equal?
27   (last (grammar-derive
28           a2nb2n
29           '(a a a a a b b b b b)))
30   'aaaaabbbbb)
```

Figure 3: An `FSM` `cfg` for L=$a^n b^n$.

the location of the failed test, and the failed test predicate (i.e., `check-equal?`). Clicking on the icon in the top left corner of each failed test reports causes the failed test to be highlighted in the program window. For instance, clicking on the icon for the second failed test report (for the test on lines 15–16 in Figure 1a) yields:

```
(check-equal? (sm-apply EVEN-A-ODD-B '())
              'accept)
(check-equal? (sm-apply EVEN-A-ODD-B '(a b b a))
              'accept)
(check-equal? (sm-apply EVEN-A-ODD-B '(b a b b a a))
              'accept)
(check-equal? (sm-apply EVEN-A-ODD-B '(a b))
              'accept)
(check-equal? (sm-apply EVEN-A-ODD-B '(a b b b b))
              'accept)
```

Observe that the entire unit test expression is highlighted.

3.2 `FSM` Grammar Example

Figure 3 displays the `FSM` implementation for a `cfg` to derive the language of all words containing an arbitrary number of as followed by the same number of bs over the alphabet {a, b}. That is, the grammar's language is L=$a^n b^n$. The grammar has a single syntactic category (i.e., nonterminal symbol) described as follows:

S Derives a word in $a^n b^n$

The production rules define how a nonterminal on the left-hand side may be substituted with the amalgamation of symbols on the right-hand side. For instance, consider the production rule `(S → aSb)`. This rule states that the nonterminal symbol `S` may be substituted with `aSb` to advance the derivation of a word. Observe that this production rule is correct. If `S` generates a word in `L`, then this rule generates $aa^n b^n b$, which is also in `L`.

To test grammars, the `FSM` primitive `grammar-derive` is used. It takes as input a grammar and a word and returns a derivation for the word or a string indicating that the word is not in the grammar's language. In Figure 3, the unit tests for `a2nb2n` are displayed on lines 8–30. Each use of `check-equal?` tests a single word derivation. Running the code in Figure 3 yields that all tests pass and no test failure reports are printed. If the tests were changed so that they all

```
<test>
  ::= (check-reject? <machine> <word>*)
  ::= (check-accept? <machine> <word>*)
  ::= (check-reject? <machine> (<word> <head-pos>)*)
  ::= (check-accept? <machine> (<word> <head-pos>)*)
```

(a) FSMt concrete syntax for state machine tests.

```
<test> ::= (check-not-derive? <grammar> <word>*)
       ::= (check-derive?     <grammar> <word>*)
```

(b) FSMt concrete syntax for grammar tests.

Figure 4: FSMt concrete syntax.

fail, the resulting failed test reports look similar to those in Figure 2. In the interest of brevity, they are omitted.

4 The FSMt Interface

Even a cursory look at the FSM code in Figure 1a and Figure 3 reveals that writing unit tests is tedious, repetitive, and verbose. In fact, this is a reason why a significant cohort of students invariably complains every semester. Much to their credit, they are not mistaken. The unit tests in Figure 1a are all very similar. They all differ by only two elements: the word being tested and the expected value expression. All the unit tests in Figure 3 are also very similar. They all differ by three elements: the word being tested, the expected value expression, and the test expression. This strongly suggests that an abstraction is needed that hides (the repetitive) testing logic. This is exactly what DSLs excel at and this section describes the interface for FSMt. The description is divided into four parts based on four of the elements relevant to all DSL users as identified in Section 2: the concrete syntax, the static semantics, the dynamic semantics, and the design environment. We note that user evaluation is discussed in Section 6.

4.1 Concrete Syntax

The concrete syntax to write unit tests for state machines is displayed in Figure 4a. There are four varieties of unit testing expressions. The first two are used for dfas, ndfas, and pdas. The first is to test words that ought to be rejected by a given machine. The second is to test words that ought to be accepted by a given machine. Each requires a machine and an arbitrary number of test words. The second two varieties are used to test tms and mttms. For these varieties, each requires a machine and an arbitrary number of pairs. Each pair contains a word and the machine's initial head position.

The concrete syntax to write unit tests for grammars is displayed in Figure 4b. There are two varieties of unit testing expressions. The first is used to test words that should not be derived by the given grammar. The second is used to test words that ought be derived by the given grammar. Each requires a grammar and an arbitrary number of test words.

Observe that each unit-testing expression uses domain jargon. For instance, accept, reject, derive, and not-derive are used instead of the generic term equal found in Figure 1a and Figure 3. This helps to clearly communicate to domain experts and DSL programmers the purpose of the test. In addition, the abstraction is easily expressed. There is no longer a need, for example, to refer to sm-apply or grammar-derive. Finally, observe that unit tests may now be written concisely without sacrificing clarity. The unit tests from Figure 1a are refactored as follows:

```
(check-reject? EVEN-A-ODD-B
               '() '(a b b a) '(b a b b a a)
               '(a b) '(a b b b b)
               '(b a b b a a b))
(check-accept? EVEN-A-ODD-B
               '(b) '(b b b) '(a b b a b)
               '(a a b) '(a a a b a b b))
```

The unit tests from Figure 3 are refactored as follows:

```
(check-not-derive? a2nb2n '(b b b) '(a b a))
(check-derive? a2nb2n
               '() '(a a b b) '(a a a b b b)
               '(a b) '(a a a a a b b b b b))
```

The reader can appreciate the striking difference in length and domain-specific expressibility.

4.2 Static Semantics

The FSMt syntax is checked for several static constraints:

- At least two expressions are provided
- The first expression provided has the following type:
  - a state machine for machine tests
  - a grammar for grammar tests
- All but the first provided expressions have the following type:
  - a word for grammar, dfa, ndfa, and pda tests
  - a test pair for tm and mttm tests
- A test pair contains a word and a valid index into the word
- A word is a list of symbols all of which are contained in the machine's alphabet or the grammar's alphabet, respectively, for machine and grammar tests

Checking these static constraints prevents writing improper unit tests (e.g., testing grammar using sm-apply) that is possible using a GPL testing framework such as RackUnit. Checking the type of the first given expression guarantees that the provided syntax is not used with the incorrect computation model, thus, reinforcing the domain knowledge. That is, a machine either accepts or rejects a word and a grammar either derives or does not derive a word. The constraints on words guarantee that only valid words are tested. Assume the alphabet of a state machine or a grammar is Σ. These static constraints on words guarantee that machines are only applied to and grammars are only used to try to derive words in Σ^*. Such a constraint makes intuitive sense to domain experts (e.g., think of a FLAT Professor that is not a programmer) and precludes attempting to use arbitrary words that are not valid for a given machine or grammar. For instance, a machine whose alphabet is {a,b} can not be tested with a word that contains a c.

4.3 Dynamic Semantics

The dynamic semantics are described as follows (assuming static checks have passed):

- For machine tests, the given machine is applied to every given word or test pair using `sm-apply`
- For grammar tests, the given grammar is applied to every given word using `grammar-derive`
- A single failed test report is generated for all test words that fail to produce the expected result
- Every failed test report contains text similar to a recipe-based error message [3, 29] indicating the step of the design recipe [27] not successfully completed and the words that fail to produce the expected result

The dynamic semantics fulfill the expectations of domain experts. That is, machines are used to determine word membership in a language and grammars are used to derive words in a language.

To illustrate how a failed test report has meaning to a domain expert (in our case, students in a FLAT course), consider the following FSMt expression for the grammar displayed in Figure 3:

```
(check-derive?
  a2nb2n
  '(b a) '(a) '(a a a b b b b) '(b))
```

The generated fail test report is:

```
Step 6 of the design recipe has not been successfully
completed. The constructed grammar, a2nb2n, does not
derive the following words: (b a) (a) (b)
```

The step of the design recipe unsuccessfully completed (i.e., the step that requires writing unit tests) has meaning for the domain experts from lectures and the textbook [27] used for instruction. Observe that a failed test report contains the grammar/machine used in the tests. In this manner, the report has immediate meaning when several grammars/machine are defined in the same program file.

Finally, to illustrate that failed test reports are succinct and concise, consider the reports generated for:

```
(check-reject?
  EVEN-A-ODD-B
  '(b) '(b b b) '(a b b a b) '(a a b)
  '(a a a b a b b))

(check-accept?
  EVEN-A-ODD-B
  '() '(a b b a) '(b a b b a a) '(a b)
  '(a b b b b) '(b a b b a a b))
```

These are the same unit tests that generate the 11 failed test reports partially displayed in Figure 2. The use of FSMt as illustrated above generates:

```
Step 6 of the design recipe has not been successfully
completed. The constructed machine, EVEN-A-ODD-B, does
not reject the following words: (b) (b b b) (a b b a b)
(a a b) (a a a b a b b)

Step 6 of the design recipe has not been successfully
completed. The constructed machine, EVEN-A-ODD-B, does
not accept the following words: () (a b b a)
(b a b b a a) (a b) (a b b b b) (b a b b a a b)
```

Undoubtedly, it is preferable to read these two succinct and concise failed test reports rather than 11 generic failed test reports that carry no domain-specific knowledge produced by the GPL's testing framework

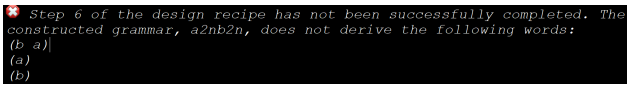
4.4 Design Environment

The DrRacket IDE [4] is used to provide the failed test locations and syntax highlighting in a program's file. To this end, for every failed test report only the tested words that cause the FSMt expression to fail are highlighted. Programmers can jump and see the highlighted results by clicking on the icon included in the report (i.e., the same icon discussed in Section 3.1).

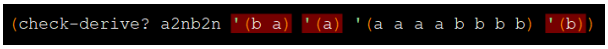
To illustrate the use of highlighting, consider, once again, the following FSMt expression first used in Section 4.3:

```
(check-derive?
  a2nb2n
  '(b a) '(a) '(a a a b b b b) '(b))
```

The generated report inside the DrRacket IDE is displayed as follows:



Clicking on the icon in the top left corner leads to the following display in the program's definition window:



As can be observed, only the words not derived by a2nb2n are highlighted. Thus, precisely identifying the locations and reasons for the failed tests report.

5 Implementation

This section describes the implementation of FSMt. The presentation is also organized around 4 of the 6 core implementation aspects identified in Section 2: concrete syntax, static semantics, dynamic semantics, and design environment. Given that the target domain is unit testing for FSM, FSMt never throws an exception (e.g., as is usually done when a static constraint is violated). Instead, it always generates a warning when a static constraint is violated or a test fails. When a static constraint is violated, the tests are not evaluated.

5.1 Concrete Syntax

FSMt is implemented using Racket macros [5, Chapter 16]. Racket macros are the right tool for the job for two reasons. The first is that they provide a straightforward means for implementing the concrete syntax described in Section 4.1. The second is that they provide access to syntax objects that are used to implement code highlighting when a warning report is generated (e.g., as described in Section 4.4).

To manipulate syntax at runtime, the syntax object must be encoded as a runtime value at compilation time to provide access to information such as the syntax type (e.g., the expression type) and the source location in the program's file. This information is used to customize failed test reports and to correctly target expressions for highlighting.

```
1 (define-syntax (check-in-lang? stx)
2   (syntax-parse stx
3     [(check-in-lang?)
4      #'(handle-no-vals (val-stx-pair stx #'stx))]
5     [(check-in-lang? val1)
6      #'(handle-one-val
7          val1
8          (val-stx-pair stx #'stx))]
9     [(check-in-lang? val1 w ...)
10     #'(execute-rt-checks
11        #t
12        (val-stx-pair val1 #'val1)
13        (list (val-stx-pair w #'w) ...))]))
```

Figure 5: Primary FSMt macro.

One of the two primary FSMt macros is displayed in Figure 5. This macro generates code for testing words that ought to be accepted by a state machine or ought to be generated by a grammar. There is an analogous macro, check-not-in-lang, for words that ought to be rejected by a state machine or ought to not be generated by a grammar. The macro in Figure 5 implements generic syntax used to expand the FSMt syntax check-accept? and check-derive?. It dispatches on whether or not the static requirement of providing one FSM value and at least one test value is met. Based on this, there are three different cases to pattern match:

(check-in-lang?) No expressions are provided for the test
(check-in-lang? val1) Only one expression is provided for the test
(check-in-lang? val1 w ...) At least two expressions are provided for the test

When zero or one expression is provided to a test, a warning message is displayed, respectively, by the macro-generated function call to handle-no-vals (line 4) and handle-one-val (line 6). When the proper number of expressions is provided to a test, static constraints violations and failed test reports, if any, are displayed by the macro-generated function call to execute-rt-checks. If no static errors are detected, this function is responsible for running all the tests. Each of these functions manipulates both runtime values and syntax objects. A value is associated with the syntax that generates it through the use of, val-stx-pair, a structure that contains the runtime value (e.g., val1 and w) and its corresponding syntax object (e.g., #'val1 and #'w).

To illustrate how these functions are implemented, consider the implementation of execute-rt-checks displayed in Figure 6. This function dispatches on the type of FSM value tested (i.e., state machine or grammar), which is extracted from the val-stx-pair instance provided for it as input[2]. If the given FSM value is not a state machine or grammar, a warning is thrown and the tests are not executed. Otherwise, the tests are executed by a test-checking function (e.g., check-grammar), which checks if the elements in wlist are valid. If they are not valid, a warning is thrown and the

[2]Unlike dfas, ndfas, and pdas, a separate stanza for tms is used, because the tested value are not just words. They also contain the head's starting position.

```
(define (execute-rt-checks accept? fsmval wlist)
  (cond
    [(tm? (val-stx-pair-val fsmval))
     (check-tm accept? fsmval wlist)]
    [(fsa-pda? (val-stx-pair-val fsmval))
     (check-machine accept? fsmval wlist)]
    [(grammar? (val-stx-pair-val fsmval))
     (check-grammar accept? fsmval wlist)]
    [else
     (display-fsmt-warning
       (warning:invalid-fsmval fsmval))]))
```

Figure 6: Top-level function to run tests.

tests are not evaluated. Otherwise, the tests are evaluated and, if necessary, failed test reports are generated.

Finally, FSMt concrete syntax is transformed to check-in-lang? syntax as follows:

```
(define-syntax check-accept?
  (make-rename-transformer #'check-in-lang?))
(define-syntax check-derive?
  (make-rename-transformer #'check-in-lang?))
```

In essence, any use of check-accept? and check-derive? syntax is rewritten as check-in-lang? syntax. An analogous transformation is done from check-reject? and check-not-derive? syntax to check-not-in-lang? syntax.

5.2 Static Semantics

Static tests are performed at runtime in FSMt to create more informative warning reports. Before any tests are evaluated, static properties are validated. Each static property tested is represented using a property-check structure that has 4 characteristics: a predicate, test values, a constructor for a structure that represents a FSMt warning, and an FSM value (i.e., a state machine or a grammar). The predicate must hold for all test values. If it does not, then a warning structure is constructed. This structure is eventually used to generate the string containing the test values that cause the predicate to fail. The FSM value is used in the displayed warning message when the syntax value that it is generated from is an identifier. That is, if the machine or grammar is bound to a name, then this name is used in the warning message.

To illustrate how checks are implemented, consider the implementation of check-machine displayed in Figure 7. Recall that this function is called by the FSMt top-level function to run tests displayed in Figure 6. It takes as input a Boolean to indicate if the given words are tested for acceptance or rejection, a val-stx-pair structure for a state machine (i.e., a dfa, an ndfa, or a pda), and a list of val-stx-pairs for the words. There are three static properties that are validated. The first check validates that each of the given expressions for words is a list of symbols (line 3). The second check validates that each word is formed by elements in the given machine's alphabet (line 4). The third check performs the tests (lines

```
(define (check-machine accept? M words)
  (run-property-checks
    (property-check (listof symbol?) words warn:fsmt:app:sm:invalid-word M)
    (property-check (valid-word-elems/c (val-stx-pair-val M)) words warn:fsmt:app:sm:invalid-sigma M)
    (if accept?
        (property-check (word-accepted? (val-stx-pair-val M)) words warn:fsmt:app:sm:accept M)
        (property-check (word-rejected? (val-stx-pair-val M)) words warn:fsmt:app:sm:reject M))))
```

Figure 7: Implementation of `check-machine`.

```
(define (run-p-checks . p-checks)
 (for
  ([p-check (in-list p-checks)]
   #:do [(define errVals
           (accumulate-err-vals
            (property-check-contract p-check)
            (property-check-testVals p-check)))]
   #:final (not (null? errVals)))
  (when (not (null? errVals))
        (display-fsmt-warning
         ((property-check-errType p-check)
          (property-check-fsmVal p-check)
          errVals)))))
```

Figure 8: Implementation of property validation.

5–7). This last check uses the given Boolean to determine if tests for accepting or rejecting the words are being evaluated.

5.3 Dynamic Semantics

The order in which the tests are performed by `check-machine` (and, similarly, for `check-tm` and `check-grammar`) implements an important dynamic feature: the precedence of warnings. For machines, first every word argument must be a list, second every word must only contain machine-alphabet elements, and third every word needs to produce the expected test outcome. The precedence of warnings is important to create an accurate warning message and avoid misleading programmers towards a symptom of a mistake rather than to the source of the mistake. For example, testing a word for acceptance/rejection before (or without determining) that the word is valid input to the machine would generate a failed test warning. This may not be incorrect, but it is not accurate enough. The source of the mistake is not that the test using this word failed. Instead, it is that the word is not valid input for the machine. In essence, the precedence of errors helps guarantee that more accurate warnings or failed test reports are generated.

To illustrate how the static checks and the tests are performed, the implementation of `run-p-checks` is displayed in Figure 8. For each property checked, the predicate in the `property-check` instance is applied to all the values being tested, which are also contained in the `property-check` instance. The values for which the property fails are accumulated. If any such values exist for a given check, the remaining properties are not checked and a warning is generated by `display-fsmt-warning` using the structure representing the instance of the warning type, the `FSM` value, and the accumulated values. This is how a single warning is dynamically generated for multiple values, thus, avoiding such a generation for each individual value.

- warn:fsmt
  - warn:fsmt:test
    - warn:fsmt:test:no-cases
    - warn:fsmt:test:no-fsm-val
    - warn:fsmt:test:no-testvals
    - warn:fsmt:test:invalid-fsm-val
  - warn:fsmt:app
    - warn:fsmt:app:gmr
      - warn:fsmt:app:gmr:invalid-word
      - warn:fsmt:app:gmr:invalid-terminal
      - warn:fsmt:app:gmr:derive
      - warn:fsmt:app:gmr:not-derive
    - warn:fsmt:app:sm
      - warn:fsmt:app:sm:accept
      - warn:fsmt:app:sm:reject
      - warn:fsmt:app:sm:invalid-elem
      - warn:fsmt:app:sm:invalid-word
    - warn:fsmt:app:tm
      - warn:fsmt:app:tm:invalid-arity
      - warn:fsmt:app:tm:invalid-head-pos
      - warn:fsmt:app:tm:invalid-head-pos-index
      - warn:fsmt:app:tm:no-left-hand-marker
      - warn:fsmt:app:tm:invalid-word
      - warn:fsmt:app:tm:invalid-elem
      - warn:fsmt:app:tm:accept
      - warn:fsmt:app:tm:reject

Figure 9: Warning structure inheritance hierarchy.

`FSMt` warning messages are created using a generic interface that abstracts over the warning structure hierarchy displayed in Figure 9. To briefly describe this hierarchy, we note that there are two variants for `FSMt` warnings: `test` and `app`. The `test` variant is used for static checks to validate that the `FSMt` syntax is correctly used (e.g., at least two expressions are provided). It has 4 variants (from top to bottom) for when no values are provided, when no `FSM` value is provided, for when no test words are provided, and for when an incorrect `FSM` value is provided. The `app` variant is

```
(define-generics fsmt-warning
  (named-multi-failure fsmt-warning)
  (named-single-failure fsmt-warning)
  (anon-multi-failure fsmt-warning)
  (anon-single-failure fsmt-warning)
  (create-warning-str fsmt-warning)
  (display-fsmt-warning fsmt-warning)
  #:fallbacks
   [(define (create-warning-str warning)
      (cond [(warn:fsmt:app? warning)
             (if (identifier?
                   (val-stx-pair-stx
                     (warn:fsmt-fsm-expr warning)))
                 (if (single-warning? warning)
                     (named-single-failure warning)
                     (named-multi-failure warning))
                 (if (single-warning? warning)
                     (anon-single-failure warning)
                     (anon-multi-failure warning)))]
            [(warn:fsmt? warning)
             (if (identifier?
                   (val-stx-pair-stx
                     (warn:fsmt-fsm-expr warning)))
                 (named-single-failure warning)
                 (anon-single-failure warning))]
            [else
              (raise-user-error
                "Invalid error type raised.")]))])
```

Figure 10: Generation of FSMt generics.

used for all other static constraints and for test evaluation. It has 3 variants: one for grammars (gmr), one for non-tm state machines (sm), and one for tms (tm). Each of these has variants. For example, the gmr has variants (from top to bottom) for when a word is not a list, for when a word contains elements not in the grammar's alphabet, for when a word that ought to be derived is not, and for when a word that ought to not be derived is derived. In the interest of brevity, the variants of sm and tm are not described.

This generic interface allows for different implementations to be defined per structure variant for the same generic function. This is important, because it allows for the generation of customized warning messages inspired by recipe-based error messages (discussed in Section 4). Such an approach has two benefits: the first is that all message generators for a warning type are encapsulated within the definition of its structure. The second is that the generic functions abstract away the use of structures to display a warning.

An abbreviated version of the implementation of this generic interface is displayed in Figure 10. The first expression is an identifier that defines the name of the interface (i.e., `fsmt-warning`). The following expressions define the headers for the generic functions. The keyword argument `#:fallbacks` allows for the definition of default implementations of the generic functions. In this example, only a default function for `create-warning-str` is provided (hence, the abbreviated nature of the figure). This function generates the string that is used in a warning message. It dispatches on the structure variant of `warn:fsmt` given as input. For each variant, the syntax that generated the given FSM value is tested to determine if it is an identifier. If so, the generation of the warning makes use of it. Otherwise, the warning message generated is anonymous. In addition, for a `warn:fsmt:app` instance, the warning structure (i.e., `warning`) is tested to determine if it represents a single check failure or multiple check failures and, accordingly, the grammar used in the warning message is adjusted.

5.4 Design Environment

The DrRacket IDE is used to highlight syntax when warnings are raised and to prevent the FSM programmer from examining the trace stack, which contains information that is not relevant to understanding the warning. To implement both features, Racket exception structures are used.

To display FSMt warning messages, Racket error output ports are used through the `error-display-handler` function. This function takes as input a Racket exception structure that must include the warning string and a list of source locations to highlight. The warning string is generated by the generic functions discussed in Section 5.3 (see Figure 10). The source locations are generated by the FSMt macros (e.g., see Figure 5), which are stored inside generated `val-stx-pair` instances.

To make the trace stack inaccessible to FSMt programmers, FSMt warnings use the Racket exception variant `exn:fail:user`. When such a variant is provided to `error-display-handler`, the stack trace is not displayed to the programmer. This is important, because it prevents FSM programmers from ignoring the highlighted syntax and looking for the reason for the warning in the FSMt implementation. In essence, providing access to the trace stack violates the abstraction provided by the FSMt interface.

6 Human Factors Formative Study

This sections presents the results obtained from a small formative study. First, the empirical data is presented. Second, the limitations and threats to validity are discussed.

6.1 Empirical Results

To measure perceptions about FSMt, Seton Hall University's spring 2025 undergraduate Formal Languages and Automata Theory course students answered an anonymous survey. All of its 10 students volunteered to participate. The students are all 3rd or 4th year Computer Science majors between the ages of 20 and 23. The sample consists of 30% females and 70% males. No student received any benefit or compensation for their participation.

Using a Likert scale [18] to respond, 1 (Strongly disagree) to 5 (Strongly agree), with 3 as a neutral response, the survey presents the following statements:

Q1.1 Writing tests using check-equal? is straightforward
Q1.2 Writing tests using check-accept? and check-reject? is straightforward

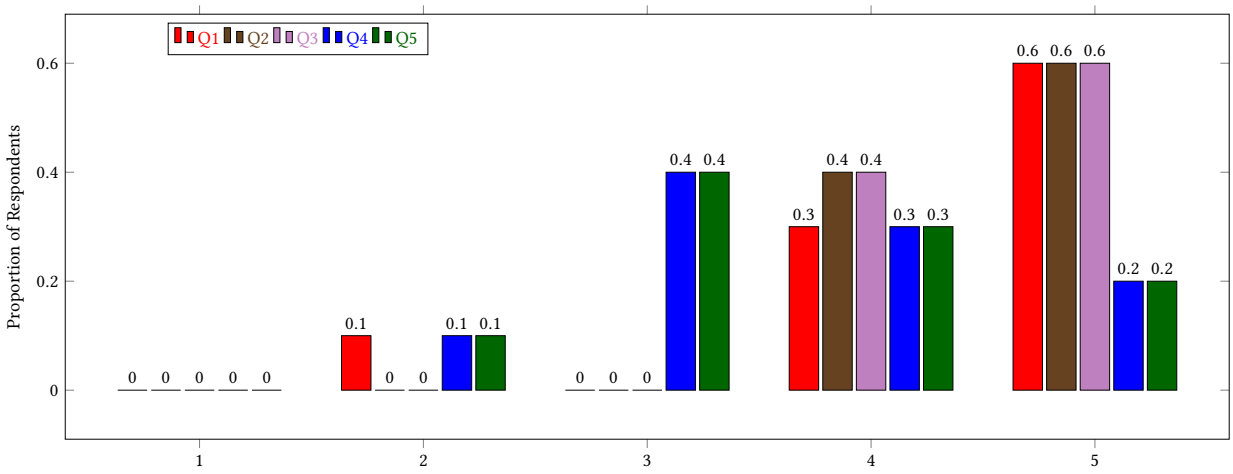


Figure 11: Data distribution for anonymous survey.

Q1.3 Writing tests using check-derive? and check-not-derive? is straightforward
Q1.4 I prefer to use check-accept? and check-reject? over check-equal? to write unit tests for state machines
Q1.5 I prefer to use check-derive? and check-not-derive? over check-equal? to write unit tests for grammars

The distribution of responses is displayed in Figure 11. For Q1.1, we observe that an overwhelming majority of respondents, 90%, tend to agree (responses 4 and 5) that writing tests using check-equal? is straightforward. There is a small minority, 10%, that tends to disagree (responses 1 and 2), but not strongly. This result establishes a high bar for `FSMt` to meet.

The distribution of responses for Q1.2 and Q1.3 reveal that 100% of respondents tend to agree (responses 4 and 5) that test writing using `FSMt` is straightforward. These are very encouraging results, because they suggest that the conciseness and use of domain jargon is well-received by the respondents. This result is also somewhat surprising given that the book of instruction [27] utilizes check-equal? to express unit tests. Our expectation was that some cohort of students would feel that it is more straightforward to emulate the textbook.

The distributions of responses for Q1.4 and Q1.5 reveal a bit more nuanced picture. For both, we observe that 50% of the respondents tend prefer (responses 4 and 5) to write unit tests using `FSMt` over check-equal?. For each distribution there is, however, a significant proportion, 40%, that feels neutral and a small minority, 10%, that disagrees but not strongly. This result is extremely positive. We have that half the respondents prefer to write unit tests using a technology that is new to them (i.e., `FSMt`) for which they have no further guide but lectures and the `FSM` documentation [26]. Our expectation was to see a much larger cohort tending to disagree (responses 1 and 2). Instead, we observe that this larger cohort feels neutral. Our expectation is that as documentation improves (and when a new edition of the textbook is released), we ought to see an even larger cohort tending to agree.

6.2 Limitations and Threats to Validity

We acknowledge that the study presented has limitations and threats to validity. An obvious threat to validity is the small sample size. In addition, the sample was drawn from a mostly homogeneous population. That is, the overwhelming majority of Computer Science students at `SHU` are European American men. Therefore, it is unclear if the same results would be observed using a larger sample chosen from a more diverse Computer Science student population. To address these problems, future iterations of the `SHU` course shall be studied and efforts will be made to make it a interuniversity study.

Another limitation of the study is its quantitative nature. Quantitative studies fail to capture the personal experiences of participants and, therefore, fail to capture the instances when respondents felt `FSMt` was useful or a hindrance. To address this limitation, future work includes the use of open-ended survey questions and observational studies.

In many social and physical sciences, it is common to study a population by comparing a control and an experimental group. In our case, the experimental group would use `FSMt` (i.e., the independent variable) and the control group would only use `RackUnit`. Since this was not possible, we cannot claim that the independent variable is fully isolated. It is unclear if we can address this issue. University officials would raise ethical questions about providing different educational experiences to students for the sake of a study.

Finally, the formulation of the survey is a limitation. In retrospect, it is obvious that the survey does not address some important issues such as the length and the quality of fail tests reports, and whether the use of domain jargon was a strength or a hindrance. To address this shortcoming, future studies will include statements about fail test reports that address these characteristics.

7 Related Work

Several efforts have been made to develop `DSL`s for expressing tests. We briefly describe some of these efforts. `jMock` is a `DSL` for test-driven development using Mock Objects in `Java` [8, 9]. This `DSL`

eases the construction on Mock Objects that are used for testing purposes. One of the achievements obtained is that code reads like a declarative specification instead of an imperative API. In this regard, jMock and FSMt are similar. In contrast, the work presented does not need to mimic (i.e., mock) machines or grammars that have not (yet) been written. Neither jMock's web presence (http://jmock.org/) nor, to the best of our knowledge, the use of other Mock-based DSLs (e.g., [11]) have reported the results obtained from user studies.

FitNesse is an acceptance testing DSL [23]. In FitNesse, tests are expressed as tables of input data and expected output data. This language facilitates the creation of these test tables. Like FSMt, how tests are evaluated is hidden in the language's implementation. In contrast, FSMt and jMock do not use tables to express tests in a declarative style. To the best of our knowledge, no user studies have been conducted.

Wu, Grey, and Menik develop a DSL testing framework by leveraging the testing framework of an underlying GPL [40]. Their system maps DSL tests to GPL tests and report the results at the DSL level (i.e., the domain expert is only aware of the testing at the DSL level). In contrast, FSMt does not map unit testing to the testing framework of the host language (i.e., Racket). Instead, it keeps the evaluation of the tests within the context of the target language (i.e., FSM). In further contrast, Wu et al. do not present an user feedback.

Juhnke and Tichy explored the development of automotive testing DSLs [14]. They conclude that a generic testing DSL is not suitable and that system specific DSLs are needed. They focus on the systematic development of testing DSLs that extract domain-specific concepts from automotive test case specifications. They do not present, as the work cited above, any user feedback. In contrast, the work presented in this article focuses on a single DSL and on improving and reporting the testing experience for the DSL users.

Providing support for testing in DSLs is considered by many a costly and challenging task [15]. To address this challenge, a generic Test Description Language standard, TDL [34], has been developed by ETSI (European Telecommunications Standards Institute). TDL allows test descriptions to be specified which, in turn, are used to implement concrete tests in a DSL. Given its generic nature, TDL does not address domain-specific knowledge required to write tests in an executable DSL and does not include execution facilities [15]. To address these shortcomings, Khorram et al. developed a generic DSL testing framework based on a refinement of TDL that extracts domain knowledge from a target DSL and on the implementation of a TDL interpreter for test execution [15]. Their evaluation reports success with generating a testing framework for some DSLs and also highlights some shortcomings associated with time-dependent behavior. No user feedback is reported.

8 Concluding Remarks

This article presents design principles for the development of embedded unit-testing domain-specific languages. It advocates 5 development aspects, directly related to DSL users, that need to be addressed: concrete syntax, static semantics, dynamic semantics, design environment, and user evaluation. The first four must be informed by domain knowledge to capture necessary abstractions and make them part of the language implementation. Given that unit-testing domain-specific languages are about making it easier for humans, both programmers and non-programmers, to read and write tests, the need for user evaluation is necessary. Poorly captured abstractions, complicated syntax, and unclear semantics, for example, may derail the adoption of a unit-testing domain-specific language. Human studies can inform developers of problem areas to improve as well as features that work well and ought to be kept. In addition, this article presents FSMt–a unit-testing domain-specific language for, FSM, a language developed for the Automata and Formal Languages classroom. Its development is outlined through the lens of the 5 identified development aspects and its macro-based implementation is illustrated. The collected empirical data from a formative study suggests that FSMt was well-received by its users on its maiden voyage.

Future work includes extending the developed language to encompass the testing of FSM regular expressions. In addition, future work also includes performing larger empirical studies over several iterations of the course to increase sample size and performing interuniversity studies to increase population diversity. Finally, future work also includes development of macro-generated contracts for FSM primitives to offer better error messages.

References

[1] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. doi:10.1145/3344429.3372508

[2] Matilda Blomqvist, Nicklas Boto, Beata Burreau, Fabian Forslund, Marcus Jorgensson, and Joel Rudsberg. 2021. A Functional Quantum Programming Language. Last Accessed: June 2025.

[3] Shamil Dzhatdoyev, Josephine A. Des Rosiers, and Marco T. Morazán. 2024. Error Messages for Students Taught Using a Systematic Program Design Curriculum. In *Trends in Functional Programming - 25th International Symposium, TFP 2024, South Orange, NJ, USA, January 10-12, 2024, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 14843)*, Jason Hemann and Stephen Chang (Eds.). Springer, 195–219. doi:10.1007/978-3-031-74558-4_9

[4] Robert Bruce Findler. 2014. DrRacket: The Racket Programming Environment. *Racket Language Documentation* (2014).

[5] Matthew Flatt, Robert Bruce Findler, and PLT. 2024. *The Racket Guide.* PLT. https://docs.racket-lang.org/guide/ last accessed: March 2025.

[6] Marin Fotache and Catalin Strimbei. 2015. SQL and Data Analysis. Some Implications for Data Analysits and Higher Education. *Procedia Economics and Finance* 20 (2015), 243–251. doi:10.1016/S2212-5671(15)00071-4

[7] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

[8] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. 2004. Mock Roles, not Objects. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 236–246. doi:10.1145/1028664.1028765

[9] Steve Freeman and Nat Pryce. 2006. Evolving an Embedded Domain-Specific Language in Java. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 855–865. doi:10.1145/1176617.1176735

[10] Emden R. Gansner and Stephen C. North. 2000. An Open Graph Visualization System and Its Applications to Software Engineering. *Softw. Pract. Exper.* 30, 11 (September 2000), 1203–1233. doi:10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N

[11] Colin S. Gordon. 2024. Mocking Temporal Logic. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E* (Pasadena, CA, USA) *(SPLASH-E '24)*. Association for Computing Machinery, New York, NY, USA, 98–109. doi:10.1145/3689493.3689980

[12] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2009. Domain-Specific Languages in Practice: A User Study on the Success Factors. In *Model Driven Engineering Languages and Systems*, Andy Schürr and Bran Selic (Eds.). Springer

Berlin Heidelberg, Berlin, Heidelberg, 423–437.
[13] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (Dec. 1996), 196–es. doi:10.1145/242224.242477
[14] Katharina Juhnke and Matthias Tichy. 2019. A Tailored Domain Analysis Method for the Development of System-Specific Testing DSLs Enabling Their Smooth Introduction in Automotive Practice. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 10–18. doi:10.1109/SEAA.2019.00011
[15] Faezeh Khorram, Erwan Bousse, Jean-Marie Mottu, and Gerson Sunyé. 2021. Adapting TDL to Provide Testing Support for Executable DSLs. *The Journal of Object Technology* 20, 3 (2021), 6:1–15. doi:10.5381/jot.2021.20.3.a6
[16] P. J. Landin. 1966. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. doi:10.1145/365230.365257
[17] Harry R. Lewis and Christos H. Papadimitriou. 1997. *Elements of the Theory of Computation* (2nd ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA. doi:10.1145/300307.1040360
[18] Rensis Likert. 1932. A Technique for the Measurement of Attitudes. *Archives of Psychology* 140 (1932), 1–55.
[19] Peter Linz. 2011. *An Introduction to Formal Languages and Automata* (5th ed.). Jones and Bartlett Publishers, Inc., USA.
[20] Ralf Lämmel. 2018. *Software Languages: Syntax, Semantics, and Metaprogramming* (1st ed.). Springer Publishing Company, Incorporated.
[21] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2011)*. Association for Computing Machinery, New York, NY, USA, 3–18. doi:10.1145/2048237.2048241
[22] John C. Martin. 2002. *Introduction to Languages and the Theory of Computation* (3 ed.). McGraw-Hill, Inc., USA.
[23] Robert C. Martin, Micah D. Martin, Patrick Wilson-Welsh, and `FitNesse` contributors. [n. d.]. FitNesse User Guide. https://fitnesse.org/FitNesse/UserGuide.html. Last Accessed: June 2025.
[24] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316–344. doi:10.1145/1118890.1118892
[25] Eric A Meyer. 2006. *CSS: The Definitive Guide.* O'Reilly Media, Inc.
[26] Marco T. Morazán. 2024. *FSM.* Seton Hall University, South Orange, NJ, USA. https://morazanm.github.io/fsm/fsm/index.html last accessed: June 2025.
[27] Marco T. Morazán. 2024. *Programming-Based Formal Languages and Automata Theory - Design, Implement, Validate, and Prove.* Springer. doi:10.1007/978-3-031-43973-5
[28] Marco T. Morazán and Rosario Antunez. 2014. Functional Automata–Formal Languages for Computer Science Students. In *Proceedings 3rd International Workshop on Trends in Functional Programming in Education, TFPIE 2014, Soesterberg, The Netherlands, 25th May 2014 (EPTCS, Vol. 170)*, James L. Caldwell, Philip K. F. Hölzenspies, and Peter Achten (Eds.). 19–32. doi:10.4204/EPTCS.170.2
[29] Marco T. Morazán, Shamil Dzhatdoyev, Josephine A. Des Rosiers, Rose Bohrer, Andrés M. Garced, and David Anthony K. Fields. 2025. Recipe-Based Errors: Methodology, Implementation, and Evaluation. In *Proceedings of the 37th Symposium on Implementation and Application of Functional Languages* (Montevideo, Uruguay) *(IFL '25)*. Association for Computing Machinery, New York, NY, USA. Under review.
[30] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: Executable Specifications of Interactive Fork flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 141–152. doi:10.1145/1291151.1291174
[31] Elaine Rich. 2019. *Automata, Computability and Complexity: Theory and Applications.* Pearson Prentice Hall.
[32] Michael Sipser. 2013. *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning, USA.
[33] Raúl Tabarés. 2021. HTML5 and the Evolution of HTML; Tracing the Origins of Digital Platforms. *Technology in Society* 65 (2021), 101529. doi:10.1016/j.techsoc.2021.101529
[34] Andreas Ulrich, Sylvia Jell, Anjelika Votintseva, and Andres Kull. 2014. The ETSI Test Description Language TDL and its application. In *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 601–608.
[35] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.* 35, 6 (June 2000), 26–36. doi:10.1145/352029.352035
[36] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages.* dslbook.org. http://www.dslbook.org
[37] Andrzej Wasowski and Thorsten Berger. 2023. *Domain-Specific Languages: Effective Modeling, Automation, and Reuse.* Springer, Germany.
[38] Noel Welsh and Ryan Culpepper. 2024. *RackUnit: Unit Testing* (v8.12 ed.). PLT Racket. https://docs.racket-lang.org/rackunit/index.html last accessed: March 2025.
[39] John Wrenn and Shriram Krishnamurthi. 2017. Error Messages are Classifiers: A Process to Design and Evaluate Error Messages. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 134–147. doi:10.1145/3133850.3133862
[40] Hui Wu, Jeff Gray, and Marjan Mernik. 2009. Unit Testing for Domain-Specific Languages. In *Domain-Specific Languages*, Walid Mohamed Taha (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–147.

Compilation of the Stochastic Language ALEA

Baltasar Trancón y Widemann
Brandenburg University of Applied Sciences
Brandenburg an der Havel, Germany
trancon@th-brandenburg.de

Markus Lepper
semantics gGmbH
Berlin, Germany

Abstract

Alea is a domain-specific declarative language for stochastic experiments. The language has clean and simple semantics for two evaluation modes: a static one for calculation of the probability distribution of outcomes, and a dynamic one for pseudo-random sampling. The reference implementation is an interpreter written in Java which follows the presentation of semantics closely, strongly prioritizing clarity over performance. Here, we describe the key requirements and concepts for an optimizing compiler to JVM bytecode, and evaluate the performance impact of compilation.

CCS Concepts

• **Software and its engineering** → **Domain specific languages**; **Compilers**; • **Applied computing** → *Mathematics and statistics*; • **Mathematics of computing** → *Distribution functions*; • **Theory of computation** → **Categorical semantics**.

Keywords

Stochastics, Domain-Specific Language, Compilation, Optimization

ACM Reference Format:
Baltasar Trancón y Widemann and Markus Lepper. 2025. Compilation of the Stochastic Language ALEA. In *Proceedings of 37th Symposium on Implementation and Application of Functional Languages (IFL '25).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Alea [11] is a domain-specific language for modeling manually performable random experiments. The language is intended for both educational and recreational use, in the teaching of stochastics, in simulations and in the design, analysis and play of games of chance.

To this end, alea is essentially a purely functional expression language with syntax and semantics that very closely mimic the conventions of elementary mathematics, extended orthogonally with randomness primitives. The language constructs are chosen such that typical informal specifications of 'educational' random experiments can be formalized rigorously and concisely. An alea program can be either analyzed statically, calculating the exact probability distribution of outcomes, or executed dynamically for pseudo-random sampling and simulation.

IFL '25, Montevideo, Uruguay

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

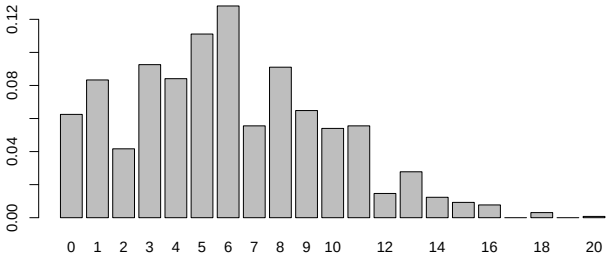


Figure 1: Example Distribution, $n = 4$

1.1 Motivating Example

Before language features are discussed in more technical detail, the following example shall give a characteristic first impression.

Consider the informal specification:

> *"Roll n identical six-sided dice, and sum only the odd values."*

Note that without the qualifier "odd" the result is described by a known distribution [1], but with it there is no such result. A formalization that is also an effective implementation in alea (except for the value of the parameter n) can be given as follows:

```
dice := ⟨ ~uniform{1..6} | _ ← [1..n] ⟩;
score := (+)⟨ d ← dice | d \\ 2 = 1 ⟩
```

(See Figure 1 for a probability distribution computed by the alea system, and visualized with an R script.)

Alea has exact (unbounded precision) rational numbers as primitive data; here only naturals are used. Besides, there are three kinds of collections distinguished by bracket shape: Square `[]` for (finite) lists; curly `{}` for (finite) sets; angled ⟨⟩ or baggy ⟅⟆ for bags (finite multisets).

Simple collections are given by enumeration or as an interval. `{1..6}` denotes the set of the natural numbers from 1 to 6, both inclusive. Likewise, `[1..n]` denotes the list of (one occurrence each of) the natural numbers from 1 to n.

By design, alea avoids both notoriously partial operations such as indexed element access, and higher-order functions. Collections are processed with comprehensions (effectively *map* and *filter* operations) and monoid morphisms (effectively *reduce* operations).

In the example above, the expression assigned (*let*-bound) to `dice` is a *map*-like comprehension that iteratively draws values from the list `[1..n]`, assigns each to an anonymous variable `_`, evaluates the random expression `~uniform{1..6}` (the uniformly distributed choice from the given set) repeatedly and independently, and collects the results in a bag, forgetting the iteration order.

The expression assigned to `score` is a *filter*-like comprehension that retains only the odd elements of `dice`, which is then reduced

with the monoidal operation +. Reduction is notated by applying a nominally binary function to a collection instead. The binary function must obey algebraic laws on the element type [3]: It must be associative for all kinds of collections, commutative for bags and sets, and idempotent for sets. Furthermore, a neutral element must be known if the collection is possibly empty.

If a final *let* binding (here for `score`) is not followed by another expression, it determines the overall value.

The example gives an ALEA program in a single expression, albeit structured with *let* bindings, that refers to predefined primitive/library operations. Indeed, in the current version of the language, there are no user-defined abstractions. While the language is planned to grow and be equipped with user-level function definitions and other means for reuse, the current simple state suffices to cover many typical real-world examples, and to exhibit many interesting implementation challenges.

1.2 Contributions

In the present paper, we address the following topics:

- Aspects of the semantic design of ALEA that are relevant to compilation (section 2);
- the reference interpreter, and how its direct implementation of the semantic structures causes overhead in several ways (section 3);
- compilation tactics that derive from the preceding observations (section 4);
- empirical evaluations that, even with a small set of benchmarks, both corroborate the compilation tactics and expose their limitations (section 5).

The major findings can summarized thus:

- There is significant 'impedance mismatch' between a declarative source language with monadic data structures and semantics, and an imperative target language.
- This issue can be addressed tactically by marrying the declarative optimization technique of deforestation to the imperative builder pattern for incremental buildup of complex data structures without temporaries.
- The employed techniques deserve a more rigorous and abstract treatment, for verification and broader application, respectively. The present paper reports on experimental work that may serve as a proof of concept; more research is needed in the future.
- Ironically, data structures other than lists (sets and bags) prove valuable at the source level for mitigating combinatorial explosion, but limit the benefits of optimizing compilation, for the same reasons.

2 Semantics

We summarize definitions of ALEA semantics, as far as they are relevant for the following discussion. See [11] for more details.

2.1 Evaluation Semantics

The semantics of core ALEA, after elimination of syntactic sugar, follows the approach of Moggi [7]: Syntax-directed rules for big-step evaluation are given, and together form an evaluation function, as a morphism in the Kleisli category of a monad that captures the effects of evaluation. Thus, the two evaluation modes of ALEA (static-probabilistic and dynamic-sampling) are quite similar, sharing most of the framework and rules, but differ in the underlying monad.

Since ALEA is not Turing-complete, and generally forbids partially defined, divergent or recursive expressions, the category of sets and total functions suffices as a base. Furthermore, the 'administrative' natural transformations associated with the semantic monads are all computable. Hence ALEA programs are always amenable to brute-force evaluation: Not only dynamic, but also static evaluation is *complete*, even though combinatorial explosion may lead to practically intractable complexity.

For static evaluation to a probability distribution of outcomes, the finitely supported rational-valued distributions are the appropriate effect monad. They come with rather beneficial algebraic properties; namely, they form a commutative strong (or symmetric monoidal) monad that yields a unique canonical interpretation of terms, and all of the monad operations are straightforwardly computable.

For dynamic evaluation to a pseudo-random sample, a state monad that contains a pseudo-random number generator (PRNG) is the appropriate effect monad. This one is not commutative, such that the actually sampled outcome depends on the order of evaluation of non-overlapping subterms. Thus, the evaluation function takes a random seed (initial state) as an additional parameter and is finitely set-valued. However, given an ideal PRNG, the probability of outcomes averaged over all seeds would converge to the statically calculated distribution.

2.2 Types and Functions

A strong and rich but mostly implicit type system ensures that evaluation of ALEA expressions does not go wrong. The (sub-)type system is organized as a lattice with top and bottom types, and generally structural subtyping rules. Expressions are assigned a most specific type by simple bottom-up inference.

2.2.1 The Type Hierarchy. There is a hierarchy of four number types, namely rationals, integers, naturals and Booleans, $\mathbb{Q} \supset \mathbb{Z} \supset \mathbb{N} \supset \mathbb{B}$. The unbounded number types include a special *not-a-number* value `NaN` for undefined arithmetics, with IEEE 754-like semantics. The Booleans are just the numbers $\{0, 1\}$.

For aggregate data, there are homogenenous finite collection types. They are distinguished by kind (list, bag, set), nonemptiness, and element type. Subtyping among collections is structural; for example, the type $[\mathbb{N}+]$ of nonempty lists of naturals is a proper subtype of the type $[\mathbb{Z}*]$ of possibly empty lists of integers.

Non-recursive algebraic datatypes can be constructed freely from record and tagged union types. The empty record type () serves as the unit type. A tagged union of units serves as an enumerated type. Subtyping among records and tagged unions is structural; for example the record type $(x\colon\mathbb{N}, y\colon\mathbb{N}, z\colon\mathbb{N})$ is a subtype of $(x\colon\mathbb{N}, z\colon\mathbb{Q})$, whereas the enumerated type $\{@\mathsf{one}(), @\mathsf{many}()\}$ is a subtype of $\{@\mathsf{one}(), @\mathsf{two}(), @\mathsf{many}()\}$. The latter rule implies that tags can be used ad-hoc, without committing to a particular algebraic datatype; the most specific type of a tagging expression $@t(a)$ is the singular union $\{@t(A)\}$ where A is the most specific type of the argument a. However, tags are marked syntactically with @ to distinguish them from variable/function identifiers.

The semantics of ALEA types is set-theoretic. Every type T is assigned an extension $[\![T]\!] \in \mathcal{P}(\mathcal{U})$, a subset of the untyped data universe $\mathcal{U}$, and the subtype relationship is *sound*; it entails the subset relationship of extensions:

$$T \sqsubseteq U \implies [\![T]\!] \subseteq [\![U]\!]$$

2.2.2 Function Type Signatures. Functions in ALEA are overloaded in the sense that they may admit more than one type signature in various ways. For example, the operation + is highly overloaded. On the one hand, as arithmetic addition it exhibits homomorphic overloading [9] on the number types. On the other hand, as list concatenation it exhibits parametric overloading on the element type T, as well as nonemptiness overloading in the sense that a concatenated list is possibly empty only if both constituents are.

$$(+) : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \qquad (+) : [T*] \times [T*] \to [T*]$$
$$(+) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \qquad (+) : [T*] \times [T+] \to [T+]$$
$$(+) : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q} \qquad (+) : [T+] \times [T*] \to [T+]$$

In order to make type inference in the presence of function overloading tractable, ALEA requires overloaded functions to be *coherent* and *compact*:

Coherence means that each function identifier f is associated with a single partial function on the untyped universe, $f : \mathcal{U} \twoheadrightarrow \mathcal{U}$. Each type signature $f : A \to B$ is merely a contract that the function f is total on the extension of the domain type A, and yields values from the extension of the range type B:

$$a \in [\![A]\!] \implies f(a) \in [\![B]\!]$$

Compactness means that for each function f a procedure is known that, for each concrete application with known argument type, selects a finite set of matching type signatures that capture all relevant information. To be precise, given an argument a with most specific type A', find n type signatures $f : A_1 \to B_1, \ldots, f : A_n \to B_n$ that all match (on the domain side) in the sense $A' \sqsubseteq A_i$. Then assign the application $f(a)$ the greatest lower bound of ranges $B' = \bigwedge B_i$, since all of the contracts apply simultaneously. In order for B' to be most specific, there must not be another set of matching type signatures, finite or infinite, that results in a more specific bound $B'' \sqsubset B'$.

Together, coherence and compactness cover the most common cases of polymorphism: Functions with disjoint domains can be combined unambiguously in ad-hoc overloading by taking their union. A family of functions with parametric overloading can be realized by giving a natural implementation [12]; since all type expressions are covariant, the unique most specific type instantiation is easy to find. Functions in homomorphic overloading over a chain of types (subalgebras) can be realized by giving the just largest implementation.

A consequence of coherent homomorphic overloading is that ALEA cannot have exact division on rationals and truncated division on integers as overloaded variants of the same operator: A function that has both type signatures $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ must not distinguish an argument of integer type and an argument of rational type that is accidentally an integer. Therefore, ALEA writes exact division as `/` and truncated division as `//`, similar to dynamically typed languages such as PYTHON, albeit for different reasons.

3 Reference Implementation

The reference implementation of ALEA reflects the structure of evaluation semantics faithfully. By traversal of an abstract syntax tree (AST) for a core ALEA program, either a probability distribution of outcomes or a pseudo-random sample (with side effects on the supplied PRNG) can be calculated.

The host language is JAVA. The interpreter uses the JAVA standard library in a straightforward way to implement ALEA data: Exact numbers are built on `java.math.BigInteger`, with rationals represented as fractions, and collections use the framework `java.util`, with bags represented as positive integer-valued maps.

While the reference interpreter is useful as proof of concept and for simple demonstrations, there are some evident performance issues that arise from its naïvely puristic approach. Performance is problematic for both evaluation modes in different ways: Calculating the probability distribution of all possible outcomes of a complex program is a potentially rather long-running computation. By comparison, pseudo-random sampling is a much simpler task, but one that would have to be repeated very often if used in Monte Carlo simulations or game engines.

Clearly, getting rid of interpretation overhead is desirable in each scenario. Hence an optimizing compiler is being developed. We envision the use of compiled ALEA for scalable applications, while retaining the reference interpreter for tracing and debugging purposes, and as a light-weight test oracle.

3.1 Causes of Overhead

We have identified a number of issues with a naïve interpretation that follows the semantic rules literally. Some of these are theoretical in nature, others technical. A few are worthwhile to solve directly as refinements to the reference interpreter, whereas the majority require deep optimizations best left to a compiler.

3.1.1 Node Classification. The most obvious technical overhead of an interpreter based on AST traversal is its data-driven control flow: At each step, the current AST node must be classified dynamically, in order to select the appropriate evaluation rule. This classification may not be a particularly expensive operation when done once (for example, if the interpreter builds on the double-dispatch *Visitor* pattern [2], classification amounts to two nested virtual method calls), but needs to be repeated every time that particular expression is evaluated. Since ALEA deals heavily with iterative constructs in both explicit (collections) and implicit (distributions) form, that cause the same subexpression to be evaluated many times, the overhead from naïve AST node classification is significant.

3.1.2 Function Overloading Resolution. Type inference only checks that all possible argument values for a function application are within the domain of the untyped coherent implementation. In practice, functions may come with ad-hoc polymorphism, and different code may need to be executed for different classes of values; thus a dynamic classification of the untyped argument is required for selecting the appropriate code variant.

This issue is similar to those faced by implementations of dynamically typed languages, since the concept of coherent overloading imposes a statically untyped view. Here, it could be resolved by

proactively changing the binding of a function name to a specialized implementation during type inference.

3.1.3 Constant Expressions. For the most part, ALEA is purely functional, deterministic and strongly terminating. The exceptions, random primitives, are syntactically marked. Thus it is both safe and easy to fold (evaluate) constant expressions ahead of time.

This cause of overhead can and should reasonably be addressed already in the reference interpreter. Since both constant folding and type inference are strictly bottom-up, they can even operate in lockstep, such that a folded constant expression can be assigned a more specific type by examining its actual value. For example, `(-4)/(-2)` could be recognized post hoc as a natural number, and in general the need for complex literals would be greatly reduced. Since type refinement changes the set of well-typed programs, the rules for constant folding need to be explicit. We plan to specify and mandate constant folding in an upcoming version of the language.

3.1.4 Bag Multiplicities. Bags are an intermediate collection kind between lists and sets: Like lists, they account for multiple occurrences of the same element, but like sets they do not account for element order. [5] Bags play a central role in many typical applications for ALEA, much more so than for other functional programming languages. A composite random experiment often performs several simpler ones and aggregates the results. If the order is of no importance, as in *"roll identical dice"*, then using bags instead of lists can remove a lot of combinatorial overhead, in particular from static analysis. For example, the rolls $\langle 2, 2, 5\rangle$, $\langle 2, 5, 2\rangle$ and $\langle 5, 2, 2\rangle$ are equivalent and can be lumped together as a single possible outcome.

Furthermore, *map*, *filter* and *reduce* operation on bags can exploit the knowledge that some element occurs multiple times. For simplicity, consider a bag that contains just one element n times:

$$b = \langle \underbrace{c, \ldots, c}_{n} \rangle$$

Filtering this bag as in `⟨ x ← b | p ⟩` requires just one evaluation of `(x := c; p)` in any case.

Likewise, mapping over this bag as in `⟨ e | x ← b ⟩` requires just one evaluation of `(x := c; e)`, if `e` is deterministic; otherwise, if `e` contains randomness primitives, such as `~uniform` in the introductory example, the evaluation is to be repeated n times with stochastically independent outcomes.

Reducing this bag as in `f(b)` requires $O(\log n)$ applications of `f`. Since all operations eligible for reduction must be associative, the result can be computed efficiently using the algorithm known as 'exponentiation by squaring'.

3.1.5 Monad Semantics for Comprehensions. Comprehension notation is provided as syntactic sugar in ALEA. The reduction to core constructs follows the approach of Wadler [14], recognizing that each collection kind is a monad with zero. While quite elegant as theoretical semantics, in practice this approach suffers from the issue that a great number of small intermediate collections is implied.

Generator clauses are reduced to an iteration primitive, a first-order counterpart of HASKELL's monad binding operator ≫=. We write it in the following form, with the appropriate bracket shape for the desired kind of collection and marked with `!`:

```
[ es | x ← C ]!
```

In primitive form, there is exactly one generator clause, no filter clause, and an expression on the left hand side that yields not a single element, but a subcollection of arbitrary size. The semantics can be explained in terms of the collection monad multiplication operation `join` which concatenates a collection of collections:

```
⇝ join [(x := v1; es), ..., (x := vn; es)]
```

Consider the first line of the introductory example:

```
dice := ⟨ ~uniform{1..6} | _ ← ⟨1..n⟩ ⟩
```

The comprehension expression is already nearly in primitive form; only the singular element-forming expression needs to be wrapped in a singleton bag:

```
dice := ⟨ ⟨~uniform{1..6}⟩ | _ ← ⟨1..n⟩ ⟩!
```

Interpreted naïvely, this translation prescribes the (avoidable) creation of n intermediate singleton bags. The situation is even worse for filter clauses. Consider the second line of the introductory example:

```
score := (+)⟨ d ← dice | d \\ 2 = 1 ⟩
```

This notation uses some syntactic sugar to mimic the conventions of mathematical set-builder notation. In HASKELL-style comprehension notation, the generator is placed to the right of the bar, with a trivial implied element expression remaining on the left:

```
score := (+)⟨ d | d ← dice; d \\ 2 = 1 ⟩
```

The reduction to core ALEA represents filter clauses as *if-then-else* expressions:

```
score := (+)⟨ (d \\ 2 = 1) ? ⟨d⟩ : ⟨⟩ | d ← dice ⟩!
```

As a result, every odd element of `dice` gets wrapped in a singleton bag while every even one gets replaced with an empty bag, and then the resulting bags, of average size between zero and one, are concatenated.

3.1.6 Immediate Reduction. Several idioms in ALEA encourage the building of a collection with a comprehension, only to reduce it immediately with a monoid operation. In the second line of the introductory example, a bag is constructed and used only to sum its elements.

An even more drastic waste of intermediate data structures arises in counting: Since the Booleans are just the numbers $\{0, 1\}$, it is natural in ALEA to express the counting of elements that satisfy a Boolean property as the sum of mapped (not filtered!) property results, seen as natural numbers:

```
count := (+)⟨ d \\ 2 = 1 | d ← dice ⟩
```

A näive interpretation replaces every even/odd element of `dice` by the constant bag `⟨0⟩`/`⟨1⟩`, respectively, concatenates the bags, and sums the elements; the number of intermediate data structures exceeds the result value even in the best case.

4 Compilation

The newly developed compiler complements the ALEA reference interpreter and is currently hosted in the same IDE. It acts as a just-in-time compiler, translating an AST on the fly to JVM bytecode, the executable format of the host platform. This bytecode is initially interpreted at a lower level by the JVM, but for long-running or often-repeated computations it is bound to become 'hot' eventually, and attract the attention of the JVM just-in-time compiler to native machine code.

For generating the bytecode, and loading and instantiating the newly defined class right away, the embedded DSL LLJAVA-LIVE [10] is used. This ensures that no external tools or storage are required; code generation operates entirely on pure JAVA libraries and local JVM heap memory.

4.1 Causes of Overhead Revisited

This section parallels section 3.1, and describes how compilation can address each cause of overhead.

4.1.1 Node Classification. By generating code from an AST, the issues of dynamic traversal and node classification are of course eliminated entirely, and reduced to the native control flow of the target language, JVM bytecode. The whole AST structure and the traversal glue code are flattened into a sequence of JVM instructions that define a single monolithic method.

4.1.2 Function Overloading Resolution. The ALEA compiler uses the results of type inference to rebind library function applications to specialized variants, if available. This eliminates not only the runtime overhead of choosing between ad-hoc polymorphic variants, but allows allows for primitive data, particularly numbers, to be encoded in more native and less boxed ways than as mere subsets of the untyped universe.

Fully rolled out, this optimization comes at a steep price: Interpreted and compiled code require technically different, but semantically equivalent library implementations. This is further compounded by constant folding during compilation, which does not truly belong to either world. In the current stage of development of the prototype compiler, library functions are being specialized incrementally as required by examples, while for the rest the untyped implementation is used as a suboptimal fallback.

4.1.3 Constant Expressions. For some expressions, constant folding during the type inference phase is of dubious value. Consider a comprehension of the form `[e | x ← l]`, where `e` is a complex expression and `l = [v1, ..., vn]` is a constant list of some length $n \gg 1$. While it is theoretically sound to replace the comprehension with the 'folded' expression `[(x := v1; e), ..., (x := vn; e)]`, this substitution amplifies the complexity of `e` n-fold.

The compiler is free to hoist such constant-valued but unfolded expressions out of the regular execution path, and compute and store their value ahead of time or lazily on demand. Thus, the collection value can be computed using iteration concepts of the target language, without having to literally replicate the element-forming expression `e`; but its value, once computed and stored, need not be recomputed for repeated evaluations of the whole.

4.1.4 Bag Multiplicities. The optimizations for multiple occurrences of an element in a bag processed by *filter* and *reduce* operations are local in nature, and hence easily implemented already in the reference interpreter. The optimization for *map* requires information about deterministic subexpressions. These can be identified by simple bottom-up propagation, implemented as an additional analysis pass in the compiler.

4.1.5 Monad Semantics for Comprehensions. The reduction of comprehension expressions to natural monad operations is not intrinsically bad. If the implementations of collections in the target language are *fractal* in nature, such that the natural and efficient way to form a larger data structure is to compose smaller ones, then there is no clear case for optimization. However, this is decidedly not the case for the JVM backend, or any other translation to an object-oriented target language for that matter.

Collections in OO languages are essentially mutable, and hence optimized for individual–incremental construction: Creating a new collection object may incur a substantial overhead, whereas adding elements one by one to an existing collection is fairly cheap on average. The translation scheme used by a compiler needs to take such characteristics into account in order to generate efficient target code. In the OO case, side effects on a context-defined collection object are vastly preferable to masses of subcollections containing at most one element each.

Fortunately, the structure of ALEA core allows for a general optimization strategy that covers the patterns resulting from comprehensions, and possibly other expressions that are accidentally similar. Positions of subexpression in the abstract syntax can be classified as either *argument* or *result* positions: The branches of (Boolean, other numeric, and tagged) case distinctions, the bodies of *let* expressions, and the left hand sides of iteration primitives are result positions, all other are argument positions. Note that result positions are called so because they may determine (parts of) the result of the containing expression, and are hence necessarily of the same type.

For subexpressions in argument positions, the compiled code must inject the result value into the data flow. On the JVM, this generally means pushing the value onto the operand stack. For subexpressions in result positions, the default action is the same. However, when compiling code for an iteration primitive, a different behavior can be set and propagated downwards to transitive result positions: If the subexpression result is a collection (necessarily of the same kind and element type) whose size and element-forming expressions are statically known, then these are added one-by-one to a collection-under-construction (CuC) created for the iteration primitive. Otherwise, the subcollection is computed and copied to the CuC one-by-one using a loop (or a framework method `addAll`). The former clause of this strategy eliminates all the synthetic subcollections of size one, or even zero, incurred by *map* and *filter* operations, while the latter is a safe fallback for other subexpressions.

Figure 2 shows a simple example computing the non-decreasing subset of the Cartesian product of two sets `L` and `R`: The evident user-level expression (A) is reduced to ALEA core (B). For clarity, implementations on the target platform are exemplified for deterministic

evaluation only; both probability distributions and pseudo-random sampling require some extra machinery.

A naïve Java-pseudocode implementation (C) is given (the actual compiler generates JVM bytecode directly), which follows the structure of the core AST one to one. It summarizes the effective computation run by the reference interpreter, dynamic AST traversal and related technical issues aside; an optimizing compiler should not produce code that performs worse than this baseline.

For contrast, an optimized, context-aware imperative implementation (D) is given, which creates a CuC for the outermost iteration primitive, and propagates the task of adding elements to all transitive argument positions. Thereby the singleton and empty subcollections arising from *map* and *filter* clauses, respectively, are entirely eliminated. Since, for safety reasons, collection values should not be mutable after they have been computed, a *Builder* pattern [2] is used to separate the (mutable) construction phase from the (immutable) use phase of the collection.

4.1.6 Immediate Reduction. If a collection created from a comprehension is used only once for a reduction, then the intermediate data structure can be eliminated altogether. This situation can be detected locally, if the comprehension expression is literally nested within the reduction expression. We assume that *let* bindings that are used at most once are eliminated by means of substitution; this reduces most realistic cases of single use to literal nesting.

Imperative elimination of reduced collections works by preempting the rule described in the preceding section: Instead of having subexpressions in result positions contribute to a CuC, make them update an accumulator using the binary reduction operation. If the source collection is possibly empty, initialize the accumulator with the appropriate neutral element; otherwise, initialize the accumulator with the first element.

Figure 3 shows the compilation scheme applied to the introductory example, in analogy to Figure 2. The optimized implementation exemplifies the hoisting of the loop-invariant constant value `s1` as discussed in section 4.1.3, as well as the replacement of a bag builder with the accumulator `accu3` for immediate reduction.

4.2 Compiling for Pseudo-Random Sampling

The state monad underlying the semantics for dynamic pseudo-random sampling causes no essential effort in an imperative implementation. Merely holding a reference to a fixed PRNG object, which encapsulates its own mutable state, throughout the evaluation, and tapping into randomness as a side effect, is sufficient.

Note that the semantics are not deterministic; the order of calls to the PRNG from independent subexpressions or from iterations on unordered collections is unspecified. Thus, interpreted and compiled code may differ in the samples that are actually produced, even for the same random seed. The two are only guaranteed to be equivalent in ideal probability (running a perfect PRNG with truly random seed).

4.3 Compiling for Static Analysis

The distribution monad underlying the semantics for static analysis is significantly more complex. Fortunately, it is similar enough to the collection monads for the same basic optimizing compilation techniques to be applied.

In a typical Alea expression, only a few subexpressions are truly random; most operations are determistic in nature. A naïve implementation would calculate probability distributions locally for all AST nodes, wrap the results of deterministic computations in 'singleton' Dirac distributions, and combine the distributions for disjoint subexpressions stochastically independently (by cross-multiplication).

A strategy that is better adopted to the target environment allocates a distribution-under-construction (DuC), and has all non-iterative subexpressions, also those in argument positions, contribute probability-weighted results. Instead of multiplying distributions out after creation, the probability tree is traversed only virtually, accumulating multiplicative local probability values along the path. For every random subexpression, there needs to be a loop over the possible outcomes and their local probabilities, with the remainder of evaluation nested within the loop body. For all of the deterministic operations, however, the loop is known to have exactly one iteration (with probability one), and can be eliminated.

Figure 4 shows the compilation scheme applied to a simple example that computes the average of two dice. The optimized implementation exemplifies the loops introduced for random subexpression, and the linear code for deterministic follow-up computations.

For iterative subexpressions (comprehensions), the situation is more complicated: A collection of distributions (of elements) must be transposed into a distribution of collections. In theory, this operation is specified naturally by a distributive law between the respective monads for lists and bags; see [6] for a recent presentation. By contrast, for sets there is no such law [15]. This fact matches well with semantic requirements for expressions that are not referentially transparent: Set iteration `[]!` is handled as if it were about bags at first, and only the result coerced to a set by forgetting multiplicities. For instance, in an expression

```
{ {~uniform{1..6}, ~uniform{1..6}} | i ← S }!
```

there should be exactly two stochastically independent die rolls for each element of S, whereas a naïve set reading would (wrongly) imply both that identical element expressions could be unified before evaluation, and that the same element expression could be re-evaluated idempotently.

In practice, a statically optimized implementation as above is not quite straightforward, since the number of nested loops required depends on the size of the collection. The current Alea compiler uses a dynamic iterative auxiliary function; possibly more efficient approaches are a topic for future research.

5 Empirical Measurement

The compilation strategy has been evaluated in practice on a number of examples. Since the compiler is currently under construction, 'compiled' Alea programs have been simulated by manual coding in Java. For a fair comparison, the hand-written code uses only optimizations that are realistically expected to be applied automatically in the compiler.[1]

The Alea compiler is designed to produce JVM bytecode that can be optimized further by the JVM just-in-time compiler (JIT). In

[1] We expect to be able to substitute actual compiled code in time for the post-proceedings publication of this paper.

A: User-Level Expression

```
{ (a, b) | a ← L; b ← R; a ≤ b }
```

B: Core Expression

```
{ { (a ≤ b) ? {(a, b)} : {} | b ← R }! | a ← L }!
```

C: Monadic Implementation (Pseudo-Java)

```
return bind(L, a → bind(R, b → leq(a, b) ? singleton(tuple(a, b)) : emptySet()));
```

D: Imperative Implementation (Pseudo-Java)

```
SetBuilder⟨Pair⟨Rational⟩⟩ cuc = new SetBuilder⟨⟩();
for (Rational a : L) {
    for (Rational b : R) {        // result positions
        if (leq(a, b)) {
            cuc.add(pair(a, b));  // singleton
        }
        else {                    // empty
        }
    }
}
return cuc.build();
```

Figure 2: Compiling a Filtered Cartesian Product

A: User-Level Expression

```
dice := ⟨ ~uniform{1..6} | _ ← ⟨1..n⟩ ⟩;
score := (+)⟨ d ← dice | d \\ 2 = 1 ⟩
```

B: Core Expression

```
(+)⟨ (d \\ 2 = 1) ? ⟨d⟩ : ⟨⟩ | d ← ⟨ ⟨~uniform{1..6}⟩ | _ ← ⟨1..n⟩ ⟩! ⟩!
```

C: Monadic Implementation (Pseudo-Java)

```
return reduce(Natural::add, 0,
              bind(bind(bagRange(1, n), __ → uniform(setRange(1, 6))),
                   d → eq(mod(d, 2), 1) ? singleton(d) : emptyBag()));
```

D: Imperative Implementation (Pseudo-Java)

```
Set⟨Natural⟩ s1 = setRange(1, 6);                    // hoisted constant
BagBuilder⟨Natural⟩ cuc2 = new BagBuilder⟨⟩();
for (Natural _tmp2 : bagRange(1, n)) {
    cuc2.add(uniform(s1));
}
Bag⟨Natural⟩ s2 = cuc2.build();
Natural accu3 = 0;                                   // immediate reduction
for (BagElement⟨Natural⟩ d : s2) {
    if (eq(mod(d.getValue(), 2), 1)) {               // boolean to Natural
        accu3 = accu3.add(1 * d.getMultiplicity());
    }
    else {
        // accu3 = accu3.add(0 * d.getMultiplicity());
    }
}
return accu3;
```

Figure 3: Compiling an Immediate Reduction

A/B: User-Level/Core Expression

```
(~uniform{1..6} + ~uniform{1..6}) / 2
```

C: Monadic Implementation (Pseudo-Java)

```
return product(uniformDistr(setRange(1, 6)),
               uniformDistr(setRange(1, 6)))
       .map(Rational::add)
       .map(x → x.divide(2));
```

D: Imperative Implementation (Pseudo-Java)

```
DistributionBuilder⟨Natural⟩ duc = new DistributionBuilder⟨⟩();
Set⟨Natural⟩ s1 = setRange(1, 6);
for (Natural v1 : s1) {               // random
    Probability p1 = 1/6;
    for (Natural v2 : s1) {           // random
        Probability p2 = p1 * 1/6;
        Natural v3 = v1.add(v2);      // deterministic
        Probability p3 = p2;          // omit * 1
        Natural v4 = v3.divide(2);    // deterministic
        Probability p4 = p3;          // omit * 1
        duc.add(v4, p4);
    }
}
return duc.build();
```

Figure 4: Compiling a Probability Distribution

particular, the Alea compiler keeps the target code small by realizing complex operations as static method calls to a runtime library, which are expected to be inlined by the JIT. We have measured only the performance of the overall result.

All measurements have been performed on a machine with a Core i7-12700H 20-core CPU and 16 GiB of RAM, running Ubuntu 24.04.02 and OpenJDK 21.0.7. Reported times are wallclock times measured with `System.nanoTime` precision, for Alea running on a JVM as a single thread, and thus utilizing a single CPU core.

For each test case, the JIT has first been warmed up by running the program under test repeatedly for several seconds. Then, timing data for the same number of repetitions are collected. To mitigate the influence of outliers caused by rare events such as large-scale garbage-collection (GC) pauses, robust statistics are used: All reported figures are median values, and the dispersion of data has been controlled by median absolute deviation (MAD), which is less than 6% for each data set.

We have observed some metastable variability between runs of the experimental setup. Without detailed investigation, it is unclear whether this is due to the JIT, the GC, and/or the heavy use of hash tables in Alea data. However, the resulting dispersion is typically less than 10%, and does not invalidate any of the findings below.

5.1 Motivating Example Revisited

Figure 5 summarizes the measurements for a family of instances of the introductory example. The size parameter n (the number of dice) ranges from 1 to 10. The program is fixed; code has not been specialized for a particular value of n.

The top row shows measurements for the dynamic evaluation mode, the computation of a single pseudo-random sample. Both interpreted and compiled code show the expected linear growth of running times. Speedup by compilation is consistently between 12 and 15.

The bottom row shows measurements for the static evaluation mode, the computation of the overall probability distribution of outcomes. Note the logarithmic time scale. Speedup by compilation is only significant for small instances; for $n = 10$ it is down to a meagre 1.04. Nevertheless, this is not actually disappointing, considering the particular nature of the example:

- For larger n, the effort is dominated by the construction of all possible values of `dice`. The interpreter and the compiler use not only the same algorithm, but also essentially the same implementation. The overhead of traversing the tiny program is negligible; hence measurable speedup can only come from the subsequent summing operation.
- The growth curve in logarithmic scaling is apparently flattening, due to the fact that there are significantly less bags than (the exponentially many) lists of n dice rolls; permutations are collapsed into a single representative. While this reduction of combinatorial pressure is beneficial for the theoretical complexity of an Alea program, it compresses the potential for optimization in a compiler. In particular, approaches that try to eliminate intermediate bags by recursive enumeration of elements are generally doomed, since they operate on virtual lists and cannot straightforwardly benefit from the collapsing of duplicates up to permutation.

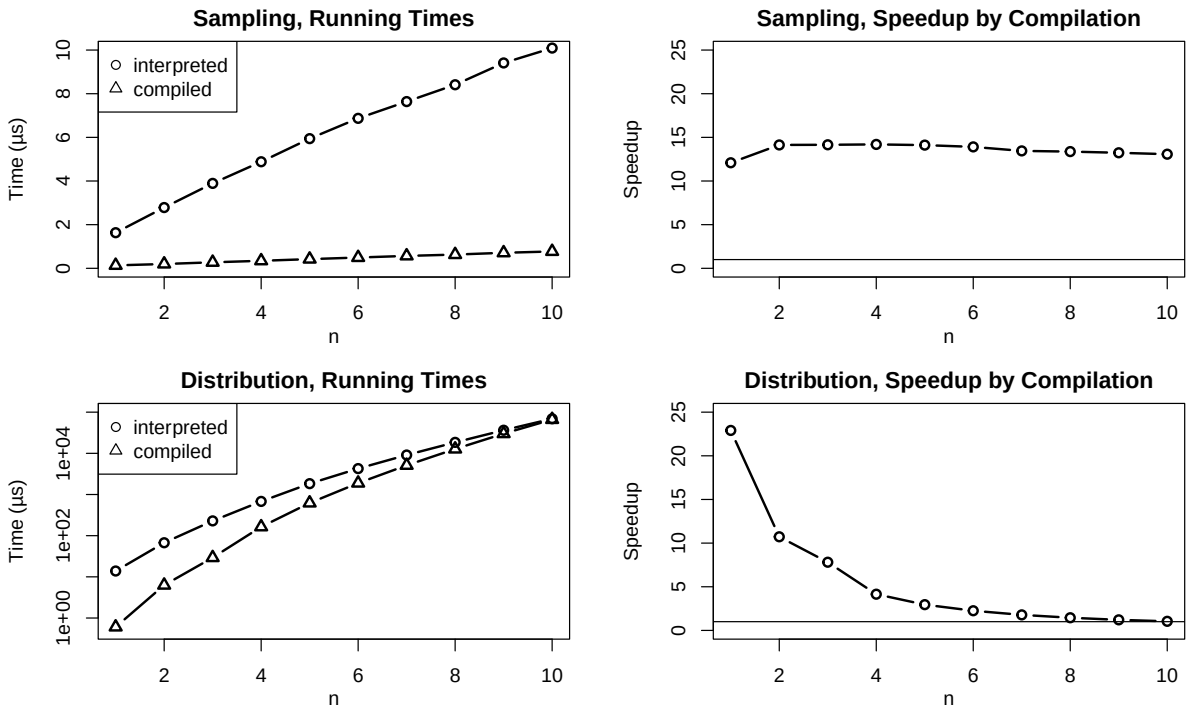


Figure 5: Compiling the Introductory Example

5.2 Complex Example

To evaluate the effect of more complex deterministic computations applied to a combinatorial bag situation, we have generalized the motivating example from [11]:

> *Roll n ten-sided dice. For every die that shows a ten, roll another and add it to the pool. Count the number of dice that show values greater than five. Compare to the number of dice that show a one. If the difference is positive, you win by that amount. If the difference is zero or negative, you lose. If there are no values greater than five but there is a one, you lose badly.*

See Figure 6 for the ALEA source code. Figure 7 summarizes the measurements. The observed behavior is qualitatively similar to the previous case, but slightly more favorable to the compiler. Sampling receives a consistent speedup of about 15 from compilation. The calculation of distributions is improved significantly only for small instances. As expected, the speedup stagnates at a higher level than for the previous case; for $n = 7$ the interpreted time is 14.66 s, and the compiled time is 6.65 s, so the speedup is still 2.2. Note that compilation is expected to take less than 0.1 s; it easily pays off even with a single use.

5.3 Highly Optimizable Example

As an example where optimizing compilation is expected to make a dramatic difference, consider the average of two random integers chosen uniformly from $\{1, \ldots, n\}$, a parametric generalization of the example discussed in section 4.3.

```
dice1   := ⟨ ~uniform{1..10} | _ ← ⟨1..n⟩ ⟩;
tens    := ⟨ d | d ← dice1; d = 10 ⟩;
dice2   := ⟨ ~uniform{1..10} | _ ← tens ⟩;
dice    := dice1 + dice2;
succs   := (+)⟨ d > 5 | d ← dice ⟩;
fails   := (+)⟨ d = 1 | d ← dice ⟩;
diff    := succs - fails;
verdict := diff > 0 ? @win(diff)
           : (succs = 0 && fails > 0
              ? @botch : @lose)
```

Figure 6: Complex Bag-of-Dice Example [11]

```
(~uniform{1..n} + ~uniform{1..n}) / 2
```

Figure 8 summarizes the measurements for a family of instances. For pseudo-random sampling, the speedup is consistently greater than 20. For larger n, the naïve running time is dominated by the creation of the set `{1..n}`. The compiler evaluates it entirely during constant folding, and sets up a lookup array for constant-time sampling. (In the upcoming version, the interpreter will also be able to benefit from mandatory constant folding at type-checking time.)

For static calculation of distributions, the compiler can eliminate intermediate distributions of size $O(n^2)$ which empirically take about $O(n^3)$ to construct. The resulting drop in the exponent of growth can be seen in the doubly logarithmical bottom-left plot.

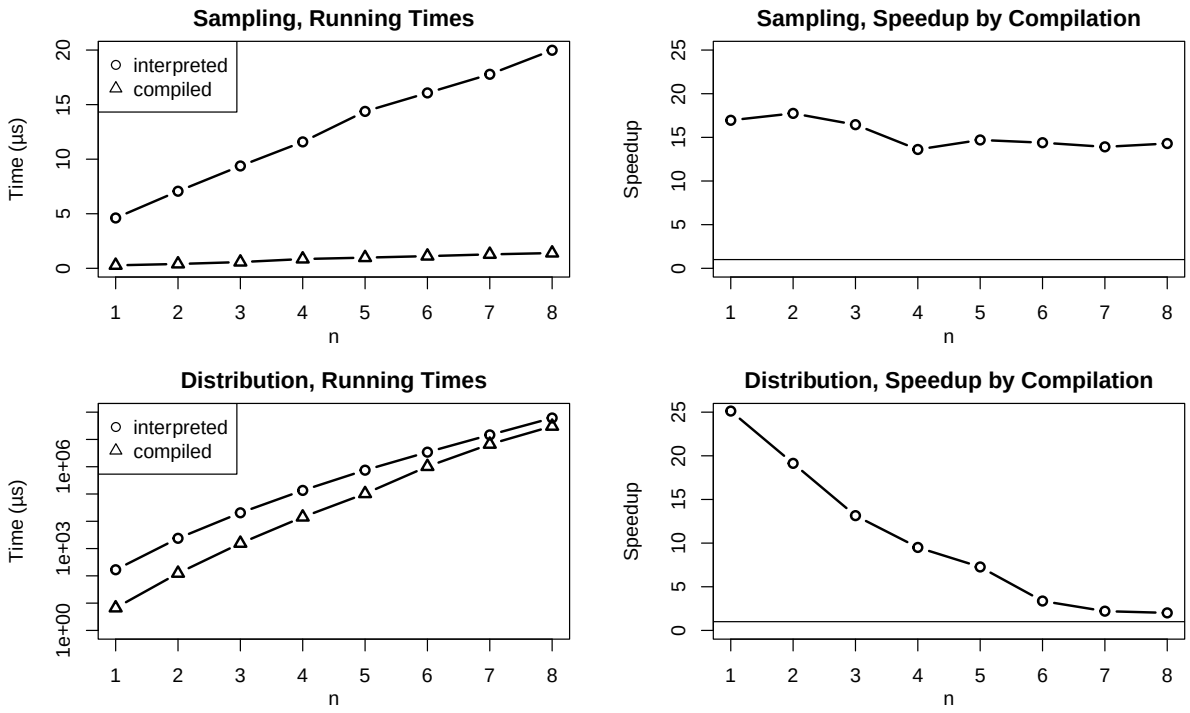


Figure 7: Compiling a Complex Bag-of-Dice Example

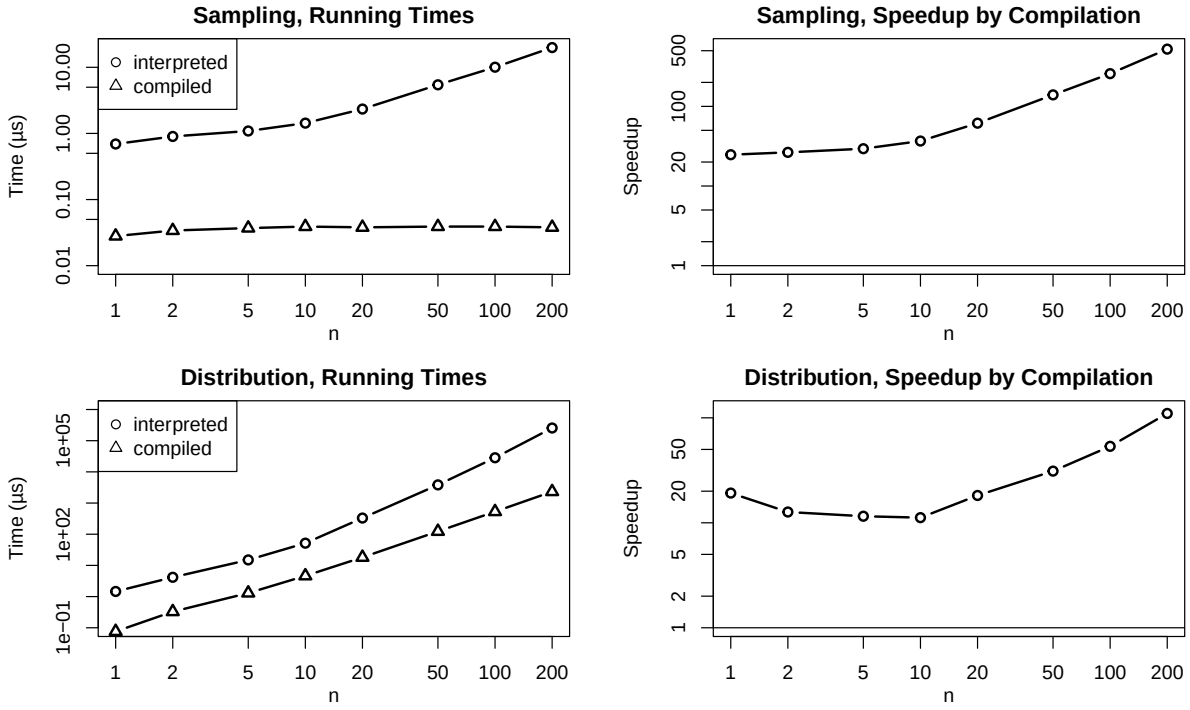


Figure 8: Compiling a Highly Optimizable Example

5.4 Resource Use During Compilation

Since the compiler is not yet fully operational, there are no actual measurements of the running time of compilation. However, from a case study using the same code generator library LLJava-live [10], we can infer that compilation of a simple program completes on the order of a few milliseconds (when the compiler is warmed up).

6 Conclusion

We have investigated the implementation of ALEA, a functional domain-specific language for stochastic experiments. ALEA stands out among stochastic languages due to its computable static semantics, which favors systematic bottom-up calculation over computer algebra heuristics. We have demonstrated how the structural subtype system and various monads structure the formal semantics of the language, in which ways naïve interpretation of the semantic rules is inefficient, and how to tackle these issues in a compiler.

Empirical measurements show consistent speedups for pseudo-random sampling of at least 10–20. In this evaluation mode, compilation pays off predictably if a large number of samples is to be served.

For the static evaluation mode resulting in overall distributions, results are more nuanced. Programs with a large proportion of deterministic computation can be optimized to great effect, whereas ironically, using bags as data structures squeezes out a lot of optimization potential already at the source level. The facts that the interpreter and compiler share the workhorse routines for interchanging collections and distributions, and that the same JIT acts on both environments, leave little room for improvement.

6.1 Related Work

There is a vast number of experimental languages that employ probabilistic constructs in some way. See [11] for an assessment of the unique position of ALEA in the design space.

6.1.1 Deforestation. The idea that intermediate data structures arising from playing by the rules of a functional language can and should be eliminated is not new at all; it has been studied as *deforestation* [13]. In typical presentations [4, 8], deforestation is conveniently formulated as a rewriting procedure, where source and target language are of the same nature.

In the ALEA compiler, the optimization is coupled with the paradigm shift to a target platform with imperative data structures. We feel that this makes the technique a bit less clear, but at the same time even more urgently required.

6.1.2 Java Stream Framework. The Java stream framework is an approach to object-oriented data structures that takes a leaf out of the functional book: In this framework, streams are abstract element sources. A stream can retrieve data elements from a collection or compute them on the fly. Stream computation is organized into linear pipelines with *map* and *filter* operations. Batch computations are performed lazily element-by-element, and possibly in parallel. Streams are consumed and either collected in a new collection, or reduced to a result value.

Converting a collection to a stream, performing computations that abstract from storage, and deciding on a target collection or reduction operation only in the end is also a way to get rid of intermediate data structures. It differs from deforestation in so far as streams just enumerate elements without further structure information; lists and sets are straightforward to handle by enumeration, but bags (elements with multiplicities) and distributions (elements with probabilities) are not.

6.2 Future Work

The development of the syntax and semantics of ALEA, as well as the reference interpreter and the compiler, are in progress.

6.2.1 User-Defined Abstractions. The current version of ALEA program syntax is restricted to expressions that invoke predefined library operations. The addition of user-defined deterministic functions and random distributions is foreseen to pose additional problems for a compiler, such as stable ABI design and inter-procedural optimization.

6.2.2 Modular Optimization Passes. The current prototype of the ALEA compiler is, apart from the AST-constructing frontend that is shared with the interpreter, essentially a single pass. While this makes for a simple and compact software architecture, it is technically difficult to isolate the various optimization techniques, and quantify their individual impact independently. A refactored, more modular compiler, acting on one or more intermediate representations, would improve not only maintainability, but also empirical measurability.

6.2.3 Parallel Evaluation. Even though Java streams are no direct match for the needs of ALEA, due to their lack of direct support for the most prominent data structures, bags and distributions, their ability to parallelize batch computations under the hood is an attractive feature. The current ALEA interpreter and compiler do not support the usage of more than one core, but the majority of ALEA iterations are in fact embarassingly parallel, and that situation should be exploited somehow.

6.2.4 Nested Collections. In the sense of section 4.1.5, generator clauses of a comprehension are *not* result positions. As a consequence, comprehensions nested to the right are not subject to deforestation (compare also section 4.1.3). This may be surprising, because for lists, which are the dominant collection shape in most functional programming languages, there is no obvious reason. A nested list comprehension

```
[ e | x ← [f | y ← L] ]
```

can be flattened with no downsides:

```
[ (x := f; e) | y ← L ]
```

The situation for bags and sets, however, is more complicated: The intermediate collection can be significantly simpler than the original source `L`, if the element-forming expression `f` happens to be non-injective. For bags, duplicate intermediate elements are lumped together and can benefit from the optimizations discussed in section 3.1.4 when it comes to the iterated evaluation of `e`. For sets, duplicates are even eliminated altogether by idempotence. See also the discussion on the number of possible collections in section 5.1. For deciding which cases are improvable by deforestation, or perhaps by specialized techniques, more research is needed.

Acknowledgments

Anonymous reviewers have contributed valuable remarks and suggestions.

References

[1] Camila Caiado and Pushpa Rathie. Polynomial coefficients and distribution of the sum of discrete uniform variables. (2007). Retrieved June 13, 2025 from https://durham-repository.worktribe.com/output/1158118.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN: 0-201-63361-2.

[3] Jeremy Gibbons. 2016. Comprehending ringads. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, (Eds.) Springer International Publishing, Cham, 132–151. doi:10.1007/978-3-319-30936-1_7.

[4] Andrew Gill, John Launchbury, and Simon Peyton Jones. 1993. A short cut to deforestation. In *Proceeding of the Conference on Functional Programming Languages and Computer Architecture*. ACM, 223–232. doi:10.1145/165180.165214.

[5] Paul F. Hoogendijk and Roland C. Backhouse. 1994. Relational programming laws in the tree, list, bag, set hierarchy. *Science of Computer Programming*, 22, 1-2, 67–105. doi:10.1016/0167-6423(94)90008-6.

[6] Dexter Kozen and Alexandra Silva. 2024. Multisets and distributions. In *Logics and Type Systems in Theory and Practice: Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday*. Venanzio Capretta, Robbert Krebbers, and Freek Wiedijk, (Eds.) Springer Nature Switzerland, Cham, 168–187. ISBN: 978-3-031-61716-4. doi:10.1007/978-3-031-61716-4_11.

[7] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation*, 93, 1, 55–92. Selections from 1989 IEEE Symposium on Logic in Computer Science. doi:10.1016/0890-5401(91)90052-4.

[8] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *2001 Haskell Workshop*. ACM SIGPLAN. Retrieved Aug. 29, 2025 from https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/.

[9] Alex Shafarenko and Sven-Bodo Scholz. 2004. General homomorphic overloading. In *Proceedings of the 16th International Conference on Implementation and Application of Functional Languages*. Springer, Lübeck, Germany, 195–210. ISBN: 3540260943. doi:10.1007/11431664_12.

[10] Baltasar Trancón y Widemann and Markus Lepper. 2021. LLJava live at the loop: a case for heteroiconic staged meta-programming. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. Herbert Kuchen and Jeremy Singer, (Eds.) ACM, 113–126. doi:10.1145/3475738.3480942.

[11] Baltasar Trancón y Widemann and Markus Lepper. 2025. Alea iacta est: a declarative domain-specific language for manually performable random experiments. In *Proceedings 14th International Workshop on Trends in Functional Programming in Education* (Electronic Proceedings in Theoretical Computer Science). Rose Bohrer, (Ed.) Vol. 424. Open Publishing Association, 67–86. doi:10.4204/EPTCS.424.4.

[12] Philip Wadler. 1989. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (FPCA '89). Association for Computing Machinery, New York, NY, USA, 347–359. doi:10.1145/99370.99404.

[13] Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73, 2, 231–248. doi:10.1016/0304-3975(90)90147-A.

[14] Philip Wadler. 1992. Comprehending monads. *Mathematical Structures in Computer Science*, 2, 4, 461–493. doi:10.1017/S0960129500001560.

[15] Maaike Zwart and Dan Marsden. 2022. No-go theorems for distributive laws. *Logical Methods in Computer Science*, 18, 1. doi:10.46298/lmcs-18(1:13)2022.

Refinement-Types Driven Development: A study

Facundo Domínguez
Tweag
France
facundo.dominguez@tweag.io

Arnaud Spiwack
Tweag
France
arnaud.spiwack@tweag.io

Abstract

This paper advocates for the broader application of SMT solvers in everyday programming, challenging the conventional wisdom that these tools are solely for formal methods and verification. We claim that SMT solvers, when seamlessly integrated into a compiler's static checks, significantly enhance the capabilities of ordinary type checkers in program composition. Specifically, we argue that refinement types, as embodied by Liquid Haskell, enable the use of SMT solvers in mundane programming tasks.

Through a case study on handling binder scopes in compilers, we envision a future where ordinary programming is made simpler and more enjoyable with the aid of refinement types and SMT solvers. As a secondary contribution, we present a prototype implementation of a theory of finite maps for Liquid Haskell's solver, developed to support our case study.

CCS Concepts

• **Software and its engineering** → **Software verification**; **Automated static analysis**; **Formal software verification**; • **Theory of computation** → *Program verification*; *Program analysis*.

Keywords

refinement types, Liquid Haskell, SMT solvers, program design

ACM Reference Format:
Facundo Domínguez and Arnaud Spiwack. 2025. Refinement-Types Driven Development: A study. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

SMT solvers are useful to the ordinary activity of programming. This is what we would like to convince the reader of. More precisely, our claim is that an SMT solver, well-integrated in a compiler, complements an ordinary type checker and can, in fact, be used much in the same way. SMT solvers and type checkers are good at enforcing different kinds of properties, broadening the ways in which we can design our programs.

SMT solvers, when it comes to their application to programming, are usually paired in the literature with terms like "formal methods" or "verification" [1, 3, 4, 11, 19, 24]. We would like to challenge the wisdom that we reach for SMT-solver-based tools when we need formal methods. We would benefit from using SMT solvers in mundane programs. Not because it makes programs more correct, but because it helps us write the programs we want.

We will be arguing, in particular, that refinement types, in the guise of Liquid Haskell [21], let you do just that. Even though Liquid Haskell is also usually invoked together with phrases like "formal methods" or "verification" [9, 10, 16, 20].

Through a case study, we will argue for a future where programming, ordinary programming, is made easier and more pleasant thanks to refinement types and SMT solvers, even though the technology is not ready yet, as we discuss in Section 4. Our case study will be the handling of binders' scopes in compilers. We distill from the experience a set of principles that were useful to us and which could apply to other scenarios with this programming style. A secondary contribution is a prototype implementation of a theory of finite maps for Liquid Haskell's solver, to support our case study, and which we discuss in Section 3.5.

2 Capture-avoiding substitutions

Binding scope management is recognized as a persistent annoyance when writing compilers. It is easy to get wrong and it is a source of mistakes to the point that many have proposed disciplines to prevent mismanagement of scopes. The canonical mistake example is name capture in substitutions like $(\lambda x.y)[y := t]$. The result of this substitution is $\lambda x.t$. Thus $(\lambda x.y)[y := x]$ is $\lambda x.x$. An easy mistake!

Compiler authors have proposed many disciplines to help make scope more manageable. The GHC Haskell compiler, for instance, uses an approach to avoid name capture called *the rapier* [15]. All term-manipulating functions carry an additional *scope* set containing all the variables that appear free in its arguments. This set is used both to decide what to rename a binder to, in order to avoid name capture, and it is also used to skip renaming a binder if it would not capture any free variables. Figure 1 shows an implementation of substitution for the untyped lambda calculus.

2.1 The foil

The rapier was not enough, however, for Maclaurin et al. [12] who report that despite using the rapier they struggled with frequent scope issues in their compiler. They set out to enforce the scope properties of the rapier with Haskell's type system. A stunt that has often been attempted, but Maclaurin et al.'s approach, that they name *the foil*, is probably the first to succeed at enforcing such invariants without incurring an unreasonable amount of boilerplate. In Section 2.3, we will argue that we can achieve similar guarantees more economically with SMT solvers.

Here is our distillation of the properties that Maclaurin et al. set out to guarantee (see also [12, Section 4]):

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
data Exp = Var Int | App Exp Exp | Lam Int Exp

substitute :: Set Int -> Subst Exp -> Exp -> Exp
substitute scope s e0 = case e0 of
  Var i -> lookupSubst s i
  App e0 e1 -> App (substitute scope s e0) (substitute scope s e1)
  Lam i e
    | member i scope,
      let j = freshVar scope ->
        Lam j $ substitute (insert j scope) (extendSubst s i (Var j)) e
    | otherwise ->
        Lam i $ substitute (insert i scope) (extendSubst s i (Var i)) e

freshVar :: Set Int -> Int
freshVar s = case lookupMax s of Nothing -> 0; Just i -> i + 1
```

Figure 1: Rapier style substitution

(1) Every traversed binder must be added to the scope set, otherwise its name could be accidentally used later where a fresh name was intended.
(2) Every traversed binder must be renamed if it is already a member of the scope set, because this name could otherwise be captured as above.
(3) When renaming a binder, the new name must not belong to the scope set.
(4) When renaming a binder, the occurrences of the old bound variable need to be substituted with the new name.
(5) The initial scope set must contain the free variables in the input term and in the range of the substitution to apply.

These properties are exigent, though they do not ensure that we can only write correct substitution functions. For instance, with all these properties it's possible to write a function which takes $(x\ y)[x := x]$ to $(y\ x)$. But as anticipated in the introduction, we are not concerned with full correctness.

Maclaurin et al. propose a library with types `Scope n`, `Name n`, and `NameBinder n l`. A value of type `Scope n` is a set of names, where the type index `n` is the name of the set at the type level. A value of type `Name n` is a name that belongs to the scope set `n`. A value of type `NameBinder n l` is a name `b` such that adding `b` to scope set `n` results in the scope set `l`. These types are to be used in the abstract syntax tree of terms:

```
data Exp n = Var (Name n)
           | App (Exp n) (Exp n)
           | forall l. Lam (NameBinder n l) (Exp l)
```

Then the operations and type checking on the new types will guide the user into respecting the scope requirements when implementing substitution.

```
substitute :: Distinct o => Scope o -> Subst Expr i o -> Expr i -> Expr o
```

This type signature says that no names shadow each other in the scope set `o`. It also says that the substitution will take an expression with free variables in a scope set `i` and produce an expression with free variables in a scope set `o`.

There are mechanisms to check that a scope set is a subset of another, to assert that no name shadows another one in a given scope set, to reason that expressions with free variables in one scope (`Exp n`) can be coerced to expressions with free variables in a superset (`Exp l`), and to introduce scope sets that extend others with freshly created names. They also provide an implementation of maps of variables to expressions, that is the substitutions to apply, with an interface that uses the new types as well. There is for instance the following function to produce fresh variables:

```
withRefreshed
  :: Distinct o
  => Scope o
  -> Name i
  -> (forall (o' :: S). DExt o o' => NameBinder o o' -> r)
  -> r
```

Using the constraint `DExt`, this type signature says that scope set `o'` extends the scope set `o` with the given `NameBinder o o'`. This binder may have the same name as the provided `Name i` if it was not present in `o`, otherwise it will be a fresh name. As another example, the following function always produces a fresh name.

```
withFresh
  :: Distinct n
  => Scope n
  -> (forall l . DExt n l => NameBinder n l -> r )
  -> r
```

With ingenious engineering and design, the foil meets its rather ambitious goal. But it is unfortunate that the authors needed to be ingenious. All things equal, we prefer program components to be straightforward. Because ingenious solutions take time, and because straightforward solutions are easier to adapt when the parameters of the problem evolve.

2.2 A Liquid Haskell primer

We will turn next to Liquid Haskell as our proposed solution, but first let us introduce Liquid Haskell briefly. Liquid Haskell is a plugin for Haskell which statically checks that programs respect signatures provided by the programmer. There are two key differences between Liquid Haskell signature checking and a classical type checker:

- The checking process consists in generating logical constraints or proof obligations which are then fed to an SMT solver, leveraging the powerful capabilities of SMT solvers to reason about numbers, arrays, strings, and other sorts.
- Signatures are expressed with *refinement types* of the form `{x:b | p}`, which denote values of base type `b` that satisfy predicate `p`. We will write sometimes `b<p>` to denote `{x:b | p x}`. Refinements are subject to subtyping in the same way as subsets in set theory, so that we have

```
{-@ f :: {x:Int | x > 1} -> {x:Int | x > 0} @-}
f :: Int -> Int
f x = x
```

Liquid Haskell reads refinement type signatures and other annotations from inside special Haskell comments `{-@ … @-}`. We will skip them in our snippets when it is unambiguous.

The predicates in the refinement types are in a language of expressions referred to as the logic language. For the sake of this paper, we can regard it as a subset of Haskell, except that predicates are assembled both from regular Haskell functions and functions that are only available in the logic language.

We will use sparingly the following form of refinement type signature.

```
{-@ idInt :: forall <p :: Int -> Bool>. Int<p> -> Int<p> @-}
idInt :: Int -> Int
idInt x = x
```

We say that `p` is an abstract predicate, and it is inferred by Liquid Haskell depending on the context in which `idInt` is used.

A function like `member`, which comes from the module `Data.Set` in the `containers` package, is linked by Liquid Haskell to the SMT solver's theory of sets.

```
import Data.Set
assume member :: Ord a
              => x:a -> xs:(Set a) -> {v:Bool | v <=> Set_mem x xs}
```

Refinement type signatures starting with the `assume` keyword declare that the corresponding Haskell function honors the signature, but it is not checked. In this case, it is because `Data.Set` is an external dependency that Liquid Haskell can not check. But it can also be applied to our own functions.

Here `Set_mem` is a symbol that Liquid Haskell maps to the theory of sets in the SMT solver. While Liquid Haskell does not check that `member` behaves as declared in the refinement type signature, it will assume the property in the return refinement type whenever `member` is used in a program.

Notice how the predicate on the return type mentions both arguments. Liquid Haskell lets us express refinement types which relate arguments with each other, and with the result in this manner. This obviates the need to give a type-level name to arguments using existential quantification.

To define a function only available to use in Liquid Haskell annotations, we can use the `measure` keyword, such as:

```
measure listElts :: [a] -> Set a
  listElts []     = {v | (Set_emp v)}
  listElts (x:xs) = {v | v = Set_cup (Set_sng x) (listElts xs) }
```

Here `Set_cup` and `Set_sng` are predefined functions to express the union of sets and the singleton set respectively.

It is also possible to define uninterpreted symbols by simply omitting the definition. It would look like this

```
measure listElts :: [a] -> Set a
```

The meaning of the function would then be given by `assume` refinement type signatures on other functions. See for instance the use of the `domain` function in the following section.

2.3 The rapier, refined

We argue, next, that using Liquid Haskell to enforce the requirements from Section 2.1 is more straightforward than using the type checker alone. The code presented in this section is available in the file `Subst1.hs`[1].

We define a function `freeVars` in the same module as `substitute`, which collects the free variables of an expression. We note that this function is only used in refinement type signatures, and in particular, it is not evaluated when calling to `substitute`.

```
freeVars :: Exp -> Set Int
freeVars e = case e of
  Var i -> singleton i
  App e1 e2 -> union (freeVars e1) (freeVars e2)
  Lam i e -> difference (freeVars e) (singleton i)
```

Next, we need to give the following refined signature to the `freshVar` of Figure 1:

```
{-@ assume freshVar :: s:Set Int -> {v:Int | not (member v s)} @-}
```

[1] https://github.com/tweag/ifl2025-liquidhaskell/blob/main/src/examples/Subst1.hs

This signature is assumed rather than checked. We could choose to check it, but Liquid Haskell does not have a good built-in understanding of the `lookupMax` function that we use. So instead, we choose to assume the signature. This is our first principle of programming with refinement types:

Principle 1. *Typically, refinement types allow you to reduce the trusted code base, but they also offer you a choice. When it is easier to prove a result by hand than with the SMT solver, you can assume the property and justify it informally.*

In this article, by *trusted code base*, we mean the portion of a codebase where the programmer must prove the desired properties herself rather than relying on static checks to enforce said properties. Tooling like compilers, type checkers, SMT solvers, and operative systems are excluded from this definition.

It is good discipline to justify systematically why assumptions should hold. An incorrect assumption could make Liquid Haskell accept programs that do not meet the properties we mean to check. The consequences range through the whole gamut from incorrect results, to security vulnerabilities and crashes, depending on the kind of checks.

Finally, we will take as a parameter a datatype representing substitutions (*i.e.* finite maps of variables to terms). To represent this parameter in our study we take an abstract type and `assume` the necessary properties that a substitution type needs to respect. Since this is ordinary programming, not a verification project, we need to test our code, and we provide a concrete type for that sake. But using an abstract type ensures that we can support any efficient substitution type.

```
data Subst t -- opaque
{-@ measure domain :: Subst e -> Set Int @-}

assume lookupSubst
  :: forall <p :: Exp -> Bool>.
     s:Subst Exp<p>
  -> {k:Int | member k (domain s)}
  -> Exp<p>

assume extendSubst
  :: s:Subst a
  -> i:Int
  -> a
  -> {v:Subst a | union (domain s) (singleton i) = domain v }
```

Notice that the logical function `domain`, which stands for the set of variables that the substitution defines, is uninterpreted. It must be since it is an assumption.

That's it, this is the entirety of our trusted code base for this example. For the most part, it required thinking about what properties we wanted to enforce, but not much about how they ought to be enforced.

In order to deal with scope checks, we define a type alias `ScopeExp S`, that is the type of all expressions whose free variables are in the set `S`[2].

```
{-@ type ScopedExp S = {e:Exp | isSubsetOf (freeVars e) S} @-}
```

Functions like `isSubsetOf` and `difference` come from the `Data.Set` module. We can give now the following signature to `substitute`

[2] In type aliases, Liquid Haskell expects parameter names corresponding to terms (i.e. not types) to start with an uppercase letter.

```
{-@
substitute
  :: scope:Set Int
  -> s:Subst (ScopedExp scope)
  -> ScopedExp (domain s)
  -> ScopedExp scope
@-}
substitute :: Set Int -> Subst Exp -> Exp -> Exp
```

Remarkably, this implementation for `substitute`, where we check static scopes, is unchanged from the implementation of Figure 1. This will not always be the case, but this exemplifies how using Liquid Haskell to enforce invariants tends to create less boilerplate than a type-based approach.

The refinement type signature of `substitute` is a direct translation of the Haskell type signature used by the foil.

```
substitute :: Distinct o => Scope o -> Subst Expr i o -> Expr i -> Expr o
```

The foil's `Scope o` type becomes a regular set `scope:Set Int` of names, there's no need for the type parameter `o`, which the foil uses as a type-level name for the scope, since we can directly refer to `scope` in terms. The foil's `Subst Expr i o` type becomes `s:Subst (ScopedExp scope)`, the parameter `i` is omitted and referred to as `domain s` instead. The foil's `Expr i` type becomes `ScopedExp (domain s)`, which still requires the free variables of the input expression to be in the domain of the substitution. And finally, both return types `Expr o` and `ScopedExp scope` require the free variables of the output to be in the given scope set.

Figure 1 uses that a substitution `s :: Subst (ScopedExp scope)` also has (refined) type `s :: Subst (ScopedExp (insert i scope))`, as there are recursive calls like

```
substitute (insert i scope) (extendSubst s i (Var i))
```

which requires

```
extendSubst s i (Var i) :: Subst (ScopedExp (insert i scope))
```

which in turn requires

```
s :: Subst (ScopedExp (insert i scope))
```

This kind of subtyping is trivial with refinement types. It is the default behavior. Whereas with an ML type system, subtyping is not a typical feature. The foil, for instance, needs an explicit function to cast substitutions when extending a scope. This is our next principle:

Principle 2. *Refinement types add a layer of subtyping on top of your type system. When your program is best modeled with subtyping you should consider refinement types.*

The type of lambda terms is also unchanged, as the well-scoping invariant is applied to a whole term at once. A nice consequence of it is that functions that do not benefit from all the scope checking business can simply take a naked term and ignore it. The `freeVars` function, for example, is implemented on naked terms.

2.4 A hybrid approach

Our refinement type signature of `substitute` follows the type signature of Maclaurin et al. to the letter. Yet we can introduce the following bug in `substitute` from Figure 1, where we omit the fresh binder `j`:

```
  ...
  Lam i e
    | member i scope ->
      Lam i $ substitute (insert i scope) (extendSubst s i (Var i)) e
    | otherwise -> ...
```

Liquid Haskell flags no errors but the program will still misbehave as follows (in pseudo-Haskell).

$$\texttt{substitute}\,\{x\}\,(\lambda x.y)[y := x] = (\lambda x.x)$$

What is going on? The binder `i` is now capturing free variables in the range of the substitution. The signature is, in fact, indifferent to whether the binder `i` is already present or not in the scope set. There is no mechanism to prevent adding a binder that is already present in the scope set. That is, we fail to enforce Property (2) from Section 2.1. And, more to the point, how could we? "Never add a binder to the scope set that is already present" is not a set theoretical property. It is not even a functional property. It is a kind of temporal invariant.

Such temporal invariants are not naturally expressed in the logic of Liquid Haskell. But they are quite easy to implement with abstract types. So let us use an abstract type. What we need to do is to ensure that whenever we see a new binder it must be tested against the scope, and that this test is packaged together with fresh name generation.

We follow the foil and introduce an abstract type `Scope` and a function `withRefreshed`. The types are a little simpler because we do not need existential quantification to reflect value-level objects at the type level, but otherwise these are the same functions and types as in Section 2.1.

```
newtype Scope = UnsafeScope { unsafeUnScope :: (Set Int) }
{-@
predicate Member E S = Set.member E (unsafeUnScope S)

withRefreshed :: s:Scope -> i:Int
  -> {p:(Scope, Int) |
       not (Member (snd p) s) && fst p == union s (singleton (snd p))}
@-}
withRefreshed :: Scope -> Int -> (Scope, Int)
withRefreshed (UnsafeScope s) i
  | Set.member i s = let j = freshVar s in (UnsafeScope (insert j s), j)
  | otherwise = (UnsafeScope (insert i s), i)
```

We needed to add a refinement type signature to `withRefreshed` to serve as glue with the Liquid Haskell world. This refinement type signature tells Liquid Haskell precisely that `withRefreshed` does both membership checking and fresh variable call: the variable returned by `withRefreshed` is not in the old scope but is in the new scope.

We make the type `Scope` abstract to enforce that binders are always refreshed when traversed, as `withRefreshed` is the only way to test for membership and to extend a scope. This is why we define a `Member` predicate alias, only available in the logic, but provide no `member` function in Haskell for `Scopes`. The full code for this example can be found in the file `Subst2.hs`[3].

This is our next principle for refinement types:

Principle 3. *Refinement types and abstract types are best at enforcing different kind of properties. You should use the simpler solution for each property that you need, as refinement types and abstract types mix well.*

[3] https://github.com/tweag/ifl2025-liquidhaskell/blob/main/src/examples/Subst2.hs

3 Unification

Now that we have established the refined rapier interface, let us show how it can be applied to a more realistic example: solving first-order equational formulas. Specifically, we will be solving a form of Horn clauses in the Herbrand domain. This is the sort of unification problem which can show up when type-checking programs with GADTs [17]. Scope management in such a solver is a much trickier business than in the case of mere substitutions and, in the authors' experience, something where any help from the compiler is welcome. The source code of this section can be found in the file `Unif.hs`[4].

In addition to variables, still represented as integers, we have unification variables. Unification variables have their own scopes: the formula $\exists x.\forall y.x = y$ does not have a solution. It will be reduced to a formula of the form $f_x = y$ where f_x is a unification variable; we very much don't want this unification problem to succeed: we shall make it so that y is not in the permissible scope for f_x.

Furthermore, the unification algorithm will perform substitutions. Substitutions are blocked by unification variables as we do not know what they stand for yet. So a unification variable, in our syntax, is a pair $(f, [x_0 := t_0, \ldots, x_n := t_n])$ of a unification variable proper and a suspended substitution. Where $\{x_0, \ldots, x_n\}$ is the scope of f. Such a pair is akin to a skolem function application $f(t_0, \ldots, t_n)$. Notice in particular, how the solution of f can only have free variables in $\{x_0, \ldots, x_n\}$, but $(f, [x_0 := t_0, \ldots, x_n := t_n])$ may live in a different scope altogether. This type of unification problem is tricky because there are multiple intermingled scopes to manage, rather than one like in the case of substitution (Section 2).

```
type Var = Int
type SkolemApp = (Var, Subst Term)
```

This way, our formula $\exists x.\forall y.x = y$ will be reduced to $(f_x, []) = y$ which does not have a solution. On the other hand $\forall x.\exists y.x = y$ becomes $x = (f_y, [x := x])$ so x is a solution for f_y and the formula is solvable.

Our unification algorithm is a first-order variant of pattern unification [13] sufficient to eliminate equalities to the left of implication in the style proposed by Miller and Viel [14]. The main functions, sans refined signatures, can be found in Figure 2. Unification algorithms can get pretty finicky, for the sake of simplicity our algorithm is not as complete as it could be and will miss some solutions[5].

At the heart of the algorithm is substitution inversion [23]: when encountering an equality of the form

$$(f_x, [y := a, z := b]) = u$$

If there is a solution, we want it to be

$$f_x := u[a := y, b := z]$$

This is the same as pattern unification, except that it does not need terms to contain functions. The `inverseSubst` function is responsible for this inversion.

[4] https://github.com/tweag/ifl2025-liquidhaskell/blob/main/src/examples/Unif.hs

[5] We have, on the other hand, tried to make the algorithm correct, so if it finds unsound solution it is a bug and we apologize.

We are choosing a language of term with both regular variables (representing variables bound by universal quantifiers), skolem applications representing unification variables with their substitutions, and sufficient constructors to encode arbitrary terms. Here is the concrete type of term, as well as that of formula where the only thing to remark is that the left-hand side of implications is a single equality.

```
data Term
  = V Var | SA SkolemApp | U | L Term | P Term Term

data Formula
  = Eq Term Term               -- equality
  | Conj Formula Formula       -- conjunctions
  | Then (Term, Term) Formula  -- a = b => f
  | Exists Var Formula         -- existential quantification
  | Forall Var Formula         -- universal quantification
```

In Figure 2, the function `unify` takes a rapier scope parameter containing all the variables that can appear free in the input formula. This set is used to rename `Forall` binders when doing substitutions. For instance, unifying the following formula

$$\forall x.\forall y.\exists z.y = L(x) \Rightarrow \forall x.y = z$$

reduces to unifying

$$\forall x.\forall y.\exists z.(\forall x.y = z)[y := L(x)]$$

and the substitution needs to rename the inner binder x.

In a preceding pass (Section 3.1), existential quantifiers are replaced with skolem applications, so in `unify` we assume that there is no existential quantifier. We have functions `substituteFormula` and `substitute` to apply substitutions in formulas and terms respectively, and `substituteSkolems` to substitute unification variables in formulas. We have a function `skolemSet` to collect the skolem applications of a term. And a function `fromListSubst` to construct a substitution from a list of pairs `[(Var, Term)]`.

The functions `substEq` and `unifyEq` are simplified here for the sake of presentation. They handle more cases in the reference source code, but these cases are not essential to our discussion.

The function `unifyEq` defines what a good solution should be. One of the conditions is that whatever term `t'` is proposed as solution for a skolem `i`, it needs to have as free variables only those in the domain of the substitution defining the skolem application (*scope check*). For instance, in $(f_x, [x := y]) = P(y, y)$, $P(x, x)$ is a solution that satisfies the scope check, but $P(x, y)$ would be a solution that doesn't since y is not in the domain of $[x := y]$.

Another condition is that the skolem `i` should not occur in the solution `t'` (*occurs check*). For instance, in the previous example $f_x := P(x, f_x)$ is a solution that doesn't pass the occurs check. In addition, since we are inverting a substitution to find `t'`, we might not find solutions if we cannot invert the substitution. This implementation only inverts substitutions where variables are mapped to variables. That is, we solve $(f, [z := x]) = L(L(x))$ to get the solution $f := L(L(z))$ but we do not try solving, say, $(f, [z := L(x)]) = L(L(x))$.

```
unify :: Set Int -> Formula -> Maybe [(Var, Term)]
unify s (Forall v f) = unify (Set.insert v s) f
unify s (Exists v f) = error "unify: the formula has not been skolemized"
unify s (Conj f1 f2) = do
    unifyF1 <- unify s f1
    unifyF2 <- unify s (substituteSkolems f2 unifyF1)
    return (unifyF1 ++ unifyF2)
unify s f@(Then (t0, t1) f2) =
    let subst = fromListSubst (substEq t0 t1)
     in unify s (substituteFormula s subst f2)
unify s (Eq t0 t1) = unifyEq t0 t1

substEq :: Term -> Term -> [(Var, Term)]
substEq (V i) t1 = [(i, t1)]
substEq t0 (V i) = [(i, t0)]
substEq _ _ = []

unifyEq :: Term -> Term -> Maybe [(Var, Term)]
unifyEq t0 t1@(SA (i, s))
  | Just s' <- inverseSubst $ narrowForInvertibility (freeVars t0) s
  , let t' = substitute s' t0
  , not (Set.member i (skolemSet t'))
  , Set.isSubsetOf (freeVars t') (domain s)
  = Just [(i, t')]
unifyEq t0@(SA _) t1 = unifyEq t1 t0
unifyEq _ _ = Nothing

-- | @narrowForInvertibility vs s@ removes pairs from @s@ if the
-- range is not a variable, or if the range is not a member of @vs@.
narrowForInvertibility :: Set Var -> Subst Term -> Subst Term
narrowForInvertibility vs (Subst xs) =
  Subst [(i, V j) | (i, V j) <- xs, Set.member j vs]

inverseSubst :: Subst Term -> Maybe (Subst Term)
inverseSubst (Subst xs) = fmap Subst (go xs)
  where
    go [] = Just []
    go ((i, V j) : xs) = fmap ((j, V i) :) (go xs)
    go _ = Nothing
```

Figure 2: Conditional unification

3.1 A look at skolemization

Figure 3 shows the function to replace existential quantifiers with unification variables. This example is interesting because the complexity of managing the scopes for both universal and existential quantifiers considerably exceeds the canonical example of the rapier.

The `skolemize` function takes a set `sf` as an argument as well as a finite map `m` as the state of a state monad. The set `sf` is the scope set of variables that have been introduced with universal quantification, and can appear free in the input formula. The finite map `m` contains the variables that have been introduced with existential quantification together with their own scopes, that is, the universally quantified variables in scope at the original existential binder.

We pass the map `m` as a monadic state, because we do not want to generate the same unification variable for existential binders appearing on different subformulas, since unification variables scope over the entire formula. For instance, the following formula

$$\forall x.\exists y.x = y \wedge \forall z.\exists y.z = y$$

should produce unification variables like

$$\forall x.x = y[x := x] \wedge \forall z.z = w[x := x, z := z]$$

```
skolemize :: Set Int -> Formula -> State (IntMap (Set Int)) Formula
skolemize sf (Forall v f) = do
    m <- get
    put (IntMap.insert v sf m)
    f' <- skolemize (Set.insert v sf) f
    pure (Forall v f')
skolemize sf (Exists v f) = do
    m <- get
    let u = if IntMap.member v m then
              freshVar (Set.fromList (IntMap.keys m))
            else
              v
        m' = IntMap.insert u sf m
    put m'
    let subst = fromListSubst [(v, SA (u, fromSetIdSubst sf))]
    skolemize sf (substituteFormula sf m' subst f)
skolemize sf (Conj f1 f2) = do
     f1' <- skolemize sf f1
     f2' <- skolemize sf f2
     pure (Conj f1' f2')
skolemize sf f@(Then (t0, t1) f2) = do
     f2' <- skolemize sf f2
     pure (Then (t0, t1) f2')
skolemize _ f@Eq{} = pure f
```

Figure 3: Skolemization

It would be a mistake to call both unification variables y and w the same. Their occurrences even have different scopes!

We expect the set `sf` to be a subset of the keys in `m`. This is to reflect the fact that, for debugging purposes, we do not want unification variables to be called the same as universally quantified variables. It is not a strict requirement, but one that makes the output of `skolemize` considerably easier to read.

Yet, we do need to keep the scope set `sf` separate from the monadic state because it is needed to construct the skolem function applications where existential variables are found.

Here is the refinement type signature of `skolemize`.

```
type ScopedFormula S = {f:Formula | isSubsetOf (freeVarsFormula f) S}

assume skolemize
  :: sf:Set Int
  -> f:ScopedFormula sf
  -> State
       <{\m0 ->
           isSubsetOf sf (IntMapSetInt_keys m0)
         && consistentScopes m0 f
        }

       , {\m0 v m ->
            consistentScopes m v
          && existsCount v = 0
          && isSubsetOf (freeVarsFormula v) sf
          && intMapIsSubsetOf m0 m
         }>
       (IntMap (Set Int)) Formula
```

This type signature is, admittedly, a bit involved. However while we were designing this case study, `skolemize` stayed without a refined signature until pretty much the very end. This is possible because the inherent subtyping of refinement types makes it easy to use unrefined and refined functions together. Of course this prevented us from having guarantees for the program end-to-end, but it is fine to add guarantees only where you need them. What you

choose to harden will not have to infect the rest of the program. Which leads us to our next principle

Principle 4. *Functions with refined signature and without mix well. You should first use refinement types on function with the best power-to-weight ratio. You can incrementally add stronger types on more functions as your program evolves.*

Liquid Haskell helpfully lets us treat the state monad as equipped with a Hoare logic `State<pre,post>`. The supporting code for the refined state monad is not readily available in Liquid Haskell. It probably should be, but in the meantime, it can be found in Liquid Haskell's test suite, so we simply copied it in the file `State.hs`[6].

The main conjuncts of the postconditon are `consistentScopes m v` and `existsCount v = 0`, the rest are invariants used by the recursive calls of `skolemize`.

- `existsCount v = 0` means that `skolemize` returns a formula without existential quantifiers. As it is a requirement of `unify`.
- `consistentScopes m v` means that `skolemize` returns a formula F such that all the occurrences of any unification variable i in F have an attached substitution whose domain is the scope of i as reported by `m`. This is our main scope invariant for this section.

While it is possible to define `skolemize` with a set of unification variables in the state instead of a finite map, the map choice makes easier to express the consistency of the unification scopes. Changing the functions to make them easier to explain is a topic which we will find again later on.

This signature for `skolemize` cannot be checked with Liquid Haskell today due to a bug, so we ended up assuming the refinement type signature in keeping with Principle 1. The rest of the code does not benefit less because of it.

3.2 The theory of `unifyEq`

Let us now turn to the `unifyEq` function, which is a traditional unification function: it takes an equation and returns definitions for its unification variables. The refined signature that we give to `unifyEq` statically enfoces scope checks, occurs checks, and the consistency of scopes in the result and in the arguments.

```
type ConsistentScopedTerm S M =
  {t:Term | isSubsetOf (freeVars t) S && consistentScopesTerm M t}

unifyEq
  :: s:Set Int
  -> m:IntMap (Set Int)
  -> t0:ConsistentScopedTerm s m
  -> t1:ConsistentScopedTerm s m
  -> Maybe
      [( v :: Var
       , {t:Term |
             consistentScopesTerm m t}
         && isSubsetOfJust (freeVars t) (IntMap.lookup v m)
         && not (Set.member v (skolemSet t))
        }]
```

The predicate `consistentScopesTerm m t` is only used in refinement types, and checks that the domains of the unification variables' substitutions in a term `t` are the scopes given by `m`.

[6] https://github.com/tweag/ifl2025-liquidhaskell/blob/main/src/examples/State.hs

```
consistentScopesTerm :: IntMap (Set Int) -> Term -> Bool
consistentScopesTerm m (V _) = True
consistentScopesTerm m (SA (i, s)) =
       IntMap.lookup i m == Just (domain s)
    && consistentScopesSubst m s
consistentScopesTerm m U = True
consistentScopesTerm m (L t) = consistentScopesTerm m t
consistentScopesTerm m (P t0 t1) =
    consistentScopesTerm m t0 && consistentScopesTerm m t1

consistentScopesSubst :: IntMap (Set Int) -> Subst Term -> Bool
consistentScopesSubst m (Subst xs) =
    all (\(_, t) -> consistentScopesTerm m t) xs
```

We would like to draw the reader's attention to the parameters of `s` and `m` in the refinement type signature of the `unifyEq` function, conspicuously absent in the implementation of Figure 2. This is because, in the source code, we have extended the implementation of `unifyEq` and many other functions with these parameters. We could reconstruct these scope assumptions in the functions' preconditions, but it is more involved, and requires a great deal more lemmas to convince the SMT solver.

Principle 5. *It is easier to express properties and to use an SMT solver when assumptions are explicit rather than reconstructing assumptions that are implicit. Do not hesitate to pass assumptions as arguments to functions, even if those arguments are not used by the function.*

Note that compilers typically remove such obviously unused arguments during compilation. GHC certainly does. So there is essentially no computational cost to these extra arguments anyway.

3.3 Totality and `unify`

There is not much more to add for the `unify` function, but let us take this opportunity to talk about the totality requirement. Here is its signature.

```
unify
  :: s:Set Int
  -> m:IntMap (Set Int)
  -> {f:ConsistentScopedFormula s m | existsCount f = 0}
  -> Maybe
       [( v :: Var
        , { t:Term |
              consistentScopesTerm m t
           && isSubsetOfJust (freeVars t) (IntMap.lookup v m)
           && not (Set.member v (skolemSet t))
          }
        )] / [formulaSize f]
```

Notice the precondition `existsCount = 0`. It is not optional. Indeed, the `Exists` case of `unify` in Figure 2 raises an error. Liquid Haskell, however, requires functions to be total. We need this precondition so that Liquid Haskell can prove that this case never occurs.

This totality requirement is not necessary to refinement types in general. However, in the case of Haskell, laziness lets us write

```
{-@ bad :: () -> { false } @-}
bad :: () -> ()
bad _ = let {-@ f :: { false } @-}
            f = error "never happens"
        in (\_ -> ()) f
```

It may seem that Liquid Haskell could accept this function because `f` appears to prove `false`. In a strict language this would not be a

big problem as `bad` would loop and any attempt at using `bad` would diverge. But `bad` is actually a total function. Liquid Haskell rejects `bad` because it fails to prove that `f` is total, hence refuses to accept its signature.

This is also why the signature of `unify` ends with `/ [formulaSize f]`. Liquid Haskell needs to prove that `unify` terminates and, because of the substitutions, `unify` is not a structurally recursive function. So Liquid Haskell needs a little help in the form of a termination metric. We use here the number of connectives in the argument formula, which is unaffected by substitution since we only substitute inside terms.

3.4 Lemmas in Liquid Haskell

In the previous sections we have seen that the refined implementation can be different from the classical version by adding computationally irrelevant arguments. Another way in which they could differ is with the addition of lemmas.

Take, for instance, the `unifyFormula` function which ties together `skolemize` and `unify`, it differs from its classical implementation as follows:

```
 unifyFormula :: Set Int -> IntMap (Set Int) -> Formula -> Maybe [(Var, Term)]
 unifyFormula s m f =
     let m' = addSToM s m
-        skf = skolemize s f
+        skf = skolemize s f ? lemmaConsistentSuperset m m' f
         (f'', m'') = runState skf m'
      in unify s m'' f''
```

This idiom `e?p` means "use lemma `p` when checking `e`". Lemmas are not used automatically, this is how Liquid Haskell is instructed to use them with parameter values supplied by the user.

Lemmas, in Liquid Haskell, are ordinary functions. Proofs by inductions arise from ordinary (total!) recursion. In the case of `lemmaConsistentSuperset` the proof is entirely straightforward

```
{-@
lemmaConsistentSuperset
  :: m0:IntMap (SetInt)
  -> {m1:IntMap (Set Int) | intMapIsSubsetOf m0 m1}
  -> {f:Formula | consistentScopes m0 f}
  -> {consistentScopes m1 f}
@-}
lemmaConsistentSuperset
  :: IntMap (Set Int) -> IntMap (Set Int) -> Formula -> ()
lemmaConsistentSuperset m0 m1 (Forall _ f) =
    lemmaConsistentSuperset m0 m1 f
lemmaConsistentSuperset m0 m1 (Exists _ f) =
    lemmaConsistentSuperset m0 m1 f
lemmaConsistentSuperset m0 m1 (Conj f1 f2) =
      lemmaConsistentSuperset m0 m1 f1
    ? lemmaConsistentSuperset m0 m1 f2
lemmaConsistentSuperset m0 m1 (Then (t0, t1) f2) =
      lemmaConsistentSupersetTerm m0 m1 t0
    ? lemmaConsistentSupersetTerm m0 m1 t1
    ? lemmaConsistentSuperset m0 m1 f2
lemmaConsistentSuperset m0 m1 (Eq t0 t1) =
      lemmaConsistentSupersetTerm m0 m1 t0
    ? lemmaConsistentSupersetTerm m0 m1 t1
```

So straightforward, in fact that the proof was largely written by AI-based code completion. Since lemmas do not have computational content (`{p}` is a shorthand for `{_:() | p }`), we only care about the existence of a proof, making code completion particularly useful. Liquid Haskell understanding the theory of finite maps (see Section 3.5) is crucial in making this proof so terse.

The lemma `lemmaConsistentSuperset` uses an analogous lemma `lemmaConsistentSupersetTerm` for terms, whose proof ultimately depends on the following lemma which we must assume of the substitution data type. Unsurprisingly, the substitution interface needs to satisfy more properties than in Section 2.3 to accommodate unification variable scopes.

```
assume lemmaConsistentSupersetSubst
  :: m0:_
  -> {m1:_ | intMapIsSubsetOf m0 m1}
  -> {s:_ | consistentScopesSubst m0 s}
  -> {consistentScopesSubst m1 s}
```

3.5 Extending Liquid Haskell to support `IntMap`

Our unification case study uses the theory of finite maps. Liquid Haskell, however does not support a theory of finite maps[7]. It is possible to do without it. In a first approximation we did much of this study in vanilla Liquid Haskell. But we lost out on automation: we got more lemmas to prove and pass around. Properties like the scope check, or the lemma `lemmaConsistentSuperset`, involved operations on finite maps and were more convoluted.

To support this study, we implemented the theory of finite maps for Liquid Haskell. It is not ready to integrate in future release yet, for one thing: we only support finite maps with `Int` as their domain and `Set Int` as their codomain. It could easily be adapted for any fixed domain and codomain types, but it is not yet a general solution that can be instantiated at any domain or codomain type. But our ultimate intent is to upstream these changes. Our modifications can be found in the file `ifl25-liquidhaskell.patch`[8] and the file `ifl25-liquid-fixpoint.patch`[9].

The theory of finite maps is a good example of a theory that Liquid Haskell wants to support: it is both powerful, and widely applicable. Pragmatically, it is also one that is reasonably easy to support with SMT solvers by translating it to the theory of arrays.

On the syntax front, Liquid Haskell allows to link a Haskell type with a particular representation in the SMT solver.

```
{-@ embed IntMap * as IntMapSetInt_t @-}
```

Here we are indicating that `IntMap b` must be represented as `IntMapSetInt_t` in the logic. `IntMapSetInt_t` is an alias for `Array Int (Option (Set Int))`. An array is an entity that associates keys with values, and which has an equality predicate, and it is defined as one of the theories in SMT-LIB, the standard interface to SMT solvers [2]. The keys in this case are integers, and the values are either `None` if the key is not in the map, or `Some s` if the key maps to a set `s`. The `Option` type is a copy of Haskell's `Maybe`. We do not reuse `Maybe` as Liquid Haskell's framework to connect to the SMT solver is reused for other languages (*e.g.* [8]), and we prefer to keep the implementation free of language specific details. Here is the declaration of the `Option` data type in SMT-LIB.

```
(declare-datatype Option (par (a) (None (Some (someVal a)))))
```

[7] Issue to support maps in the Liquid Haskell repository: https://github.com/ucsd-progsys/liquidhaskell/issues/2534
[8] https://github.com/tweag/ifl2025-liquidhaskell/blob/main/src/patches/ifl25-liquidhaskell.patch
[9] https://github.com/tweag/ifl2025-liquidhaskell/blob/main/src/patches/ifl25-liquid-fixpoint.patch

We arranged for Liquid Haskell to include this declaration in the preamble of any queries to the SMT solver. The types `Array`, `Int`, and `Set` are already known to the tooling. It does not matter what type `b` is instantiated to, the `embed` annotation will always set the same representation for `IntMap b`, and this is a limitation that would need to be addressed to support maps properly.

The array theory allows to describe how to retrieve the value associated with a key, and how to update the value. On the Haskell front, we link these operations to those of the `IntMap b` type.

```
define IntMap.empty = (IntMapSetInt_default None)
define IntMap.insert x y m = IntMapSetInt_store m x (Some y)
define IntMap.lookup x m =
  if (isSome (IntMapSetInt_select m x)) then
    (GHC.Internal.Maybe.Just (someVal (IntMapSetInt_select m x)))
  else
    GHC.Internal.Maybe.Nothing
```

The operations `IntMapSetInt_default`, `IntMapSetInt_store`, and `IntMapSetInt_select` are aliases that we implemented in Liquid Haskell to call to the array operations. In the case of `lookup`, we translate the `Option` type to Haskell's `Maybe`.

The implementation of union, intersection, difference, and subset checks for maps, however, need operations beyond the standard interface, and not all SMT solvers can support them. In our implementation we used the `map` operation of the Z3 SMT solver. The following snippet contains the implementation of `intMapIsSubsetOf` in SMT-LIB, and we also feed these declarations to the SMT solver in a preamble to the queries.

```
; Similar to do {a0 <- oa0; a1 <- oa1; guard (a0 /= a1); pure a0}
(define-fun difference_strict_p2p
  ((oa0 (Option (Set Int)))
   (oa1 (Option (Set Int))))
  (Option (Set Int))
  (match oa0
    ((None None)
     ((Some a0) (match oa1
                  ((None oa0)
                   ((Some a1) (ite (= a0 a1) None oa0)))))))))

; Similar to: empty == zipWith difference_strict_p2p xs ys
; where zipWith applies the function pointwise to the values in the
; arrays
(define-fun IntMapSetInt_isSubsetOf
  ((xs (Array Int (Option (Set Int))))
   (ys (Array Int (Option (Set Int)))))
  Bool
  (= ((as const (Array Int (Option (Set Int)))) None)
     ((_ map IntMapSetInt_difference_strict_p2p) xs ys)))
```

Besides the limitation of the `embed` annotation, another barrier for proper support is that old versions of SMT-LIB require user defined functions to have monomorphic types. This means, for instance, that the type of `IntMapSetInt_isSubsetOf` cannot be generalized to work on any `IntMap`.

While newer versions of the standard allow for polymorphic types, these still need to be implemented by SMT solvers. Until the implementations catch up with the standard, feeding operations with monomorphic types will require Liquid Haskell to be smart about generating these operations with the appropriate types, instead of putting them in a preamble once and for all queries.

4 Evaluation

The substitution case study of Section 2 allows for a direct comparison between type methods and refinement type methods. We can see that the trusted code base of the Liquid Haskell version of Section 2.3 is quite small compared to that of the foil [12] (reviewed in Section 2.1). This is in large part because refinement types can enforce invariants without the need for abstract types, and such an open interface can be extended by the user. Contrast with the abstract-type approach where you have to design, upfront, a set of invariant-preserving operations sufficient to express downstream programs. None of these functions will benefit from the abstract types invariant, hence will be part of the trusted code base. Even when we mix refinement and abstract types as in Section 2.4, we do not have quite as large a trusted code base to consider.

This is not to mean that refinement types are superior to type abstractions. They are best at enforcing different types of invariants, as discussed in Section 2.4.

When the invariants of a program naturally involve mathematical objects such as arithmetic or sets, refinement types are likely to be more approachable, requiring less careful a design than coming up with an encoding inside and ML-like type system. Proposing refinement type signatures requires determining appropriate invariants for a task, which is a requisite for any static checking approach. But it doesn't impose the burden of encoding the invariants with lower-level constructs. On the other hand, when a program needs a theory that Liquid Haskell, say, does not have support for, it may not be that clear and the program author may need to mobilize comparable effort for refinement types as she would have for an abstract-type encoding.

Error reporting. A type-checker approach, however, is likely to produce error messages that are easier both to understand and to fix, provided that the user goal is feasible. The user is guided into correcting the errors by the types and the operations of the supporting library. With SMT solvers, there is always the question of whether a goal is provable or not in the theories at hand. Is there some additional lemma that is necessary about the user defined functions? The user has to figure it out on her own. How are the assumptions insufficient to prove the goal? The user has to compute it on her own too, although it is plausible that counterexamples or better location information [22] can be offered when the tooling matures.

But there are informative error messages too. Let us consider the lemma `lemmaConsistentScopesSubst` discussed in Section 3.2. If we drop this lemma from the definition of `unifyEq`, we get the following error message (heavily edited for presentation):

```
publications/ifl25-rtdd/examples/Unif2.hs:580:18: error:
    Liquid Type Mismatch
    The inferred type
      ss' : {ss' : Subst {v : Term | consistentScopesTerm m v} |
                 Set_com Set.empty == domain ss'}
    is not a subtype of the required type
      VV : {VV : Subst Term | consistentScopesSubst m VV}
    in the context
      ?g : {?g : Maybe (Subst Term) |
               ?g == Just ss'
            && ?g == inverseSubst s m
                       (narrowForInvertibility (freeVars t1) ss)}
```

```
      t0 : {t0 : (Int, (Subst Term)) | t0 == SA (i, ss)
               isSubsetOf (freeVars t0) s
            && consistentScopesTerm m t0}

      t1 : {t1 : Term |
               isSubsetOf (freeVars t1) s
            && consistentScopesTerm m t1}

      i : Int
      s : Set Int
      m : IntMap (Set Int)
      ss : Subst Term
    Constraint id 168
    |
578 |     , let t' = substitute (freeVarsSubst ss') m ss' t1
    |                                                  ^^^
```

We can get quickly that the predicate in the required type is one of the conjuncts in the refinement type of a parameter of `substitute`. That is `ConsistentScopedSubst`, a type alias we declared in the same module, and in this case expands as follows.

```
    {ss':Subst Term |
          isSubsetOf (freeVarsSubst ss') (freeVarsSubst ss')
       && consistentScopesSubst m ss'
    }
```

To get at the missing lemma, in this case we only need to connect the predicates in the inferred and the required refinement types. Let us prune the irrelevant bits from the error message first.

```
    The inferred type
      ss' : Subst {v : Term | consistentScopesTerm m v}
    is not a subtype of the required type
      VV : {VV : Subst Term | consistentScopesSubst m VV}
```

And then we can substitute `VV` by `ss'` in the goal, which gives pretty much the lemma statement.

```
    The inferred type
      ss' : Subst {v : Term | consistentScopesTerm m v}
    is not a subtype of the required type
      ss' : {ss' : Subst Term | consistentScopesSubst m ss'}
```

When there are static check failures, insight is often necessary to identify a missing lemma or a missing precondition. Recursive functions like `skolemize` start with a core set of conjuncts that sometimes needs to be grown as static checks reveal the need of stronger postconditions for the result of the recursive calls.

Maturity. Maybe relatedly, the maturity of refinement type checkers in general, and Liquid Haskell in particular, is rather lacking still. We have encountered a non-negligible number of bugs (18) in the Liquid Haskell tooling and usability issues while conducting our study. Our source code contains comments explaining the defects where we were affected. The sources of most of these defects seem to locate in the Liquid Haskell implementation rather than the SMT solver, and there was an issue encountered in the SMT solver[10]. Fortunately, none of them look very difficult to address, but they do have a severe impact on user experience in aggregate.

Besides, Liquid Haskell lacks support for many standard features of Haskell. In our code we have been using the simplest possible style of programming. There are no GADTs, no type families, and minimal use of type classes (since Liquid Haskell has some support for type classes [10]). At the moment, pushing for more demanding programming patterns is likely to surface more inconveniences. Aiming for the simplest style is, therefore, a pragmatic constraint of the current implementation. For further insight on the challenges of using Liquid Haskell, Gamboa et al. [6] report on a study that collects the voices of its users.

On the performance front, all of the SMT-LIB queries in the unification example run in 11 seconds, 0.04 seconds for `Subst2.hs`, and 0.03 seconds in `Subst1.hs`. That is sometimes faster than compiling a module with the GHC compiler. Where things get slower is when measuring Liquid Haskell end-to-end, which spends several seconds checking the examples and interacting with the SMT solver (3 minutes when checking unification, 4 seconds checking `Subst2.hs`, 1.5 seconds checking `Subst1.hs`). The authors deem that performance of Liquid Haskell can be improved to approach that of the SMT solver queries, and probably further by reducing the number of queries.

Composability. Perhaps one of the biggest compromises when encoding properties in the type-checker is that one needs to narrow the expressible properties to a feasible set that allows to write a supporting library. If we wanted to have static checks like those of the unification example, we would need new type encodings. Or in other words, new type indices need to be conceived to relate the parameters of our functions.

$$\begin{aligned} \mathit{skolemize} :: \quad & \mathit{Scope}\ s_1 \dots s_n \\ & \to \mathit{Formula}\ f_1 \dots f_j \\ & \to \mathit{State}\ t_1 \dots t_k\ (\mathit{Scope}\ e_1 \dots e_l)\ (\mathit{Formula}\ o_1 \dots o_m) \end{aligned}$$

Then there would be the effort of writing a library, and later on there would be the effort of composing the encodings of different libraries when more than one such is needed. Suppose we started with the static checks to avoid name captures as in Section 2, and we wanted to add the scopes checks required to deal with unification variables. With refinement types we need to add the corresponding conjuncts to the refinement types, and perhaps some *phantom* parameter like `m` here.

```
substituteFormula
  :: s:Set Int
  -> m:IntMap (Set Int)
  -> ss:ConsistentScopedSubst s m
  -> {f:ScopedFormula (domain ss) | consistentScopes m f}
  -> {v:ScopedFormula s |
          formulaSize f == formulaSize v
       && consistentScopes m v
       && existsCount v = existsCount f
     }
```

Besides the usual scope checks, we are checking that the size of the formula is preserved, that the amount of existential binders is preserved, and that the unification scopes in the output are those in the input formula and in the range of the substitution. We also check that substitution preserves the consistency of the unification scopes.

5 Comparable systems

Liquid Haskell is not the only tool reaching to SMT solvers for static checks. The most similar tool is F* [18], which is based on

[10] We found a problem in the Z3 SMT solver, which sprung some follow up issues further linked in the original issue: https://github.com/Z3Prover/z3/issues/7770

a refinement type system as well. Another family of related systems are those with Hoare-style pre- and post-conditions to functions such as Why3 [4] and Dafny [11] (impure functional programming languages), or ESC/Java [5] and Frama-C [7] (imperative languages).

All of the above systems could have served as a vehicule for our case study, though the further we go down that list, the more different the language is too Liquid Haskell, and the more adaptation that would require. The type systems also get weaker and the latest the language is in the list, the more one has to lean on the SMT for static checks.

6 Conclusions

The tooling is not ready for widespread use. Yet it is plausible that in a decently close future, we have access to SMT solvers and refinement-types to assist us in our programming.

Refinement types enable a more direct expression of properties, particularly when the SMT solver supports the relevant theories. Reasoning mechanisms are reused from the existing tooling, instead of encoding them in the type checker. This makes easier both to enforce our own invariants and to compose properties coming from different sources.

The generality of the approach, and the simplicity with which it enables composition of different properties, are unique features that make it a strong candidate to impact programming practice in the future.

Through our two case studies, we have tried to make a first step in understanding how we will be best able to leverage future such tools, even in situations where we can manage to use current type-checkers today. As a closing note, let us reproduce the principles that we have proposed throughout the article.

Principle 1. *Typically, refinement types allow you to reduce the trusted code base, but they also offer you a choice. When it is easier to prove a result by hand than with the SMT solver, you can assume the property and justify it informally.*

Principle 2. *Refinement types add a layer of subtyping on top of your type system. When your program is best modeled with subtyping you should consider refinement types.*

Principle 3. *Refinement types and abstract types are best at enforcing different kind of properties. You should use the simpler solution for each property that you need, as refinement types and abstract types mix well.*

Principle 4. *Functions with refined signature and without mix well. You should first use refinement types on function with the best power-to-weight ratio. You can incrementally add stronger types on more functions as your program evolves.*

Principle 5. *It is easier to express properties and to use an SMT solver when assumptions are explicit rather than reconstructing assumptions that are implicit. Do not hesitate to pass assumptions as arguments to functions, even if those arguments are not used by the function.*

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: a modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects* (Amsterdam, The Netherlands) *(FMCO'05)*. Springer-Verlag, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *The SMT-LIB Standard: Version 2.7*. Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[3] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[4] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128.

[5] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2013. PLDI 2002: Extended static checking for Java. *SIGPLAN Not.* 48, 4S (July 2013), 22–33. https://doi.org/10.1145/2502508.2502520

[6] Catarina Gamboa, Abigail Elena Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. In *Proceedings of the 46th ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Seoul, South Korea) *(PLDI 2025)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3729327

[7] Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles (Eds.). 2024. *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications* (1 ed.). Springer Cham. XXIV, 697 pages. https://doi.org/10.1007/978-3-031-55608-1

[8] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (June 2023), 25 pages. https://doi.org/10.1145/3591283

[9] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 441–459. https://www.usenix.org/conference/osdi21/presentation/lehmann

[10] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying replicated data types with typeclass refinements in Liquid Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 216 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428284

[11] K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

[12] Dougal Maclaurin, Alexey Radul, and Adam Paszke. 2023. The Foil: Capture-Avoiding Substitution With No Sharp Edges. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (Copenhagen, Denmark) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 10 pages. https://doi.org/10.1145/3587216.3587224

[13] Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation* 1, 4 (09 1991), 497–536. https://doi.org/10.1093/logcom/1.4.497 arXiv:https://academic.oup.com/logcom/article-pdf/1/4/497/3817142/1-4-497.pdf

[14] Dale Miller and Alexandre Viel. 2022. The undecidability of proof search when equality is a logical connective. *Annals of Mathematics and Artificial Intelligence* 90, 5 (01 May 2022), 523–535. https://doi.org/10.1007/s10472-021-09764-0

[15] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* 12 (July 2002), 393–434. https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner/

[16] Patrick Redmond, Gan Shen, Niki Vazou, and Lindsey Kuper. 2023. Verified Causal Broadcast with Liquid Haskell. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (Copenhagen, Denmark) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 13 pages. https://doi.org/10.1145/3587216.3587222

[17] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and decidable type inference for GADTs. *SIGPLAN Not.* 44, 9 (Aug. 2009), 341–352. https://doi.org/10.1145/1631687.1596599

[18] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. *SIGPLAN Not.* 51, 1 (Jan. 2016), 256–270. https://doi.org/10.1145/2914770.2837655

[19] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 2022. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference*

on Programming Language Design and Implementation (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 31–45. https://doi.org/10.1145/3519939.3523708

[20] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: experience with refinement types in the real world. *SIGPLAN Not.* 49, 12 (Sept. 2014), 39–51. https://doi.org/10.1145/2775050.2633366

[21] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. *SIGPLAN Not.* 49, 9 (Aug. 2014), 269–282. https://doi.org/10.1145/2692915.2628161

[22] Robin Webbers, Klaus von Gleissenthall, and Ranjit Jhala. 2024. Refinement Type Refutations. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 305 (Oct. 2024), 26 pages. https://doi.org/10.1145/3689745

[23] Beta Ziliani and Matthieu Sozeau. 2015. A unification algorithm for Coq featuring universe polymorphism and overloading. *SIGPLAN Not.* 50, 9 (Aug. 2015), 179–191. https://doi.org/10.1145/2858949.2784751

[24] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. Cryptology ePrint Archive, Paper 2017/536. https://eprint.iacr.org/2017/536

Draft papers

Unrestricted Grammar Design and Visualization

A Design Recipe and Dynamic Visualization Tool

Marco T. Morazán
Seton Hall University
South Orange, NJ, USA
morazanm@shu.edu

Andrés M. Garced
Seton Hall University
South Orange, NJ, USA
maldona2@shu.edu

Tijana Minić
University of Washington
Seattle, WA, USA
tminic@uw.edu

Abstract

Unrestricted grammars are an integral part of many Formal Languages and Automata Theory courses. Typically, Computer Science students struggle developing such grammars given that most courses restrict design to pencil and paper. A new trend in Formal Languages and Automata Theory education takes a programming-based approach. Such an approach allows students to get immediate feedback on their designs before submitting for grading or developing a correctness proof. Nonetheless, developing unrestricted grammars remains a challenging and frustrating task for many students. This article presents a novel design recipe and dynamic visualization tool to help students implement unrestricted grammars. The design recipe provides scaffolding steps to guide students from a problem statement to a verified implementation. Given a grammar and a word in the grammar's language, the dynamic visualization tool displays a step-by-step creation of a history-preserving derivation graph. In addition, when provided with an invariant predicate for the yield of the derivation graph, the dynamic visualization tool uses node coloring to indicate if the invariant holds. The implementation of the visualization tool is discussed. Empirical results suggest that the use of a heuristic search and the use of a bootstrapped skew binomial heap to implement a priority queue have a significant impact on performance. In addition, empirical results from a formative human-factors study are presented. These results suggest that students find the tool useful to understand and debug unrestricted grammars. The broad implications and the limitations of the human-factors study are also discussed.

CCS Concepts

• **Software and its engineering** → *Domain specific languages*; **Functional languages**; • **Theory of computation** → **Grammars and context-free languages**; • **Applied computing** → **Education**; • **Human-centered computing** → **Visualization**.

Keywords

Unrestricted Grammar, Derivation Visualization, Formal Languages and Automata Theory Education, Functional Programming, Design Recipes, Grammar validation and verification

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Marco T. Morazán, Andrés M. Garced, and Tijana Minić. 2025. Unrestricted Grammar Design and Visualization: A Design Recipe and Dynamic Visualization Tool. In . ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Formal Languages and Automata Theory (FLAT) textbooks (e.g., [25, 27, 28, 32, 39]) expose students to unrestricted grammars (urgs)[1]. Such grammars are the counterparts of Turing Machines (tms). Whereas Turing machines decide or semidecide a language, unrestricted grammars generate words in recursively enumerable languages. It is important for students to study such grammars because they are used extensively, for example, in natural language processing [22, 23, 42] and in visual programming languages [4]. More recently, closely related grammars, called mildly context-sensitive grammars, are used to improve the complexity of determining word membership in a language [16, 24].

Most FLAT textbooks do not provide guidance for the systematic development of urgs. Instead, the unspoken assumption is that students learn to develop them by studying a few examples. Furthermore, unlike for regular grammars (rgs) and context-free grammars (cfgs), the visualization of word derivations is commonly not addressed. In fairness, this is likely due to the complexity involved in visualizing derivations using urgs. For rgs and cfgs, it suffices to present a derivation tree for a given word. Such a visualization preserves the derivation history (i.e., the production rules used) to help students understand how rules are nondeterministically applied. For unrestricted grammars, word derivation visualization is more complex given that a substitution context may involve mutating several elements in the yield to become several different elements or none at all. Thus, requiring using a derivation graph to visualize a derivation in which nodes have multiple predecessors.

To address these problems, the domain-specific language FSM (Functional State Machines)–embedded in Racket [13]–was developed for the FLAT classroom. In FSM, programmers may define urgs and use them to derive words. Tightly-coupled with FSM is the textbook *Programming-Based Formal Languages and Automata Theory* (PBFLAT) [32]. It presents a design recipe for grammar development. A design recipe is a series of steps, each with a specific outcome, that guide the design and implementation of urgs. Design recipes were first developed by Felleisen et al. to teach beginners how to program [10] and later expanded by Morazán to a two-semester curriculum for beginners. In PBFLAT, Morazán extends the use of design recipes to the FLAT classroom. Recently, he extended the

[1] Such grammars are also referred to as Type-0 in the Chomsky hierarchy [7].

grammar design recipe found in PBFLAT with verification steps to develop cfgs [34].

This article extends the design recipe work done for cfg verification to urg verification. In addition, it presents a dynamic visualization tool for word derivation using urgs. Given a urg and a word in the grammar's language, the tool presents a history-preserving step-by-step construction of a derivation graph. To aid in grammar validation and verification, the dynamic visualization tool also accepts an optional user-defined invariant predicate as input. The invariant predicate tests if the (current) yield satisfies properties that must hold for the grammar to be correct. In this manner, students can test their design and implementation before submitting for grading or attempting a proof.

The article is organized as follows. Section 2 briefly presents the syntax for urgs in FSM, the grammar design recipe, and the new verification steps. Section 3 presents the dynamic visualization tool for word derivation using urgs. Section 4 discusses the implementation of word derivation and presents empirical performance measurements. Section 5 presents how the dynamic visualization tool is used for debugging. Section 6 presents the results of a formative human-factors study conducted to test the effectiveness of the tool and discusses its implications and limitations. Section 7 compares and contrasts with related work. Finally, Section 8 presents concluding remarks and directions for future work.

2 Unrestricted Grammars in FSM

2.1 Syntax

The 4-input urg constructor is make-grammar and its signature is: N Σ R S → urg. The arguments from left to right correspond to the nonterminals, the input alphabet, the production rules, and the starting nonterminal. Each production rule in R is of the form $\alpha \rightarrow \beta$, where α is a nonempty arbitrary number of nonterminal and terminal symbols containing at least one nonterminal and β is an arbitrary number of nonterminal and terminal symbols. The interface for urgs includes an observer for each component. For instance, grammar-nts returns N and grammar-rules returns R. The observer grammar-derive, given a grammar and a word, returns a trace of the derivation rules used when the given word is in the given grammar's language. Given that determining word membership in the language of an unrestricted grammar is undecidable [25, 32], grammar-derive may not terminate when given a word not in the grammar's language.

An embedded domain-specific language is used to write FSM unit tests. This embedded language includes check-derive? and check-not-derive?. Each requires a grammar and an arbitrary number of words. Both use grammar-derive to test each word. If the correct result is achieved for all words the test passes. Otherwise, a failed test is reported and the words that made the test fail are highlighted in the code. Unit tests must be written with care because, as mentioned above, grammar-derive may not terminate when given a word not in the grammar's language.

2.2 Design Recipe for Grammars

To provide design scaffolding, students are presented with the design recipe for urgs displayed in Figure 1. The first six steps constitute the design recipe for grammars defined in [32]. Step 1 asks for a meaningful grammar name and the input alphabet. Step 2 asks the student to define the meaning of needed nonterminals. In this step, both a design idea for derivation and generation "promises" are developed. At this point, these promises define, in informal language, how terminal symbols are generated. Based on the results of Steps 1 and 2, Step 3 asks for the development of the production rules. Step 4 asks for unit tests. The tests must include words in the grammar's language. Tests for words not in the grammar's language may be written as comments. Step 5 asks for the grammar's implementation. Step 6 requires running the tests from Step 4. If tests fail or there are construction errors, debugging is done by revisiting the answers for the design recipe steps performed.

Figure 1 The Design Recipe for urgs

(1) Pick grammar name and specify the input alphabet
(2) Define the needed nonterminals and specify the starting nonterminal
(3) Develop the production rules
(4) Write unit tests
(5) Implement the grammar
(6) Run the tests and redesign if necessary
(7) Develop a loop invariant predicate for derivation
(8) For words in L(G), prove invariant predicate holds
(9) Prove that L = L(G)

Steps 7–9 (highlighted in green) are the new grammar verification steps for urgs. These steps are inspired by analogous verification steps developed for rgs and cfgs [34]. To describe these steps assume that the language is L and the grammar is G. For Step 7, students are encouraged to think of the production rules as performing mutations on the yield of a derivation graph. At a high-level of abstraction, the derivation process is described as follows:

```
while (contains-nonterminals? yield)
    apply-rule()
```

The apply-rule procedure mutates the yield by nondeterministically applying a rule in G that leads to a successful derivation. At this point in the course, students feel comfortable enough with nondeterminism to accept this abstraction. Step 7 asks for the implementation of a loop-invariant predicate (akin to what is required by program verification using Hoare logic [19, 21]). This predicate takes as input a yield and tests conditions that must be true to establish grammar correctness. Step 8 asks for a proof that for w∈L(G) the loop invariant holds throughout the derivation process. This proof is done by induction on the number of times the loop is executed. Building on Step 8, Step 9 asks for a proof that L = L(G).

2.3 An Illustrative Example

To illustrate the design recipe in action, we develop a urg for L={w|w=$a^n b^n c^n$, n∈ ℕ}. Figure 2 displays the developed urg. The results for Step 1 are displayed on lines 8 (grammar name is anbncn) and 10 (the input alphabet is (a, b, c)).

For Step 2, students outline a design idea for word generation:

(1) Generate an arbitrary number of ABC
(2) Rearrange As, Bs, and Cs to $A^n B^n C^n$
(3) From As, Bs, and Cs, first generate cs, then bs, and finally as

Figure 2 An FSM urg for $a^n b^n c^n$

```
1  #lang fsm
2
3  ;; Nonterminal Documentation:
4  ;; S: Generates words in a^n b^n c^n, where n is a natural number
5  ;; A: A promise to generate an "a" in the context AI
6  ;; B: A promise to generate a "b" in the context BH
7  ;; C: A promise to generate a "c" in the context CG
8  (define anbncn
9    (make-grammar '(S A B C G H I)
10                 '(a b c)
11                 '((S → ϵ)   (S → ABCS) (S → G)
12                   (BA → AB) (CA → AC)  (CB → BC)
13                   (CG → Gc) (BG → BH)  (BH → Hb)
14                   (AH → AI) (AI → Ia)  (I → ϵ))
15                 'S))
16
17 (check-derive? anbncn '(a b c) '(a a b b c c) '())
18 ;; (check-not-derive? anbncn '(a) '(b) '(c) '(c b a) '(a a b b c c c))
```

The generation of terminal symbols must be done in the right context. A c is only generated when everything to the right (in the yield) is a c. A b is generated when everything to the right is an arbitrary number of bs followed by cs. An a is generated when everything to the right is an arbitrary number of as followed by an arbitrary number of bs which are followed by cs. Throughout the derivation process, |A|+|a|=|B|+|b|=|C|+|c|. To achieve the correct generation of terminals, I is used to generate as, H is used to generate bs, and G is used to generate cs. To satisfy Step 2, the role of the nonterminals is documented on lines 3–7.

For Step 3, the production rules are displayed on lines 11–14. The rules for S on line 11 generate an arbitrary number of ABC (satisfying the first step of the design idea) ending with a G to start the generation of cs. The rules on line 12 rearrange the nonterminals to be in the right order (satisfying the second step of the design idea). The rules on lines 13–14 generate the nonterminals in the right context.

For Step 4, the unit tests are displayed on lines 17–18. The program in Figure 2 satisfies Step 5. Running the program reveals that there are no errors and all the tests for words that ought to be in L(anbncn) pass.

For Step 7, students develop a predicate for the loop invariant. The design from Step 2 suggests the following conditions need to be invariant:

- Number of A/a = number of B/b = number of C/c
- The yield contains at most one of the following nonterminals: S, G, H, I
- S∈yield ⇒ yield ends with S
- G∈yield ⇒ yield ends with Gc^*
- H∈yield ⇒ yield ends with Hb^*c^+
- I∈yield ⇒ yield ends with $Ia^*b^+c^+$
- yield∈Σ^* ⇒ yield∈L

The resulting predicate is displayed in Figure 3. In the interest of brevity, the auxiliary predicates are not displayed.

Figure 3 Loop invariant for grammar displayed in Figure 2

```
1  ;; (listof (N ∪ Σ)) → Boolean
2  ;; Purpose: Determine if loop invariant holds for the given yield
3  (define (anbncn-inv yield)
4    (and (equal-num-abc? yield)
5         (one-of-S-G-H-I? yield)
6         (implies (member 'S yield) (S-INV yield))
7         (implies (member 'G yield) (G-INV yield))
8         (implies (member 'H yield) (H-INV yield))
9         (implies (member 'I yield) (I-INV yield))
10        (implies (no-nt? yield)    (in-L? yield))))
```

For Step 8, students prove that the loop invariant holds by induction on, n, the number of rules applied (or equivalently on the number of times the loop is executed). The base case is established for n=0 as follows:

> The yield only contains S. Observe that the only nonterminal is S and the yield ends with it. Thus, the loop invariant holds.

The inductive step is established by assuming that the loop invariant holds and showing that it holds after applying a rule. For instance, consider the use of CG → Gc. The needed Hoare triple is:

```
;; yield∈XCGc*, where X is everything before C
apply-rule()
;; yield=XGcc* ⇒ yield∈XGc* ⇒ anbncn-inv
```

The precondition holds because we assume anbncn-inv holds (i.e., by inductive hypothesis). The post condition holds because a G remains in the yield, the C before G in the yield is removed, and a c is added to the end of the yield. Such a triple is developed for all rules and in the interest of brevity they are omitted.

For Step 9, a proof for L = L(anbncn) is required. This is established as follows:

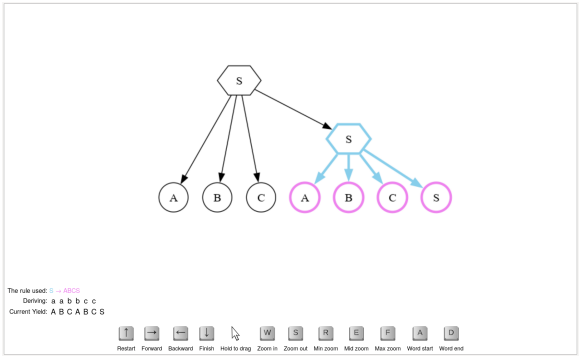
> Assume w∈L
> This means w=$a^n b^n c^n$. Given that the loop invariant always holds, there is a derivation starting with S that yields w. Thus, w∈L(anbncn).
>
> Assume w∈L(anbncn)
> Given that the loop invariant always holds, any yield generated starting with S must be of the form $a^n b^n c^n$. Thus, w∈L.

It is noteworthy that Steps 8 and 9 are not beyond any Computer Science student that has taken an Introduction to Discrete Mathematics course.

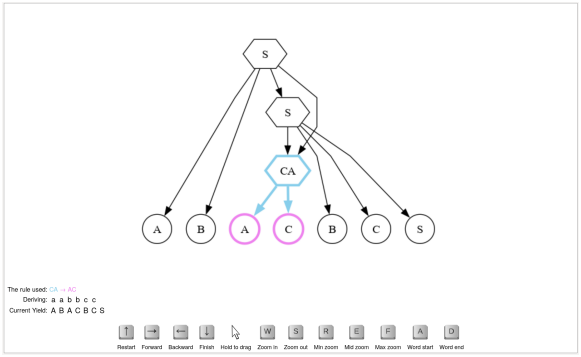
3 Dynamic Visualization Tool

Despite working with a design recipe, designing and implementing urgs remains a difficult task for some students. To further aid students in their efforts to understand urgs, a new dynamic visualization walks students through the steps of a derivation. Given as input a grammar, a word in the grammar's language, and, optionally, a loop invariant predicate, the tool visually traces rule applications used in the word's derivation. At each step, it displays a derivation graph that preserves the history of rules applied. In

Figure 4 Snapshots of the dynamic visualization tool.



(a) After generating ABC twice.



(b) After swapping A and C.

addition, the yield nodes are filled in green or red, respectively, to indicate whether the invariant holds or fails when the predicate is provided.

To reduce the extraneous cognitive load associated with how to use and understand the tool, development follows the Norman principles of effective design [36]. These principles are applied as done previously for other FSM visualizations [15]. Snapshots of the dynamic visualization deriving `'(a a b b c c)` using the grammar from Figure 2 are displayed in Figure 4. There are 3 vertically aligned parts: the derivation graph, the informative messages, and the instructions. In the derivation graph, circular nodes denote elements in the (current) yield, which are always aligned at the bottom for reading ease. Hexagon nodes denote prior yield elements that have been replaced. The informative messages consist of three parts: the last production rule used, the word being derived, and the current yield. In this manner, the user is provided information to understand the current state of the visualization. The instructions contain a graphic for each action available to the user. The user may interact with the visualization via keystrokes or by clicking the instruction graphics with the mouse. The first four graphics, the arrow keys, allow the user to navigate through the visualization. The cursor graphic informs the user they may use their mouse to move the derivation graphic, and scroll the yield or word being derived when they are too long to fit within the visualization frame. The next four key icons are for zooming

Figure 5 urg for $\mathtt{i}^n\mathtt{bi}^m\mathtt{bi}^{n+m}$.

```
1  #lang fsm
2
3  (define ADD-CSG
4    (make-urg
5      '(S A E I)
6      '(b i)
7      `((S ⟶ AbAbE)
8        (A ⟶ ,ϵ)  (A ⟶ iIA)
9        (Ii ⟶ iI) (Ib ⟶ bI)  (IE ⟶ Ei)
10       (E ⟶ ,ϵ))
11     'S))
```

actions. Finally, the last two graphics scroll to the beginning and the end of the informative messages.

The informative message for the last production rule used and the last nodes changed in the derivation graph are color coded to correspond. For example, in Figure 4a the last rule used is: S ⟶ ABCS. The S is highlighted in light blue and ABCS is highlighted in pink. The corresponding nodes in the derivation graph are highlighted using the same colors. The S expanded is rendered as a hexagonal node given that it has been substituted.

The history of the derivation is preserved by the paths in the derivation graph. For instance, Figure 4b displays the state of the visualization after one step forward from the state displayed in Figure 4a. The rule used is: CA ⟶ AC. Given that CA is expanded, it is rendered as a hexagonal node. The predecessors of CA are the two S nodes, which indicate that the elements in CA were generated by the two S nodes. In this case, the root S generated the C and the other S generated the A.

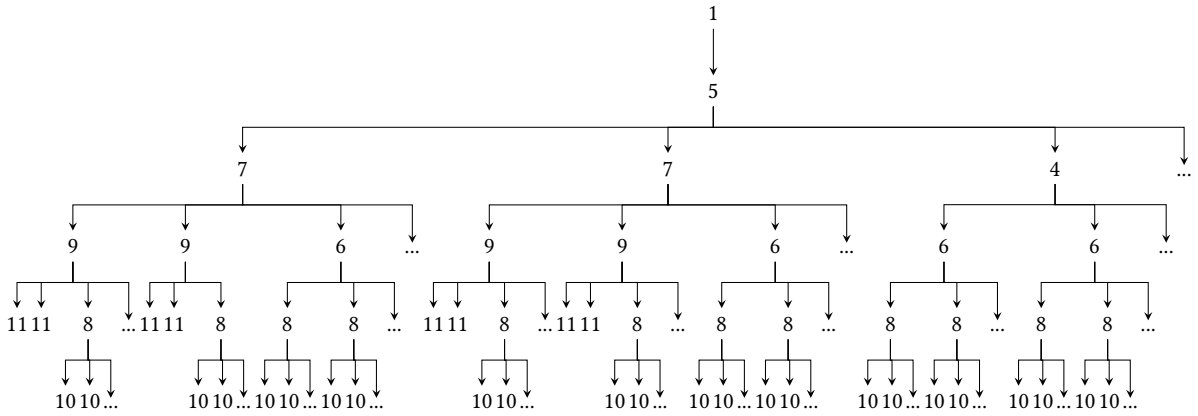
If the loop invariant predicate is provided, the yield nodes are color-filled to indicate if the invariant holds: green if it holds and red if it does not hold. Figure 10 displays a dynamic visualization snapshot when the loop invariant does not hold. Observe that the coloring of the yield nodes clearly communicate this fact.

4 Implementation

4.1 Design

There are two primary sets of technical challenges that are faced when developing a visualization tool for derivations using a urg. The first is the generation and storing of images. These challenges are addressed by the use of a restricted form of parallelism for image generation and the use of thunks to store images in nonvolatile memory as done for all FSM visualizations [15]. The second is the generation of a derivation. The search for a derivation using a urg is difficult, because the search space may grow exponentially. To illustrate this, consider deriving `'(i i b i b i i i)` using the urg for valid unary-addition words displayed in Figure 5. The language of the grammar is $\mathtt{L} = \mathtt{i}^n\mathtt{bi}^m\mathtt{bi}^{n+m}$. This grammar nondeterministically generates the unary arguments using the nonterminal A. For each terminal symbol, i, generated, a nonterminal symbol, I, is generated for the generation of the result's corresponding unary number. The Is are migrated to the end of the yield where E is used to generate the result's corresponding, i, terminal symbol.

Figure 6 Depiction for the search space to derive `'(i i b i b i i i)`.



The derivation search space may be described as a tree, where each tree path represents a (potential) derivation. Some paths may be infinite given that a derivation may not terminate. Figure 6 displays a representation of the tree-like search space for `'(i i b i b i i i)`. In the graphic, each step in the derivation is denoted by the length of its yield. At the root, there is a single derivation, `(S)`, containing the starting nonterminal. Thus, 1 is the root in the search space's rendering. There is a single rule that applies to `S`. As a consequence, there continues to be a single derivation, `(AbAbE S)`, at the next level of the tree, whose yield has length 5. Briefly, the next level of the tree captures that two rules apply to either `A` and one rule applies to `E` resulting in 5 possible derivations (due to space limitations 2 derivations are captured by ellipsis): two of length 7 by applying `(A -> iIA)` to each `A`, two of length 4 by applying `(A -> ϵ)` to each `A`, and one of length 4 by applying `(E -> ϵ)`.

Initially, a breadth-first traversal of the search was implemented. Although this implementation strategy works well for short words, it quickly became apparent that for words of modest length derivation became a bottleneck. This resulted in students becoming frustrated when using the tool. Simply stated, the performance of a (standard) breadth-first traversal of the search space degrades too quickly.

To improve performance, the well-known path-finding A^* algorithm [14] is used to perform a heuristic search for a derivation. Heuristic development was led by two goals: favor finding the shortest derivation and avoid exploring infinite derivations. To this end, the heuristic converged on is: explore the derivation with the shortest yield first. By exploring the shortest yield first, two expectations are pursued: explore infinite derivations that always make the yield longer less frequently and favor exploring derivations that are likely obtained using fewer derivation steps. The heuristic, of course, is not infallible. A programmer, for example, can write a `urg` that artificially makes derivations longer. Thus, defeating the purpose of the heuristic, which will not find a derivation faster by exploring shorter yields. We are, therefore, betting on programmers writing grammars to generate short derivations.

To make the heuristic search more efficient, derivations must be accumulated in a manner that makes finding the derivation with the shortest yield fast. To this end, a priority queue that maintains the derivations sorted by the length of their yield is used. The implementation of the derivation algorithm is outlined in Figure 7. The function `urg-derive` returns, if it exists, a derivation for the given word using a priority queue of derivations and a set of yields previously generated. Initially, the queue of derivations contains a single derivation for the starting nonterminal and the set of generated

Figure 7 Word derivation impleentation.

```
;; word (queueof derivations) (setof yield) → derivation
;; Purpose: Return a derivation for the given word
(define (urg-derive word derivs generated-yields)
  (if (heap-empty? derivs)
      '()
      (let* ([first-deriv (find-min derivs)]
             [next-yields
               (filter (λ (yd) (same-len? word yd))
                       (apply-one-step
                         (car first-yield)
                         generated-yields))])
        (if (member word next-yields)
            (reverse (cons word first-deriv))
            (let* ([new-derivs
                     (map (λ (yd)
                            (insert-queue yd
                                          first-yield))
                          next-yields)]
                   [new-generated-yields
                     (set-union next-yields
                                generated-yields)]
                   [new-derivs
                    (add-queue
                      new-derivs
                      (delete-min-queue tovist))])
              (urg-derive word
                          new-derivs
                          new-generated-yields)))))
```

yields only contains a yield consisting of the starting nonterminal. If the queue of derivations is empty, the empty derivation is returned (line 5), given that the given word cannot be derived. Otherwise, the first queue element (i.e., the derivation with the shortest yield) is used to create new yields (lines 7–11). New yields are generated by an auxiliary function, `apply-one-step`, that only returns yields that have not been generated before, thus, avoiding repeated exploration and getting caught in some infinite derivations. The new yields are filtered to remove any that only contain terminal symbols that are not of the same length as the given word (this is done by the auxiliary function `same-len?`). If the given word is a member of the new yields generated, the found derivation is returned (line 13). Otherwise, `urg-derive` is recursively called with the given word, a new queue of derivations obtained by adding the new derivations, and a new set of generated yields obtained by adding the newly found yields (lines 14–25).

To briefly illustrate the derivation algorithm, consider how the algorithm proceeds when the priority queue contains the derivations denoted by level 2 of the tree in Figure 6. Recall that there are 5 derivations: two of length 7 and three of length 4. The shortest derivation is processed first (i.e., any of the 3 derivations of length 4). These derivations fail to derive `'(i i b i b i i i)` or produce yields that are no longer the shortest (e.g., the yields of length longer than 7 depicted under the subtree rooted at 4). When this occurs, the algorithm backtracks to explore the derivation with the shortest yield accumulated in the queue (e.g., 4's siblings of length 7).

4.2 Performance

Two criteria are used to measure performance. The first measures the number of times a production rule is applied during the search of a derivation. This aims to give an indication of how much work is performed by the derivation algorithm. The second measures execution time using different priority queue implementations. This aims to determine the best data structure to use to implement a priority queue.

Measurements are presented using two `urgs`: `ANBNCN` displayed in Figure 2 and `ADD-CSG` displayed in Figure 5. These grammars have different derivation characteristics. `ANBNCN` grows the length of the yield linearly (i.e., by applying `(S → ABCS)` the yield's length is expanded by 4 along the same derivation). The yield for `ADD-CSG` grows exponentially (i.e., for every `A` two (sub)derivations, using `(A- > ϵ)` and `(A- > iIA)`, are generated). All execution times are measured using a Lenovo X13 Yoga ThinkPad (the laptop issued to students at our institution).

4.2.1 Derivation Steps. We first examine word derivation using `ANBNCN`. When deriving `'(a a b b c c)` without using the heuristic, the number of times a production rule is applied is 25. Deriving the same word using the heuristic results in 59 production rule applications. The gap in the number of rules applied increases with word length. Deriving `'(a a a a b b b b c c c c)`, results in 76 production rule applications when the heuristic is not used and 4954 production rule applications when the heuristic is used. To understand this performance gap, the dynamic behavior of `urg-derive` needs to be examined. Repeated use of `(S → ABCS)` creates new longer derivations. The successful derivation must use this rule 4 times for `'(a a a a b b b b c c c c)`. The heuristic favors exploring (partial) derivations with shorter yields. This means that all derivations that use `(S → ABCS)` fewer that 4 times must first be determined unsuccessful before exploring the derivation that uses this rule 4 times. In contrast, breadth-first search does not delay the exploration of the derivation that uses this rule 4 times. It simultaneously explores all derivations[2]. Thus, discovering a successful derivation using `(S → ABCS)` 4 times before fully exploring all derivations that use this rule fewer times. In essence, `ANBNCN` is an `urg` instance that defeats the use of the heuristic.

In contrast, the use of the heuristic search has a striking positive impact on the number production rule applications when using `ADD-CSG`. Deriving `'(i b i b i i)` using breadth-first results in 5962 production rule applications. Using the heuristic search, results in 107 production rule applications. This is a 98.2% reduction. The performance of the heuristic search becomes even more striking when longer words are derived. Deriving `'(i i b i b i i i)` using breadth-first search results in 251,299 production rule applications. When using the heuristic search, the number of production rule applications is 418. This represents a reduction of over 99.8%. This significant performance gap is explained by examining the exponentially growing number of derivations generated as depicted, for example, in Figure 6 illustrating the derivation of a word of length 8. The number of yields that have a length longer than 8 are the majority. Breadth-first simultaneously search explores them all. The use of the heuristic, on the other hand, never explores derivations longer than 8 by favoring the exploration of shorter yields. Thus, performing fewer production rule applications. In essence, `ADD-CSG` is an `urg` instance that favors the use of the heuristic.

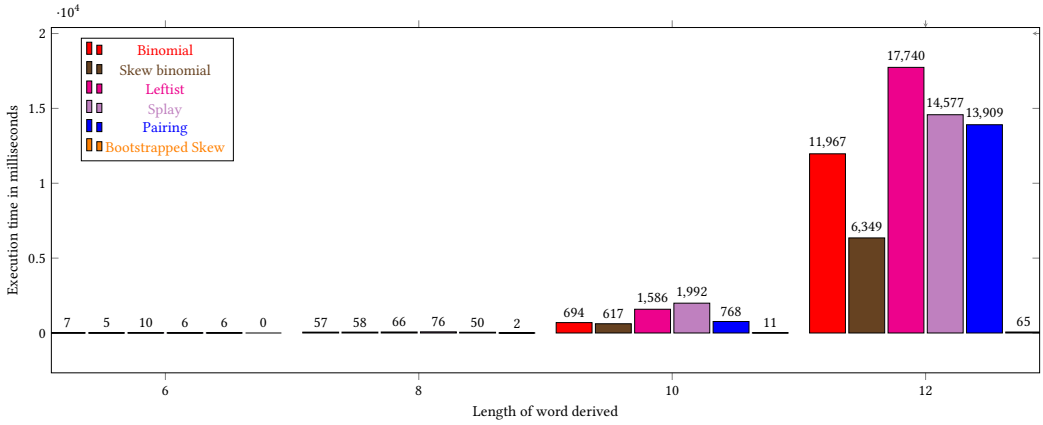
The impact of the number of production rules applied is reflected in running times. Finding a derivation of a word of length 15 (a nontrivial modest length classroom) using `ANBNCN` without the heuristic takes 112 ms. Using the heuristic, on the other hand, takes 66040 ms. Finding a derivation of a word of length 14 (also a nontrivial modest length classroom example) using `ADD-CSG` without the heuristic is unfeasible. Using the heuristic, on the other hand, takes 498 ms.

Based on the reported empirical evidence, it is clear that the use of the heuristic can have a negative impact on performance. For instance, we expect this to occur when the number of derivations grows linearly. On the other hand, the use of the heuristic has a significant positive impact on performance when the number of derivations grows exponentially. This means that the use of the heuristic allows students to effectively use the tool for a greater variety of grammars. In addition, it is reasonable to expect that the derivation search space for `urgs` grows exponentially for most languages that require such a grammar to be defined. This is another reason for concluding that the use of the heuristic is the best choice.

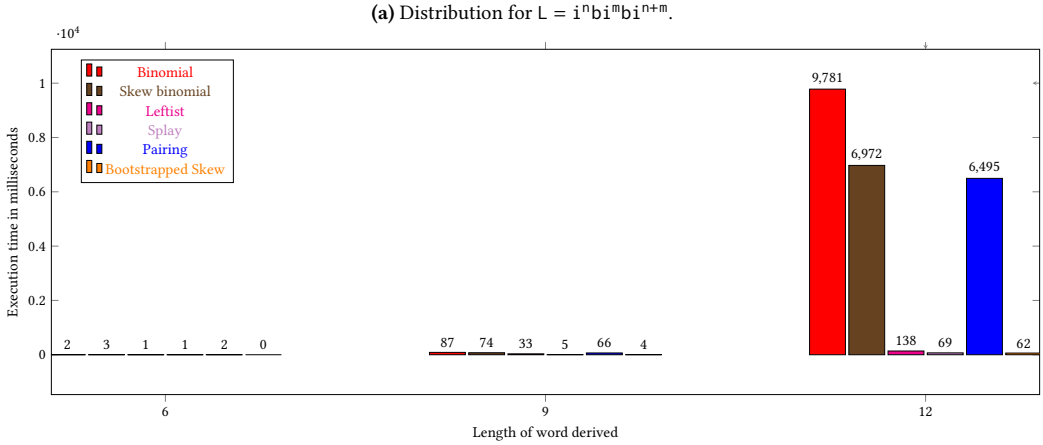
4.2.2 Priority Queue Implementations. Finally, there is a plethora of ways a priority queue may be implemented. For the development of the described tool, 6 different implementations using different types of heaps were explored: binomial, skew binomial, leftist, splay, pairing, and bootstrapped skew binomial (see Okasaki for a description of these [37]). Words of length 6, 8, 10, and 12 in $\mathtt{L} = \mathtt{i}^n\mathtt{b}\mathtt{i}^m\mathtt{b}\mathtt{i}^{n+m}$

[2]Including those that use the rule more than 4 times.

Figure 8 Data distribution for execution time for different heap implementations.



(a) Distribution for $L = i^n bi^m bi^{n+m}$.



(b) Distribution for $L = a^n b^n c^n$.

and words of length 6, 9, and 12 in $L = a^n b^n c^n$ are used to measure execution time on a Lenovo X13 Yoga ThinkPad. For each of the 7 test words, derivation execution time (in ms) is measured 50 times, for a total of 350 experiments.

Figure 8a and Figure 8b present the distribution of average execition times for each word length using each of the 6 heap implementations. The best results across all word lengths are seen for the bootstrapped skew binomial heap. For words of length 12, we observe that the performance using the bootstrapped skew binomial heap is 2-3 orders of magnitude faster than the other implementations. Given this large performance gap when compared to other heap implementations, despite being harder to implement, we recommend (and have adopted) the use of a bootstrapped skew binomial heap. We attribute the observed performance to a bootstrapped skew binomial heap having constant time operations for finding the minimum and inserting an element, both of which are operations used in `urg-derive` in Figure 7.

5 Visual Debugging

To illustrate the tool's usefulness in debugging, consider the student-designed buggy `urg` displayed in Figure 9. The student

Figure 9 Buggy student `urg` for $a^n b^n c^n$.

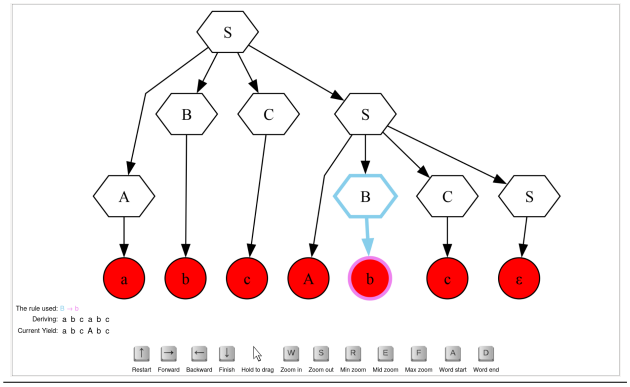
```
#lang fsm

(define anbncn
  (make-grammar '(S A B C)
                '(a b c)
                '((S → ϵ)   (S → ABCS)
                  (BA → AB) (CA → AC)  (CB → BC)
                  (C → c)   (B → b)    (A → a))
                'S))

(check-derive? anbncn '(a b c) '(a a b b c c) '())

(define (anbncn-inv yield)
  (and (equal-num-abc? yield)
       (terminals-in-order? yield)
       (implies (member 'S yield) (S-INV yield))
       (implies (no-nt? yield) (in-lang? yield)))
```

Figure 10 Snapshot displaying the invariant failing to hold for the `urg` from Figure 9.



claims to have successfully implemented a simpler `urg` for $a^n b^n c^n$ using a similar loop invariant to the one displayed in Figure 3. The main difference between the two implementations is that the student has removed the contexts for generating terminal symbols. That is, the nonterminals `G`, `H`, and `I` as well as the associated production rules have been removed from the grammar. In their place, the student has added to the invariant predicate a condition for all terminal symbols to be in relative correct order. To guarantee the relative ordering of terminal symbols, the student has kept the production rules for rearranging `A`s, `B`s, and `C`s. This indicates a degree of misunderstanding regarding nondeterminism. Apparently, the student believes that the rearrangement production rules must be used before using the rules to produce terminal symbols. Under this light, it is suggested to add `'(a b c a b c)` to the unit tests (i.e., to `check-derive?`). Much to the student's surprise, the tests pass.

Given that the word is derived, it is suggested to the student to use the dynamic visualization to debug. Upon stepping through the derivation, the student discovers the first step where the invariant fails as displayed in Figure 10. From the derivation history displayed in the derivation graph, the student can observe that the rearrangement rules are never used. Therefore, the last rule used, `B` $\longrightarrow$ `b`, mutates the yield to contain a `b` that does not satisfy the condition for all terminal symbols to be in relative order. This cements the understanding that production rules may be nondeterministically used in any order (or not at all) and that using production rules in the right context matters.

6 Formative Human-Factors Study

To measure student perceptions about the described visualization tool, a small anonymous survey-based formative study was performed with the students enrolled in Seton Hall University's undergraduate Formal Languages and Automata Theory course. The course enrolled 10 students: all Computer Science majors in their 3rd or 4th year of studies between the ages of 20 and 23. They self-identified as: 30% female and 70% male. All 10 students volunteered to participate in the study. None of them received any benefit or compensation for their participation.

The survey presents students with a series of statements and asks respondents to indicate how strongly they agree or disagree. Responses are provided using a Likert scale [26]: 1 (Strongly disagree) to 5 (Strongly agree), with 3 as a neutral response. The data collected is presented around 4 themes: visualization quality, word derivation visualization, informative messages, and invariant visualization. In the presentation below, we denote responses 4 and 5 as tend to agree and responses 1 and 2 as tend to disagree.

6.1 Visualization Quality

To measure perceptions on the overall quality of the visualization tool, students are presented with the following statements:

Q1.1 The visualization is clear.
Q1.2 The visualization is easy to use.
Q1.3 The visualization is visually appealing.
Q1.4 The visualization is useful.

The distribution of responses is displayed in Figure 11. For Q1.1, we observe that an overwhelming majority of respondents, 90%, tend to agree that the visualization is clear. None disagreed with the statement. This is an important result because it suggests users understand the graphics displayed. For Q1.2, we observe that half of the respondents tend to agree that the visualization is easy to use, 40% feel neutral about this statement, and 10% tend to disagree. This suggests that the respondents tend to feel less strongly about the visualization's ease of use than its clarity. This result is not entirely surprising for two reasons: how quickly the graph grows in size and placing the yield of the derivation graph at the bottom. As a result of fast graph growth, a user needs to either drag or zoom out of the main visualization graphic to observe the derivation history and the changing yield at the same time. During our design phase, we considered not placing the yield at the bottom and discarded this idea given that it became hard to determine the yield of the derivation graph. Nonetheless, the overwhelming majority of respondents do not tend to disagree with the statement.

For Q1.3, we observe that a majority of respondents, 60%, tend to agree that the visualization is visually appealing. We also observe that 20% are neutral about this statement and 20% tend to disagree (but not strongly). Thus, an overwhelming majority does not tend to disagree. Nonetheless, a nontrivial proportion of the respondents do not tend to agree. This lukewarm result is also expected given that large derivation graphs are inherently difficult to read. That is, they contain a large number of nodes and crisscrossing edges. Unfortunately, this is unavoidable. Placing the yield at the bottom, in our opinion, does appear to result in more readable graphs.

For Q1.4, we observe that an overwhelming majority of respondents, 90%, tend to agree that the visualization is useful. In fact, an overwhelming majority, 70%, strongly agree. This is the statement that respondents tend to agree with the most in the entire survey. This result is very encouraging, because it suggests that our goal to help students understand `urg`s has been achieved. The remaining questions in the survey shed light on why the respondents believe the visualization is useful.

6.2 Word Derivation Visualization

To further explore, in part, why the visualization tool is useful, the survey presents the following statements:

Figure 11 Data distribution for visualization quality.

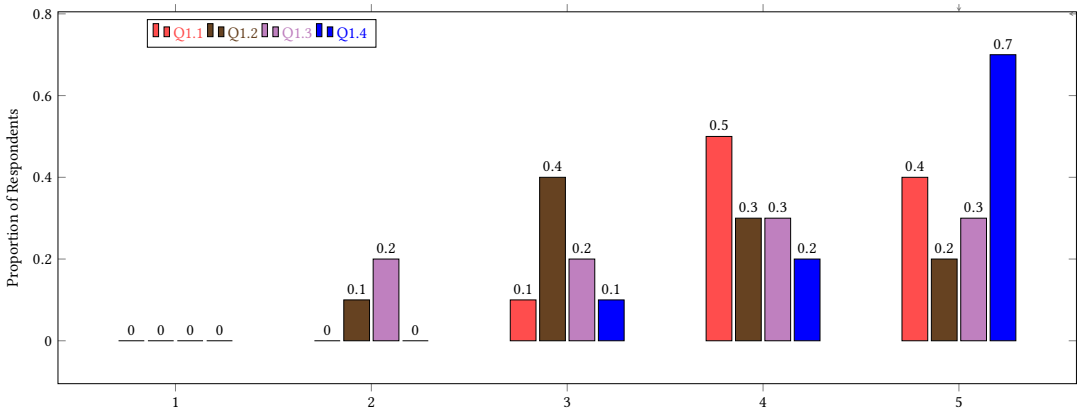
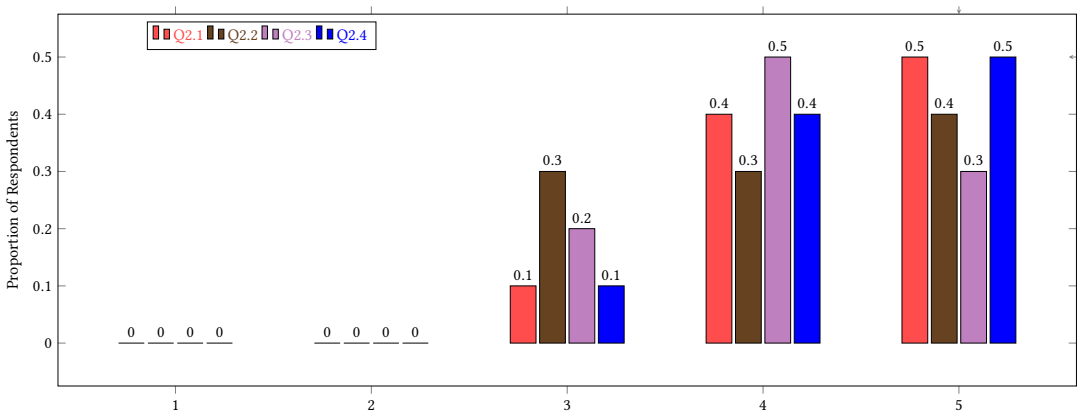


Figure 12 Data distribution for word derivation visualization.



Q2.1 The visualization helped me understand how words are derived.

Q2.2 The visualization helped me understand how production rules may be nondeterministically applied.

Q2.3 The visualization clearly preserves the history of the derivation.

Q2.4 Word derivation visualization is useful for debugging grammars.

The distribution of responses is displayed in Figure 12. For Q2.1, we observe that an overwhelming majority of respondents, 90%, tend to agree that the visualization helped them understand how words are derived. None tended to disagree with the statement. This result is encouraging, because it suggests that visualizing the construction of a derivation graph has achieved our goal to help students understand word derivation. For Q2.2, we observe that an overwhelming majority of respondents, 70%, tend to agree that the visualization helped them understand how production rules may be nondeterministically applied. This is also a very encouraging result, because it is challenging, even for some small `urgs`, to find a sequence of production rule applications to derive a word. It is expected for respondents to tend to agree less with this statement than with Q2.1. The visualization clearly illustrates a word's derivation, but is less useful in explaining why a rule is chosen for use. For Q2.3, we observe that an overwhelming majority of respondents, 80%, tend to agree that the visualization clearly preserves the history of the derivation. This is an encouraging and surprising result. Our expectation, was that understanding the history of the derivation using a derivation graph would be significantly more difficult when compared to, for example, understanding the derivation history using derivation trees for context-free and regular grammars. Collectively, these results shed light on why respondents find the visualization useful (i.e., Q1.4). They suggest that most students, in part, find it useful, because it helps them understand word derivation, nondeterministic production rule application, and the derivation history.

For Q2.4, we observe that an overwhelming majority of respondents, 90%, tend to agree that word derivation visualization is useful for debugging grammars. None of the respondents disagree. This result further sheds light on why respondents feel that visualizing word derivation is useful. It directly helps them to correctly implement `urgs`.

Figure 13 Data distribution for informative messages.

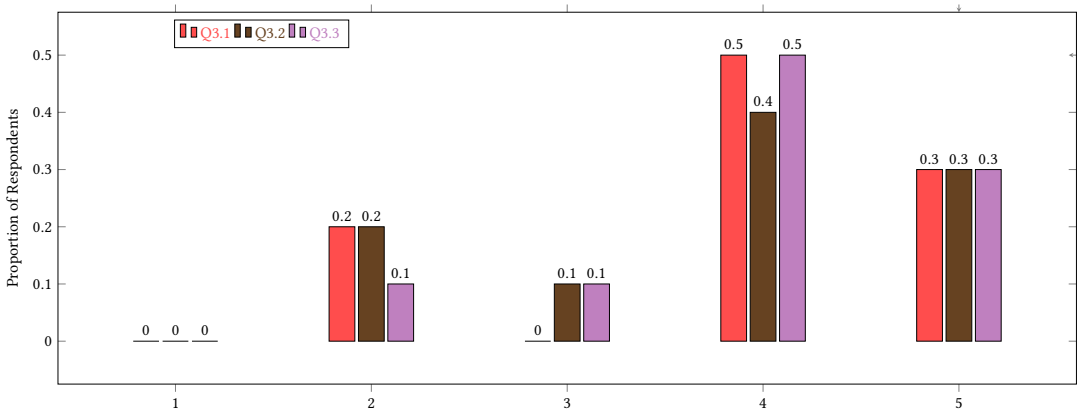
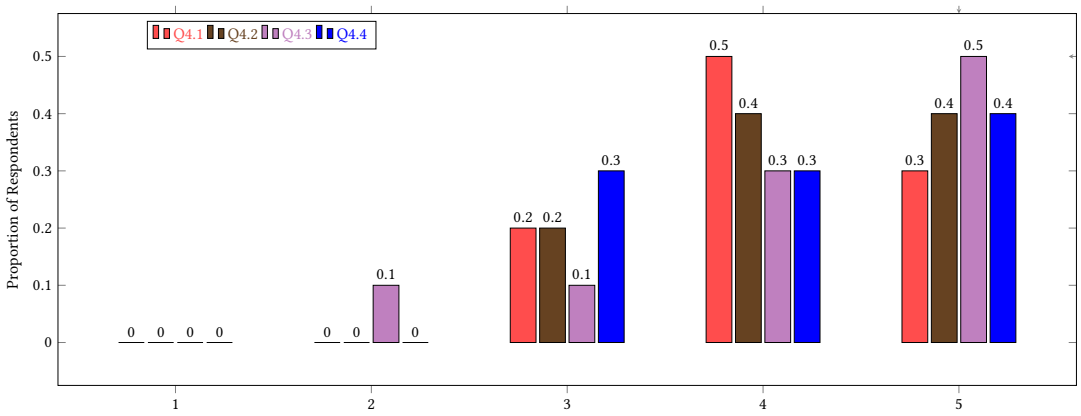


Figure 14 Data distribution for invariant visualization.



6.3 Informative Messages

To measure perceptions regarding informative messages, the survey presents the following statements:

Q3.1 The informative messages are clear.
Q3.2 The informative messages are useful.
Q3.3 The informative messages help me understand the last step performed in the visualization

The distribution of responses is displayed in Figure 13. For Q3.1, we observe that an overwhelming majority of respondents, 80%, tend to agree that the informative messages are clear. We also observe that a minority, 20%, tends to disagree (but not strongly). This result suggests that, overall, respondents understand the informative messages, but that more work needs to be done to make informative messages clear for an even larger proportion of students. One design possibility we plan to explore is including a brief explanation for why hexagon nodes are created.

For Q3.2, we observe that an overwhelming majority of respondents, 70%, tend to agree that informative messages are useful. This is an encouraging result, because it suggests that our goal to satisfy the Norman principles related to discoverability and conceptual model are met. This is further confirmed by the results obtained from Q3.3, which indicate that an overwhelming majority of respondents, 80%, tend to agree that the informative messages help them understand the last step performed in the visualization.

6.4 Invariant Visualization

Invariant visualization perceptions are explored through the following survey statements:

Q4.1 Invariant visualization is clear.
Q4.2 Invariant visualization is useful.
Q4.3 Invariant visualization is useful for debugging grammars.
Q4.4 Invariant visualization is useful to develop correctness proofs.

The distribution of responses is displayed in Figure 14. For Q4.1, we observe that an overwhelming majority of respondents, 80%, tend to agree that invariant visualization is clear. None of the respondents tend to disagree. This suggests that respondents understand when and why invariants hold and fail. In part, we attribute this success to always placing the yield of the derivation graph at the bottom and to the informative messages. Thus, students can examine with little search effort the yield and the last production rule applied to determine why an invariant holds or fails.

For Q4.2, we observe that an overwhelming majority of respondents, 80%, tend to agree that invariant visualization is useful. None of the respondents tend to disagree. This result is very encouraging, because it suggests that our goal to help students formally design urgs has been achieved. The remaining questions about invariant visualization shed light on why the respondents believe such visualization is useful.

For Q4.3, we observe that an overwhelming majority of respondents, 80%, tend to agree that invariant visualization is useful for debugging grammars. For Q4.4, we observe that an overwhelming majority of respondents, 70%, tend to agree that invariant visualization is useful to develop correctness proofs. Collectively, these results help us understand why respondents find invariant visualization useful. They suggest that respondents appreciate the practical value (i.e., useful in debugging) and the theoretical value (i.e., useful in developing proofs).

6.5 Discussion and Limitations

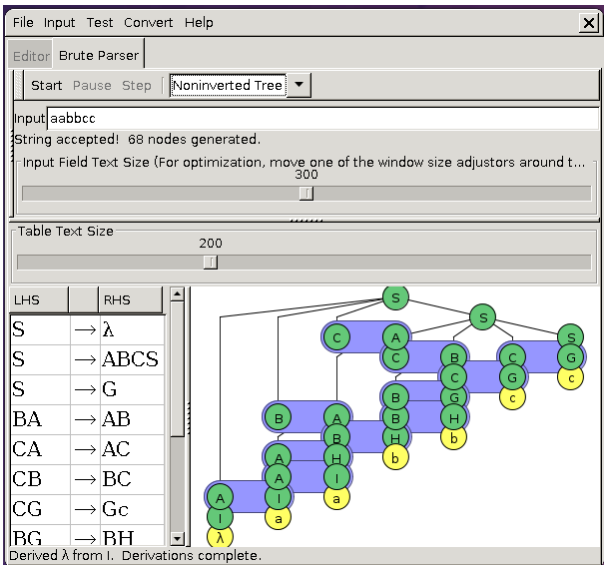
This sections discusses some of the broad implications suggested by our results that merit further study. In addition, it discusses some of the limitations of the empirical study.

6.5.1 Implications. The unsystematic treatment of nondeterminism in the Computer Science curriculum, in part, contributes to students finding it difficult to understand [3]. This is unfortunate, because nondeterminism plays a role in modern programming. For example, it is relevant in probabilistic programming [1, 6, 30], quantum programming [11, 43], and logic programming [5, 12]. The work presented in this article suggests a pedagogy for systematically teaching Computer Science students about nondeterminism. That is, the results obtained from Q2.1 and Q2.3 suggest that systematic program development along with the use of dynamic visualizations may be an effective way to teach students about nondeterminism.

The resistance to formal methods among Computer Science students is well documented [17, 44]. In essence, many Computer Science students are frustrated by proof development for two reasons. First, they lack experience with proof development and, therefore, commonly fail to understand how to proceed. Second, they usually have no way of getting feedback on their assertions before, for example, submitting for grading. The results obtained from Q4.1–Q4.4 suggest that visualizations might offer a means to dismantle the resistance to formal methods. The described visualization tool offers immediate feedback on assertions (i.e., invariant predicates). When these do not hold, students have an opportunity to revise them. When they represent the weakest precondition [8] and they hold, students can more confidently proceed to writing a proof. The described approach is not limited to the derivation of words using a urg. It can be used, for example, to teach program verification using Hoare Logic [20]. For instance, a domain-specific imperative language may be developed for Hoare triples that allows assertion/invariant visualization to help students.

Many modern dynamic visualization tools fail to provide informative messages to their users regarding the process visualized. For instance, JFLAP provides for conversions between models of computation (e.g., a nondeterministic finite-state automaton to a deterministic finite-state automaton), but performs such operations with no informative messages to help the user develop a sound mental model. We have evidence, from previous studies [31, 33, 35], that visualizations with informative messages are well-received by students. The results obtained from Q3.1–Q3.3 further confirm that informative messages can play a pivotal role in developing understanding among students.

Figure 15 JFLAP derivation graph using the urg from Figure 2



6.5.2 Limitations. The presented empirical study suffers from several limitations. Perhaps, the most salient is that the sample is small and is drawn from a mostly homogeneous population where most Computer Science students are European American men. It is unclear, therefore, if the observed results would hold for larger samples drawn from a diverse population. Although it is not unreasonable to believe that they would, such a hypothesis does require empirical validation. To address this shortcoming, empirical studies shall be conducted over several iterations of the course and efforts will be made to make such studies multi-university.

This study, as is common in Computer Science Education studies, did not compare a control group with an experimental group. Therefore, the effect of the independent variable (i.e., the introduction of the described visualization tool) is difficult to isolate. It is not a straightforward matter to address this shortcoming because many university administrators would raise ethical concerns about treating students differently solely for the benefit of a study.

The quantitative nature of our study is also a limitation. Although the data collected is amenable to statistical analysis, it precludes the nuances of individual student experiences. The presented study, for instance, does not capture the moments when students found a visualization feature extremely useful and when they wished for a new feature. To address this concern, future studies will include open-ended questions, interviews, and observational sessions.

7 Related Work

7.1 Visualizing Grammar Derivations

The best known tool for visualizing word derivation using a `urg` is `JFLAP` [40, 41]. Similar to `FSM`, this tool allows users to input a `urg` and a word to visualize the derivation and displays the stepwise construction of the derivation graph. An example of such a derivation graph is displayed in Figure 15. In this example, the word `aabbcc` is being derived using the same `urg` for $a^n b^n c^n$ displayed in Figure 2. We can observe several characteristics that make derivation graphs in `JFLAP` hard to read. First, `JFLAP` groups nodes to mutate the yield by connecting them with blue ovals without uniting them into a single node. In contrast, `FSM` unites such nodes into a single hexagonal node. Second, `JFLAP` constrains the visualization to the size of the visualization frame. This results in overlapping nodes being rendered. In contrast, `FSM` does not constrain the visualization to the size of the visualization frame. Users may zoom in and drag into the visualization frame the parts of the derivation graph they wish to inspect more closely. A third contrasting feature is that `JFLAP` does not allow the user to step backwards in the derivation. The only way a user may review previous steps is by restarting the visualization. In contrast, `FSM` allows the user to step backwards through the visualization to reexamine previous derivation steps without restarting the visualization. Finally, a fourth contrasting feature is that `JFLAP` does not provide support for invariant visualization.

Another popular visualization tool for the `FLAT` classroom is `Automata Tutor` [9]. This popular tool, however, does not provide support for `urgs`. To the best of our knowledge, in fact, besides `JFLAP` there are no other popular tools for visualizing word derivation using `urgs`. Furthermore, the dynamic visualization tool described in this article is the first to support the grammar verification process by providing visual support for invariant validation.

The heuristics used for the derivation algorithm described in this article are inspired by the heuristics used by Rekers and Schürr to parse context sensitive graph grammars. They also used a priority queue as a heuristic to improve the runtime characteristics of their algorithm, although in contrast to the algorithm described in this article their algorithm is bottom-up and makes uses of a cost function based on probabilities.

7.2 Derivation Search

Rekers and Schürr describe a parsing algorithm for a visual programming language using a context-sensitive graph grammar [38]. As the work presented in this article, they suggest the use of heuristics to reduce the search space and priority queues to improve performance. In contrast, they discuss a mechanism for pruning failed derivations attempts as early as possible, which is something that forms part of our future work. In further contrast, it is unclear what parts of their algorithm, if any, are implemented and they do not provide any empirical performance measurements.

Harris describes extensions to SLR(1) and LALR(1) bottom-up deterministic parsing [2] for unrestricted grammars [18]. The approach presents an optimized algorithm that builds a parse tree from the leaves to the root. A disadvantage of the presented approach is that there is no algorithm which produces the SLR(1) or LALR(1) parse tables for a given unrestricted grammar. Instead, Harris presents a "computational procedure which can be carried out by hand in many cases." In contrast, the work presented in this article takes a top-down approach that we believe is more intuitive for students being exposed to unrestricted grammars for the first time. Bottom-up optimizations such as the one described by Harris are more appropriate for students that have prior experience with `urgs`.

8 Concluding Remarks

A novel design recipe and a dynamic visualization tool for unrestricted grammars in `FSM` is presented. The design recipe includes new steps to help students with grammar verification and serves as a *lingua franca* for instructors and students to discuss grammar design, validation, and verification. In addition, a novel dynamic visualization tool for word derivation is presented. The tool was developed using the Norman principles of effective design to reduce the extraneous cognitive load on users. Given a grammar and a word, it presents the user with a stepwise construction of a derivation graph. In addition, the tool supports the design process by providing an invariant visualization feature designed to be used before formally proving grammar correctness. The data gathered from a small formative study suggests that the presented visualization tool is well-received and considered useful by students to understand word derivation, to debug grammars, and to develop correctness arguments.

Future work involves three goals. The first is conducting an in-depth human factors study in the classroom to measure student perceptions of the tool. The focus will be to determine if the tool helps students understand derivation using a `urg`, helps students debug their designs, and helps students utilize their invariants effectively for grammar verification. The second is the creation of a dynamic visualization to help students better understand the transformation of an unrestricted grammar to Kuroda normal form [29]. The third is to explore techniques to reduce the exploration of derivations that are likely to fail.

References

[1] Aguirre, A., Birkedal, L.: Step-Indexed Logical Relations for Countable Nondeterminism and Probabilistic Choice. Proc. ACM Program. Lang. **7**(POPL) (Jan 2023). https://doi.org/10.1145/3571195

[2] Aho, A.V., Ullman, J.D.: The theory of Parsing, Translation, and Compiling. Prentice-Hall, Inc., USA (1972)

[3] Armoni, M., Ben-Ari, M.: The Concept of Nondeterminism: Its Development and Implications for Teaching. SIGCSE Bull. **41**(2), 141–160 (jun 2009). https://doi.org/10.1145/1595453.1595495

[4] Ates, K.L.: Context-Sensitive Graph Grammar Induction. Ph.D. Thesis, The University of Texas at Dallas, Dallas, TX (December 2013)

[5] Ballantyne, M., Sanna, R., Hemann, J., Byrd, W.E., Amin, N.: Multi-stage Relational Programming. Proc. ACM Program. Lang. (POPL) (Jan 2025), to appear

[6] Bingham, E., Chen, J.P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., Goodman, N.D.: Pyro: Deep Universal Probabilistic Programming. J. Mach. Learn. Res. **20**(1), 973–978 (Jan 2019)

[7] Chomsky, N.: Three Models for the Description of Language. IRE Transactions on Information Theory **2**(3), 113–124 (1956). https://doi.org/10.1109/tit.1956.1056813

[8] Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)

[9] D'Antoni, L., Helfrich, M., Kretinsky, J., Ramneantu, E., Weininger, M.: Automata Tutor v3. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II. p. 3–14. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-53291-8_1

[10] Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs: An Introduction to Programming and Computing. MIT Press, Cambridge, MA, USA, Second edn. (2018)

[11] Feng, Y., Xu, Y.: Verification of Nondeterministic Quantum Programs. In: Proceedings of the 28th ACM International Conference on Architectural Support

for Programming Languages and Operating Systems, Volume 3. p. 789–805. ASPLOS 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3582016.3582039
[12] Fischer, S., Kiselyov, O., Shan, C.c.: Purely Functional Lazy Non-Deterministic Programming. SIGPLAN Not. **44**(9), 11–22 (Aug 2009). https://doi.org/10.1145/1631687.1596556
[13] Flatt, M., Findler, R.B., PLT: The Racket Guide. PLT (2024), https://docs.racket-lang.org/guide/, last accessed: June 2025
[14] Foead, D., Ghifari, A., Kusuma, M.B., Hanafiah, N., Gunawan, E.: A Systematic Literature Review of A* Pathfinding. Procedia Computer Science **179**, 507–514 (2021). https://doi.org/10.1016/j.procs.2021.01.034, 5th International Conference on Computer Science and Computational Intelligence 2020
[15] Garced, A.M., Minić, T., Morazán, M.T.: On the implementation of dynamic visualizations: A case-study using fsm visualizations. In: Proceedings of the 36th Symposium on Implementation and Application of Functional Languages. p. 51–64. IFL '24, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3723325.3723340
[16] Gómez-Rodríguez, C., Kuhlmann, M., Satta, G.: Efficient Parsing of Well-Nested Linear Context-Free Rewriting Systems. In: Kaplan, R., Burstein, J., Harper, M., Penn, G. (eds.) Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics. pp. 276–284. Association for Computational Linguistics, Los Angeles, CA (jun 2010), https://aclanthology.org/N10-1035/
[17] Gries, D.: The Science of Programming. Springer-Verlag, Berlin, Heidelberg, 1st edn. (1987)
[18] Harris, L.A.: SLR(1) and LALR(1) Parsing for Unrestricted Grammars. Acta Informatica **24**, 191–209 (April 1987). https://doi.org/10.1007/BF00264364
[19] Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259
[20] Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259
[21] Hoare, C., Jifeng, H.: Unifying Theories of Programming. Prentice Hall series in computer science, Prentice Hall (1998)
[22] Huybregts, R.: The Weak Inadequacy of Context-Free Phrase Structure Grammars. Van periferie naar kern pp. 81–99 (1984)
[23] Joshi, A.K.: Tree Adjoining Grammars: How Much Context-Sensitivity is Required to Provide Reasonable Structural Descriptions?, p. 206–250. Studies in Natural Language Processing, Cambridge University Press (1985)
[24] Kanazawa, M.: The Pumping Lemma for Well-Nested Multiple Context-Free Languages. In: Diekert, V., Nowotka, D. (eds.) Developments in Language Theory. pp. 312–325. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
[25] Lewis, H.R., Papadimitriou, C.H.: Elements of the Theory of Computation. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edn. (1997). https://doi.org/10.1145/300307.1040360
[26] Likert, R.: A Technique for the Measurement of Attitudes. Archives of Psychology **140**, 1–55 (1932)
[27] Linz, P.: An Introduction to Formal Languages and Automata. Jones and Bartlett Publishers, Inc., USA, 5th edn. (2011)
[28] Martin, J.C.: Introduction to Languages and the Theory of Computation. McGraw-Hill, Inc., USA, 3 edn. (2002)
[29] Meduna, A.: Automata and languages: theory and applications. Springer-Verlag, Berlin, Heidelberg (2000)
[30] van de Meent, J.W., Paige, B., Yang, H., Wood, F.: An Introduction to Probabilistic Programming (2021), https://arxiv.org/abs/1809.10756
[31] Minić, T., Morazán, M.T.: Visualizing Construction Algorithms for Closure Properties of Regular Languages. In: ITiCSE 2025: Proceedings of the 2025 Conference on Innovation and Technology in Computer Science Education V. 1. ITiCSE 2025, Association for Computing Machinery, New York, NY, USA (2025)
[32] Morazán, M.T.: Programming-Based Formal Languages and Automata Theory - Design, Implement, Validate, and Prove. Texts in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-43973-5
[33] Morazán, M.T., Minić, T.: Finite-State Automaton To/From Regular Expression Visualization. Electronic Proceedings in Theoretical Computer Science **405**, 36–55 (jul 2024). https://doi.org/10.4204/eptcs.405.3, in Proceedings TFPIE 2024
[34] Morazán, M.T.: Grammar Verification for Students: A Grammar Design Recipe with Verification Steps. In: Proceedings of the 17th International Conference on Computer Supported Education - Volume 2: CSEDU. pp. 662–669. INSTICC, SciTePress (2025). https://doi.org/10.5220/0013216500003932
[35] Morazán, M.T., Minić, T.: Finite-State Automaton To/From Regular Expression Visualization. Electronic Proceedings in Theoretical Computer Science **405**, 36–55 (Jul 2024). https://doi.org/10.4204/eptcs.405.3
[36] Norman, D.A.: The Design of Everyday Things. Basic Books, Inc., USA (2002)
[37] Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, USA (1999). https://doi.org/10.1017/CBO9780511530104
[38] Rekers, J., Schürr, A.: A Parsing Algorithm for Context-Sensitive Graph Grammars. Tech. Rep. 95-05, Leiden University (1995), https://liacs.leidenuniv.nl/assets/PDF/TechRep/tr95-05.pdf
[39] Rich, E.: Automata, Computability and Complexity: Theory and Applications. Pearson Prentice Hall (2019)
[40] Rodger, S.H.: JFLAP: An Interactive Formal Languages and Automata Package. Jones and Bartlett Publishers, Inc., USA (2006)
[41] Rodger, S.H., Bressler, B., Finley, T., Reading, S.: Turning automata theory into a hands-on course. In: Baldwin, D., Tymann, P.T., Haller, S.M., Russell, I. (eds.) Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2006, Houston, Texas, USA, March 3-5, 2006. pp. 379–383. ACM (2006). https://doi.org/10.1145/1121341.1121459
[42] Shieber, S.M.: Evidence Against the Context-Freeness of Natural Language. In: The Formal complexity of natural language, pp. 320–334. Springer (1985)
[43] Zhou, L., Barthe, G., Strub, P.Y., Liu, J., Ying, M.: CoqQ: Foundational Verification of Quantum Programs. Proc. ACM Program. Lang. **7**(POPL) (Jan 2023). https://doi.org/10.1145/3571222
[44] Zingaro, D.: Another Approach for Resisting Student Resistance to Formal Methods. SIGCSE Bull. **40**(4), 56–57 (nov 2008). https://doi.org/10.1145/1473195.1473220

AI-Assisted Program Design Using Structural Recursion

Marco T. Morazán
Seton Hall University
South Orange, NJ, USA
morazanm@shu.edu

Abstract

The age of artificial intelligence is upon us and with it comes the responsibility of using and teaching how to use coding assistants ethically and effectively. All too often, students believe that using a poorly designed prompt to generate code makes them successful programmers. More often than not, nothing could be further from the truth as all that they are doing is engaging in plagiarism–attributing to themselves code used in the training set of a large language model that was designed and implemented by someone else. This article presents the first attempt to apply a successful programming pedagogy based on design recipes to the use of coding assistants. It presents the design and implementation of software based on structural recursion using functional programming in a mainstream programming language. The methodology put forth recognizes that prompts are programs and, as such, must be designed. Furthermore, it illustrates how prompt engineering ought to be informed by program design steps that are performed when programming without coding assistants. Finally, it also illustrates how debugging, instead of being an ad hoc endeavor, ought to be done by revisiting design steps.

CCS Concepts

• **Applied computing** → **Education**; • **Software and its engineering** → **Software design techniques**; • **Computing methodologies** → **Artificial intelligence**.

Keywords

AI-Assisted Programming, Prompt Engineering, Design Recipes, Functional Programming

ACM Reference Format:
Marco T. Morazán. 2025. AI-Assisted Program Design Using Structural Recursion. In *Proceedings of the 37th Symposium on Implementation and Application of Functional Languages (IFL 2025), October 1–3, 2025, Montevideo, Uruguay*. ACM, New York, NY, USA, 13 pages.

1 Introduction

The age of programming assisted by artificial intelligence (AI) is here with promises to greatly enrich software development prowess and perils to inhibit the education of future software developers. The potential for coding assistants to be both helpful and harmful means that their use must be carefully tempered in Computer Science (CS) education. On the one hand, a coding assistant harnesses the power of generative AI [15, 29] using large language models [8, 25] to simplify daily programming tasks. For example, they make code suggestions and provide answers to questions (such as the meaning of a cryptic error message). On the other hand, a coding assistant can become a crutch to programmers, especially students, that become overly dependent on them [1]. They can lead programmers to engage in unethical conduct (e.g., attribute to themselves programs they did not develop), develop poor programming habits (e.g., accepting the use of features that they do not understand), and overreliance (e.g., accepting code suggestions without carefully checking them) [5]. All of these behaviors undermine, especially in students, the ability to learn and to solve problems.

To understand how to teach productive use of coding assistants, it is necessary for students to understand (at least at a very high level of abstraction) that the quality of a code suggestion or the answer to a question is linked to the quality of a *prompt*. A prompt is a set of instructions crafted in a natural language to help a generative AI model provide the desired result. It is important to realize that when a programmer uses a coding assistant, at least two programs need to be written: a prompt (i.e., a program for the generative AI model) and the actual program that is desired (e.g., a solution to a problem). The natural question that arises, of course, is why write two programs when only one program is needed? The answer is that often, but not always, to obtain a working program it is easier to write a prompt than to directly write the program. Much like programs written in the strict syntax of a programming language are designed, given that they are programs, prompts must also be designed. The science of writing prompts is coined *prompt engineering*.

The question now becomes: *How is a prompt designed?* Popular internet sources state that prompts need to be clear and precise to obtain high-quality results from a coding assistant [12, 30]. Although this is sound advice, it ignores that a prompt, as stated, must be designed. That is, there are systematic steps that may be followed that provide scaffolding for prompt development. Such scaffolding is extensively used in CS pedagogy based on functional programming and type-driven design [13, 22–24]. This article describes an application of functional programming in the CS education domain. It puts forth a design-based approach for the development of programs using prompts and coding assistants. The focus is on the development of programs using structural recursion. Program development is illustrated using GitHub Copilot and functional programming in Python. The methodology put forth, however, is programming language and coding assistant agnostic. The result is a systematic approach to software development using a coding assistant that squarely places programming and prompt engineering in the realm of science and not the realm of art as decribed by some CS authors (e.g., [26]).

IFL 2025, Montevideo, Uruguay

The article is organized as follows. Section 2 briefly discusses the advantages and perils of using coding assistants. Section 3 presents design recipes for function development: one to design functions and one to design prompts. Section 4 illustrates the use of the outlined design recipes using a nontrivial example to compute π using the Chudnovsky algorithm [10, 11]. Section 5 contrasts and compares with related work. Finally, Section 6 presents concluding remarks and directions for future work.

2 Advantages and Perils of Coding Assistants

The use of coding assistants to implement software presents both advantages and perils. The advantages need to be harnessed throughout the software development process. The perils, on the other hand, means that programmers, especially students, need to be vigilant to not fall victim to them.

2.1 Advantages

After gaining some programming experience, there are many reasons to use a coding assistant. Implied in this statement is not advocating for the use of a coding assistant by newcomers to programming. Using a coding assistant requires writing clear and succinct prompts. That is, writing well-designed programs in a natural language is needed. Therefore, prompts require some experience with program design (including iterative refinement and divide-and-conquer). Furthermore, it is all too easy for beginning students, that do not know any better, to be allured by the idea of having a coding assistant generate programs for them with minimum or no effort. This, of course, sets beginners on a path to failure by hindering the development of design and communication skills.

For those with some software design experience, the use of coding assistants present the following advantages:

(1) Moving focus away from syntax to problem solving
(2) Ease to get started with problem solving
(3) Exposure to different ways to solve a problem
(4) Help with communicating solutions to problems clearly
(5) Help with refactoring code
(6) High-quality code explanations
(7) Circumventing cryptic error messages

Advantage (1) refers to producing syntactically correct programs with ease. The strict syntax of a programming language no longer needs to be an obstacle to learning a new programming language and no longer needs to slow down the problem solving process. Coding assistants have become very good at mundane syntax-based tasks, but it is important to remember that they may suggest syntactically incorrect code.

Advantage (2) refers to the ease provided for getting started with a solution. Even a poorly designed prompt is likely to suggest an answer. Through the process of iterative refinement, the prompt may be improved to have the coding assistant produce a better suggestion.

Advantage (3) refers to coding assistants having multiple suggestions. A programmer can judge different ways to solve a problem (even some the programmer has not considered) and select the one they feel is best suited for the software project under development. Multiple solutions, even when none is adopted, can provide ideas for improving the pursued design.

```
1  def double(n):
2      """
3      Signature: number -> number
4      Purpose: Double the given number.
5      Design Idea: Multiply the input number by 2.
6      """
7      return n * 2
8
9  def quadruple(n):
10     """
11     Signature: number -> number
12     Purpose: Quadruple the given number
13     Design Idea: Double given number twice and add
14     """
15     return double(n) + double(n)
16
17 print(quadruple(None))
```

Figure 1: Code to illustrate `AI` error message explanation.

Advantage (4) refers to a skill that is developed as programmers become better prompt engineers: the development of clear and succinct prompts. This skill hones the programmer's ability to communicate how a problem is solved and is important, because large software projects are developed by teams that need to understand the solution to a problem.

Advantage (5) refers to the help a coding assistant can provide to improve the solution to a problem. This is closely related to advantage 3. Ideas for code refactoring may present themselves by seeing different solutions to a problem. A coding assistant, however, may be able to analyze a program and directly suggest a different approach. For example, it may suggest using a `for`-loop instead of making recursive calls. In some cases, it may suggest the use of a different data structure to improve performance.

Advantage (6) refers to code explanations a coding assistant can provide. This is particularly useful when a programmer is not familiar with the syntax or the library used to solve a problem. Although coding assistants have become relatively good at explaining the code they suggest, programmers must remain vigilant. Coding assistants may explain a code suggestion that is buggy.

Advantage (7) refers to a coding assistant's ability to explain cryptic errors, which are known to be common [3, 4, 6, 7, 18, 19, 31]. Coding assistants often offer detailed descriptions for the source of an error. As experienced programmers know, an error is thrown where a bug is detected, which may or may not be where the error must be fixed. For instance, consider running the `Python` program displayed in Figure 1. It yields the following error:

```
File "quadruple.py", line 7, in double
    return n * 2
           ~~^~~
TypeError: unsupported operand type(s) for *:
'NoneType' and 'int'
```

Observe that the bug is the call to `quadruple` on line 17 (i.e., not where it is detected). Asking `GitHub Copilot` to explain the error yields:

```
Calling quadruple(None) will raise a TypeError
because double(None) attempts to multiply None
by an integer, which is not supported. Consider
adding input validation or handling None values
appropriately.
```

Observe that the coding assistant offers a much better explanation for the reason behind the bug than the error message.

2.2 Perils

There are substantial disadvantages associated with the use of coding assistants. Some of the more salient perils are anthropomorphism, attribution, development of questionable habits, and overreliance. Anthropomorphism refers to characterizing coding assistants and AI as having human traits [2, 14]. In part, this phenomena is due to AI companies and researchers referring to their software as having human capabilities. Commonly, this leads to a fundamental misunderstanding of AI-based software. Perhaps, the biggest problem on this front is overestimating what a coding assistant can do. It is important to understand that coding assistants do not have an understanding of the programming goals as the members of a software development team. Coding assistants try to predict the code requested based on a probabilistic distribution of the importance assigned to elements found in the prompt. Confusing the ability to predict the code expected with understanding easily leads to trusting solutions proposed by a coding assistant and failing to properly validate and review them.

Attribution refers to who deserves credit for the development of software. This problem emerges, because coding assistants are trained using a massive amount of coding examples. For example, the large language model behind GitHub Copilot is trained using public repositories hosted on GitHub [9]. Naturally, one can ask who deserves credit for a program generated using a coding assistant. The best way to illustrate the problem is by example. Consider the following prompt presented to GitHub Copilot:

```
def factorial(n):
    """
    Write a function to compute n!
    """
```

The function header written by the programmer provides the only context/guidance for the evaluation of the prompt that is inside triple quotes[1]. The suggested body is:

```
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Clearly, the author of the prompt can not claim that they have programmed a function to compute n!. This is not programming and claiming ownership of the above code is plagiarism, which is an act of academic dishonesty at all reputable universities. The problem arises because GitHub Copilot's training set contains such a function and GitHub Copilot simply returns it as a prediction for what the prompt's author needs. Clearly, whoever authored the function in GitHub Copilot's training set deserves the credit.

[1]In the Python world, triple quotes are called *docstrings*.

The use of coding assistants may also lead to the development of bad programming habits. It is important to realize that publicly available software is not scrutinized and is contributed by programmers with many different levels of experience. This means that coding assistants are likely trained with poorly designed code. For instance, many programmers lacking training in program design become obsessed with solving every problem using a single function or a single line of code. Consider the following prompt to sort a list of numbers:

```
def qs(lst):
 """
 Write quicksort using a single line of code.
 """
```

The function body suggested by GitHub Copilot is:

```
  return
    lst if len(lst) <= 1 \
        else qs([x for x in lst[1:] if x<lst[0]]) + \
             [lst[0]] + \
             qs([x for x in lst[1:] if x>=lst[0]])
```

The prompt is poorly designed and offers no guidance on how the program ought to be, for example, modularized. It is an example of an unsophisticated prompt that results in a poorly designed and unsophisticated function. It is important for programmers, especially students, to remember that programming is about much more than writing a program to solve a problem. An equally important goal is to communicate the solution to a problem to others. The above function does not clearly communicate how the problem is solved despite the use of list comprehensions (i.e., the idea of partitioning the numbers is not clearly communicated). In addition, the function above is very difficult to refine to make it more efficient (e.g., creating 3, not 2, partitions: numbers less than the pivot, numbers equal to the pivot, and numbers greater than the pivot).

Finally, another major danger of using coding assistants is becoming overly reliant on them. This can easily lead to not thinking about how a problem ought to be solved as is the case for `factorial` and `qs` above. This is dangerous, because it can lead a programmer to forfeit carefully thinking about code suggested by a coding assistant and integrating into a software project syntactically or semantically incorrect code that does not fulfill its purpose. Furthermore, problems associated with overreliance may be compounded to become even bigger problems. For instance, if a programmer overly relies on a coding assistant and enough bugs need to be fixed, then the programmer may mistakenly conclude that coding assistants mostly produce buggy code and lose confidence in their use. It is unfortunate if this occurs, because automatic feedback provided by a coding assistant has proven useful and now programmers are expected to be able to design prompts for their every day programming tasks.

3 Function and Prompt Engineering Using Structural Recursion

Given that prompts are programs, they need to be systematically designed and their suggestions need to be carefully checked. Otherwise, there is a significant risk that incorrect or dangerous code is

(1) Outline the representation of values and the computation.
(2) Write the function's header.
(3) Design and implement a prompt for the function.
(4) Implement the function's body
(5) Write the testing function's header.
(6) Design and implement a prompt for the testing function
(7) Implement the testing function's body.
(8) Run the tests and, if necessary, refine answers for previous steps.
(9) Optionally, refactor and run the tests again.

Figure 2: The design recipe for using a coding assistant.

integrated into the software package being developed. It is important to keep in mind that coding assistants are nondeterministic. That is, given a prompt they may provide a different answer each time the prompt is evaluated. For instance, a programmer may start working on a problem today using a prompt to help generate a function. The programmer may accept the suggestion made by the coding assistant and then take a break from programming. A week later the programmer may return to the program to discover that they not understand the suggested function. Promptly, they delete the function and let the prompt be evaluated again. This time a different function may or may not be suggested. The suggestion may change because a coding assistant is continually training itself and adjusting the importance it gives to words in a prompt.

To mitigate nondeterminism, it is important for prompts to be carefully and purposefully designed to provide the coding assistant with enough context to make a prediction that is unlikely to change. To do this systematically, design recipes are used to introduce students to how to engage a coding assistant in software development. A design recipe is a series of systematic steps, each with a concrete result, to scaffold the development of a program given a problem statement. Design recipes were pioneered by Felleisen et al. to instruct programming beginners [13] and later expanded by Morazán to teach a two-semester sequence for first-year Computer Science university students [22, 23] and to teach CS theory [24]. The work presented in this article is the first step to extending the use of design recipes in conjunction with coding assistants.

To teach students how to develop functions using a coding assistant, they are presented with two design recipes. The first is a modified design recipe from those found in the literature [13, 22, 23] for function development. The second is a design recipe for prompt development. These design recipes, naturally, are intertwined much like mutually recursive functions are intertwined. That is, both must be used in tandem during software development.

3.1 A Design Recipe for Functions Using a Coding Assistant

A design recipe for functions has two primary goals: the implementation of a solution to a problem and the development of unit tests. In `Python`, we may develop a function for each of these. Thus, every solution consists of two functions. For each of these functions, a prompt may be written to aid in its development. This means that a design recipe for function development needs to include prompt development steps.

(1) Write the signature.
(2) Write the purpose statement.
(3) Write the design idea.
(4) Analyze the suggestion and, if necessary, refine.

Figure 3: The design recipe for prompts.

Given that coding assistants make predictions based on elements already in the input sequence, the order of the steps is slightly changed from what appears in the literature [13, 22, 23]. The refined design recipe asks for the development of the function that solves the problem first. After this function is developed, the testing function is developed. Intuitively, this is justified by observing that a coding assistant may use the implementation of a function that solves a problem as context to help developing a testing function.

The design recipe for using a coding assistant is displayed in Figure 2. It has 9 steps and each step has a specific outcome that advances the solution to a problem. Step 1 requires the development of data definitions and function templates along with a signature, a purpose statement, and design idea. The data definitions and function templates are written at the top of the program's file inside triple double quotes. This places them as context for `GitHub Copilot` to use.

Steps 2–4 target the development of a function to solve a problem. Step 2 asks for the function header. Step 3 is an optional step that asks for the implementation of a prompt that describes to the coding assistant how to solve the problem. The prompt is developed using the design recipe for prompts discussed in Section 3.2. Step 4 asks for the development of the function's body. When using a prompt, this step requires examining the coding assistant's proposed body. If it conforms to the design idea outlined in Step 1 and if it appears correct, the proposed body may be accepted. If not, then the programmer may either correct the proposed body or refine the prompt to get a better suggestion. The importance of the design idea being consistent with the function's body to communicate how a problem is solved is emphasized to students.

Steps 5–7 target the development of the testing function. Step 5 asks for the header of the testing function. Step 6 asks for the development of a prompt for the testing function. Like Step 3, this step is optional. If performed, the prompt ought to provide guidance on how the function that solves the problem ought to be tested. It can, for example, outline the values used for testing. Step 7 asks for the testing function's body. If using a coding assistant, the programmer needs to inspect the proposed body to make sure that it is syntactically correct and that the tests are sensible.

Step 8 asks to run the tests and to redesign if any errors are thrown or any tests fail. Debugging is performed by revisiting the steps of the design recipe. Finally, Step 9, optionally, asks for code refactoring. This may be done with or without the use of a coding assistant.

3.2 A Design Recipe for Prompts

A prompt needs to guide the coding assistant to produce a solution using an outlined design idea. Naturally, the prompt needs to communicate the design idea in a concise and succinct manner. To accomplish this task, students are presented with the 4-step design

recipe displayed in Figure 3. Steps 1–3 ask for a signature, a purpose statement, and a design idea. The design idea can be elaborate and include, for example, instructions to use specific auxiliary functions to solve subproblems. In this manner, students are encouraged to follow a top-down type-based design strategy. Problems may be solved using a divide-and-conquer approach based on the type of data being processed. For instance, the rest of a list and the subtrees of a binary tree may be processed recursively.

As mentioned before, it is important for the programmer to remember that a coding assistant may produce an incorrect solution or a solution that is not well-designed. Therefore, Step 4 asks for the suggestions made by the coding assistant to be analyzed. This is the step where the design recipe for using a coding assistant and the design recipe for prompts explicitly intersect. This intersection exists in both design recipes, because using a prompt is optional. If the examination reveals that the prompt needs to be refined, then refinements are made by revisiting the previous steps of the design recipe prompts. If the examination reveals that the prompt does not need to be refined, then Step 4 of the design recipe for prompts and Step 4 of the design recipe for using a coding assistant are satisfied. It is highlighted to students that of the 3 steps in this design recipe, the one that may be refined the most is the design idea. This is where the most detail may be added to obtain a solution that is correct, clear, and adheres to the design idea.

3.3 A Warm-Up Example: Composing Functions

To provide the first illustration of the design recipes in action, we consider the development of a higher-order function for the composition of two one-input functions. This is a topic that all university students have studied in their previous (college or high school) Mathematics courses. Given two functions, `f(x)` and `g(x)`, a function, `f(g(x))`, that applies `f` to the result of applying `g` to an input `x` is needed. To be able to build such a function, `g`'s range must be `f`'s domain[2]. The returned function needs a single input, `x`, and its body composes `f` and `g`.

In tandem, the design recipes displayed in Figure 2 and in Figure 3 are used to implement the above design idea. The function header and the prompt may be written as follows:

```
def compose(f, g):
    """
    Signature: (A -> B) (C -> A) -> (C -> B)
    Purpose: Compose the given functions
    Design Idea: Return a function that applies f to the
                 result of g
    """
```

Observe that Steps 1–3 of the general design recipe for using a coding assistant and Steps 1–3 of the design recipe for prompts have been completed. The signature states that two one-input functions are expected such that the range of `g` is the domain for `f` and that the output is a function from the domain of `g` to the range of `f`. The purpose statement is clear and the design idea specifies how to apply the given functions. The suggestion returned by `GitHub Copilot` is:

```
    return lambda x: f(g(x))
```

[2]Technically, `g`'s range must only be a subset of `f`'s domain. We shall not delve into such nuances for the purpose of this example.

```
def test_compose():
    """
    Signature: () -> None
    Purpose: Test the compose function
    Design Idea:
      Define some simple functions and check their
      composition with several inputs
    """
```

(a) Prompt

```
    # Define some simple functions
    def add1(x):
        return x + 1
    def mul2(x):
        return x * 2
    # Test the compose function
    f = compose(mul2, add1)
    assert f(3) == 8  # (3 + 1) * 2
    g = compose(add1, mul2)
    assert g(3) == 7  # (3 * 2) + 1
```

(b) Suggested body

Figure 4: An attempt to develop `test_compose`.

The suggested body looks correct. It returns a one-input function that applies `g` to said input and then applies `f` to the result of `g`. Based on this analysis, the suggestion may be accepted and no refinements are required. Thus, Step 4 of the design recipe for prompts is satisfied as well as Step 4 of the design recipe for using coding assistants.

With the completion of Step 4 of the design recipe for using coding assistants, focus turns to completing Steps 5-7 to design and implement the testing function. This function takes no input and returns `None`. Its purpose is to test `compose`. This may be done by locally defining some simple functions and checking the results obtained by their composition using several inputs. The function header and prompt may be written as displayed in Figure 4a and the suggested body returned by `GitHub Copilot` is displayed in Figure 4b. The suggested body defines two simple functions and tests both of their possible compositions. The suggested tests are correct, but the solution is unsatisfactory. First, the composed functions are only tested with, 3, a single value. Second, only numeric functions are given to `compose`.

Clearly, the coding assistant needs a better prompt to produce a more thorough testing suite. An iterative refinement step is performed by reexamining the answers for the steps of the design recipe for prompts. The signature and purpose statement appear to be fairly clear and succinct. The design idea, however, could offer more guidance on how to test `compose`. It can specify that numeric and string functions need to be defined and that the composition of the numeric functions and the composition of the string functions need be tested with, say, 4 different values as displayed in Figure 5a. Observe that the prompt now offers a great deal more guidance about how testing ought to be done. Using this prompt, `GitHub`

```
def test_compose():
    """
    Signature: () -> None
    Purpose: Test the compose function
    Design Idea:
      Define some simple numeric and string functions.
      Check the composition of the numeric functions
      and of the string functions. Test the resulting
      composed functions with 4 different inputs.
    """
```

(a) Prompt

```
    # Define some simple numeric functions
    def add1(x):
        return x + 1
    def mul2(x):
        return x * 2
    # Test the composition of numeric functions
    f = compose(mul2, add1)
    assert f(3) == 8  # (3 + 1) * 2
    assert f(5) == 12  # (5 + 1) * 2
    assert f(0) == 2   # (0 + 1) * 2
    assert f(-1) == 0  # (-1 + 1) * 2
    # Define some simple string functions
    def append_exclamation(s):
        return s + "!"
    def make_uppercase(s):
        return s.upper()
    # Test the composition of string functions
    g = compose(make_uppercase, append_exclamation)
    assert g("hello") == "HELLO!"
    assert g("world") == "WORLD!"
    assert g("") == "!"
    assert g("test") == "TEST!"
```

(b) Suggested body

Figure 5: First refinement for `test_compose`.

```
def test_compose():
"""
Signature: () -> None
Purpose: Test the compose function
Design Idea:
  Define some simple numeric and string functions.
  Check the composition of the numeric functions
  and of the string functions. Test the resulting
  composed functions with 4 different inputs. Add
  a failed string for each  test: Test n failed,
  starting with n equal to 0.
"""
```

(a) Prompt

```
    # Define some simple numeric functions
    def add1(x):
        return x + 1
    def mul2(x):
        return x * 2
    f = compose(mul2, add1)
    assert f(3) == 8, "Test 0 failed"
    assert f(5) == 12, "Test 1 failed"
    assert f(0) == 2, "Test 2 failed"
    assert f(-1) == 0, "Test 3 failed"
    # Define some simple string functions
    def append_exclamation(s):
        return s + "!"
    def make_uppercase(s):
        return s.upper()
    g = compose(make_uppercase, append_exclamation)
    assert g("hello") == "HELLO!", "Test 4 failed"
    assert g("world") == "WORLD!", "Test 5 failed"
    assert g("") == "!", "Test 6 failed"
    assert g("test") == "TEST!", "Test 7 failed"
```

(b) Suggested body

Figure 6: Second refinement for `test_compose`.

`Copilot` suggests the body displayed in Figure 5b. Clearly, a better testing suite is generated. The same two numeric functions as before are generated and, in addition, two string functions are generated. The composition of the numeric functions and the composition of the string functions are each tested with 4 values.

No doubt progress has been made, but some details are missing. For instance, there are no failed test strings associated with any of the `assert`-statements. The answer to Step 3 of the design recipe for prompts is refined to guide `GitHub Copilot` to include failed test strings as displayed in Figure 6a. The prompt is explicitly asking for failed test strings and specifies the format for such strings. In other words, the prompt is offering a lot of guidance. We expect, therefore, a better implementation of the unit tests. Using this prompt, the suggested function body for the testing function is displayed in Figure 6b. The testing function is much improved and may be accepted. There are, of course, further refinements possible (e.g., testing both possible compositions of two given functions and testing numeric functions with both `int` and `float` values). In the interest of brevity, further refinements are not presented.

4 The Design Recipe in Action: Computing π

The section illustrates applying the design recipes to a problem that is not trivially solved. The chosen example is the computation of π. This example is chosen, because it provides the opportunity to drive home the importance of spending time on design and on how Mathematics can help simplify the job a software developer needs to do. In addition, all CS students are familiar with it and π now has a bit of a cult following in popular culture. Both of these characteristics make it interesting for students. As is well known, π represents the ratio of a circle's circumference to its diameter. Commonly, the value of π is stated to be $\frac{22}{7}$, 3.14, or 3.14159. It is important to note in class, however, that these values are, albeit poor, approximations for the value of an irrational number. To pique student interest, the use of π in different branches of human knowledge

and popular culture is highlighted. In Physics, it is used to describe electromagnetism, wave motion, and the operation of the universe. In Engineering, it is used to calculate trajectories. In Geometry and Mathematics, it is used to calculate the area and volume of different geometric shapes. It intrigues computer scientists because accurately computing its digits is a challenge. We also highlight that π has permeated popular culture with competitions for the most digits of π memorized. The current Guinness World Record is 70,000 digits recited by Rajveer Meena in 2015 [20]. In 1988 the physicist Larry Shaw founded Pi day [33]–an annual celebration on, 3/14, Albert Einstein's birthday.

After rallying the troops about π, attention turns to accurately computing an approximation. It is explained to students that this is important especially in high-precision computations. Based on the infinite series for π developed by Srinivasa Ramanujan [28], David Volfovich Chudnovsky (1947–) and Gregory Volfovich Chudnovsky (1952–) developed a fast algorithm to approximate π [11]. In April of 2025, this algorithm was used to compute 300 trillion digits of π [21]. Currently, this algorithm is used by modern mathematical software [34], which gets students excited.

At the heart of the Chudnovsky algorithm is the following equation:

$$\frac{1}{\pi} = 12\Sigma_{k=0}^{\infty} \frac{-1^k(6k)!(13591409+545140134k)}{(3k)!(k!)^3 640320^{3k+\frac{3}{2}}}$$

The next class goal is to implement the Chudnovsky algorithm and work through the steps of the design recipes using a coding assistant.

4.1 Chudnovsky Algorithm Version 0

In the form stated above, the Chudnovsky series is not friendly enough for implementation. For instance, students are explained that there is a radical in the denominator of the terms of the series which makes it difficult to perform accurate numeric calculations. The first step, therefore, is to use the rules of high school algebra to rationalize the fraction and eliminate the radical in the denominator of the terms of the series to obtain[3]:

$$\pi = \frac{1}{\frac{\sqrt{10005}}{4270934400}\Sigma_{k=0}^{\infty} \frac{-1^k(6k)!(13591409+545140134k)}{(3k)!(k!)^3 640320^{3k}}}$$

Without a radical in a denominator, the design continues by introducing the following notation:

$$\texttt{c} = \frac{\sqrt{10005}}{4270934400} \qquad \texttt{f(k)} = \frac{-1^k(6k)!(13591409+545140134k)}{(3k)!(k!)^3 640320^{3k}}$$

With these definitions in place, the equation above may be more simply written as follows:

$$\pi = \frac{1}{c*\Sigma_{k=0}^{\infty} f(k)}$$

In this form, a design idea more easily emerges. We ought to define the variables above, use an auxiliary function to compute the value of the series, and return the approximation for π. All numbers need to be high-precision decimals. It is, of course, not possible to add an infinite number of terms. To make the summation finite, the user of the function will need to provide the number of terms to add.

4.1.1 The Function to Compute π. Following the steps of the design recipe for using a coding assistant, the above design idea enables the writing of the function header and prompt:

```
def computePi(iterations):
  """
  Signature: natnum -> decimal.Decimal
  Purpose: Approximate the value of pi
  Design Idea:
   Represent numbers using decimal.Decimal for
     high-precision
   Use the following variables and function:
    c = sqrt(10005)/4270934400
    f = (-1**k * fact(6k) * (13591409+545140134k))/
        (fact(3k) * fact(k)**3 * 640320**3k)
    ChudnovskySum(f,iterations) to compute series
   return 1/(c * series)
  """
```

The suggested body returned by `GitHub Copilot` is:

```
1  c=decimal.Decimal(math.sqrt(10005))/ \
2    decimal.Decimal(4270934400)
3  f=lambda k: \
4     (decimal.Decimal((-1)**k)*fact(6*k) * \
5     (decimal.Decimal(13591409)+ \
6      decimal.Decimal(545140134)*k)) \
7     / \
8     (fact(3*k)*(fact(k)**3)* \
9      (decimal.Decimal(640320)**(3*k)))
10 series=ChudnovskySum(f, iterations)
11 return 1/(c * series)
```

Although it may be a bit difficult to read, the suggested body implements the design idea outlined in the prompt. The numerator for the term computed by `f` is on lines 4–6 and the denominator is on lines 8–9. Given that the proposed body is clear and appears correct, we may cautiously accept it.

The next task is to develop a testing function for `computePi`. Given that high precision decimals are tested, the prompt ought to define a margin of error (or tolerance value). To keep the discussion simple, we arbitrarily choose 10^{-15}. The absolute value of the difference between the approximation `computePi` returns and a known approximation of π, say with 50 decimals, needs to be less than the tolerance value[4]. Finally, the function may be tested using multiples of 50 in `[0..1000]`. Using this design idea, the function header and prompt may be written as follows:

```
def test_computePi():
 """
 Signature: -> None
 Purpose: Test computePi
 Design Idea:
  Define Pi as a Decimal('3.14159265358979323...')
  define a tolerance equal to 10**-15
  Use iterations = multiples of 50 in [0, 1000]
  Include fail test strings "Test n failed"
    starting with n=0,1,2,3,...
  """
```

The body suggested by `GitHub Copilot` is:

[3]The series' factor, 12, in the equation above is absorbed into the new factor.

[4]Such a value is easily obtained through an internet search.

```
Pi = decimal.Decimal('3.14159265358979323...')
tolerance = decimal.Decimal(10)**-15
for n in range(1,1000,50):
    computed_pi = computePi(n)
    assert abs(computed_pi - Pi)<tolerance,\
           f"Test {n} failed"
```

The failed test strings are not numbered as outlined in the prompt. Instead, they include the argument provided to `computePi`. This different design is based on a good idea. When tests are executed inside a loop, distinguish failing tests using the argument provided to `computed_pi`. Therefore, it may be cautiously accepted as long as the prompt is refined to reflect the accepted design.

A word of caution is issued at this time. The function is designed without knowing how many iterations are needed to have an approximation that is within the margin of error defined by `tolerance`. They must be prepared, therefore, for testing to fail for values that are not large enough to generate a good approximation.

4.1.2 The Function to Compute the Chudnovsky Series. The higher-order function to compute the series takes as input a function for the terms of the series and a natural number, `iterations`, for the number of terms. The goal is to compute the series using the natural numbers in `[0..iterations]`. This is done using structural recursion on, `iterations`, a natural number. To guide the coding assistant, the data definition and template for a function on a natural number are written as global prompts (not shown). The function header and the prompt are written as follows:

```
def ChudnovskySum(f, iterations):
 """
 Signature: (natnum -> decimal.Decimal) natnum ->
            decimal.Decimal
 Purpose: Compute the Chudnovsky series from 0
          to the given number of iterations
 Design Idea:
   Represent numbers using decimal.Decimal
   Add the terms of the series produced by f
   Use structural recursion on iterations
   Structure the code using the template for a
     function on a natnum
```

The suggested body generated by `GitHub Copilot` is:

```
if iterations == 0:
   return(decimal.Decimal(0),decimal.Decimal(0))
else:
   ak=f(iterations)
   asum,bsum=ChudnovskySum(f,iterations-1)
   return (asum+ak,bsum+(ak*iterations))
```

Clearly, the suggested code does not implement the specified design. The suggestion is likely based on a function in `GitHub Copilot`'s training set that it associates with one or more elements in the prompt (perhaps, Chudnovsky or ChudnovskySum). Therefore, Step 4 of the design recipe for prompts suggests a refinement is needed.

To provide more guidance to `GitHub Copilot`, the design idea can specify the value to return for the base case (i.e., `iterations`=0): 13591409. The refined design idea for the prompt is:

```
 """
 Design Idea:
       Represent numbers using decimal.Decimal
       Add the terms of the series produced by f
         for i from 1 to iterations
       Use structural recursion on iterations
       Structure the code using the template for a
         function on a natnum
       When iterations is 0, return 13591409
         Otherwise, add the next term to the sum
         of the rest of the terms
 """
```

The refined prompt suggests the following function body:

```
if iterations == 0:
    return decimal.Decimal(13591409)
else:
    k = decimal.Decimal(iterations)
    return f(k)+ChudnovskySum(f,iterations-1)
```

The extra guidance provided has successfully led the coding assistant to suggest a body that implements the design idea. Note that `iterations` is converted to a high-precision decimal before providing it as input to `f`. It is unclear why `GitHub Copilot` has chosen to do this, but the suggested body is cautiously accepted given that it appears to be correct. In the interest of brevity, the design of the testing function and of the factorial function are not presented.

4.1.3 Debugging the Initial Design. Upon running the tests, the following failed test report is generated:

```
assert abs(computed_pi-Pi)<tolerance, f"Test n failed"
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: Test 0 failed
```

This test failure informs the programmer that performing 0 iterations of the Chudnovsky sum produces an approximation of π that is not within the defined margin of error. To resolve this issue, a student may be tempted to simply delete 0 as low-end of the tested range. A better approach is to determine through experimentation the smallest number of iterations needed to produce a good approximation. Luckily, the search is short given that 1 iteration suffices to produce an good approximation. In the test range, 0 is substituted with 1 and the prompt is refined to reflect this new design.

Upon running the tests again, the following error is thrown:

```
for k in range(1, n + 1):
         ^^^^^^^^^^^^^^^
TypeError: 'decimal.Decimal' object cannot be
interpreted as an integer
```

This bug is detected inside of `fact`, which does not need to be shown to understand the error message. The error message indicates that `n` is a high-precision decimal, which is invalid input to construct a range. Students are typically tempted to immediately edit the program where the bug is detected and this is an opportunity to emphasize debugging by revisiting the steps of the design recipe. The given input, `n`, violates `fact`'s signature: `natnum -> decimal.Decimal`. A high-precision decimal is provided as input instead of a natural number. This suggests that some call is made

to `fact` with the wrong type of argument. This function is only called from the body of the `lambda`-expression for `f` in `computePi`. Observe that all calls to `fact` depend on the input, `k`, given to the function obtained by evaluating the `lambda`-expression. The only place where this function is used is in the body of `ChudnovskySum`. This means the programmer ought to revisit Step 4 of the design recipe for prompts and reanalyze the suggested body. Observe how the function is used in the `else` clause:

```
k = decimal.Decimal(iterations)
return f(k) + ChudnovskySum(f,iterations-1)
```

The function, `f`, is given as input a high-precision decimal instead of a natural number. Thus, Step 4 of the design recipe for using a coding assistant has not been successfully completed. The body of `ChudnovskySum` is not properly implemented. It is emphasized to students that the bug is not located where the bug was detected. The solution is this case appears to be fairly straightforward. Let us not convert `iterations` to a high-precision decimal:

```
return f(iterations)+ChudnovskySum(f,iterations-1)
```

Upon making this change and running the tests again, no tests fail and no errors are thrown. We have a program that produces an approximation for π.

4.2 Chudnovsky Algorithm Version 1

Experimenting with the implementation developed is encouraged. Such experimentation, may lead some students to approximate π using, say 950, iterations. Running this computation takes a noticeable, albeit small, amount of time to compute. Students are advised that such delays may become magnified with larger inputs and become a problem. To avoid this potential problem, revisiting Step 1 of the design recipe for using a coding assistant (i.e., problem analysis) is advised to determine if there are any optimizations that may be performed.

4.2.1 Revisiting Step 1 of the Design Recipe for Using a Coding Assistant. The only place where simplifications may be possible in the equation for π is in the computation of the series:

$$\Sigma_{k=0}^{\infty} \frac{-1^k (6k)!(13591409+545140134k)}{(3k)!(k!)^3 640320^{3k}}$$

Students are asked: Where can there be a time bottleneck? There are two salient characteristics that ought to be explored. The first is that an accumulator is not used to compute the sum. To compute the next term of the series, `k` must be plugged into a fairly complex equation. It is explained that it may be faster to implement a recurrence relation. The second is that a factorial is computed three different times. Therefore, reducing the number of times a factorial is computed should also be explored.

A significant number of students immediately observe that:

```
(6k)! = 6k*(6k-1)*(6k-2)*...*(3k+1)*(3k)!
```

This means that the above series may be simplified to:

$$\Sigma_{k=0}^{\infty} \frac{-1^k * 6k*(6k-1)*(6k-2)*...*(3k+1)*(13591409+545140134k)}{(k!)^3 640320^{3k}}$$

This is progress. Instead of computing 3 different factorials, only a single factorial needs to be computed for each term of the series.

Class discussion leads to the observation that every term of the series is adding two subterms. Thus, the summation may be written as follows:

$$13591409 * \Sigma_{k=0}^{\infty} \frac{-1^k * 6k*(6k-1)*(6k-2)*...*(3k+1)}{(k!)^3 640320^{3k}} + 545140134 * \Sigma_{k=0}^{\infty} \frac{-1^k * 6k*(6k-1)*(6k-2)*...*(3k+1)*k}{(k!)^3 640320^{3k}}$$

Students immediately observe that this does not look like progress, because this form suggests that two series need to be computed. In addition, two factorials need to be computed (one for each series). It is pointed out to them that each series is now simpler and may lead to further optimizations.

Each of the series can be named to yield:

$$\mathrm{A} = \Sigma_{k=0}^{\infty} \frac{-1^k * 6k*(6k-1)*(6k-2)*...*(3k+1)}{(k!)^3 640320^{3k}}$$

$$\mathrm{B} = \Sigma_{k=0}^{\infty} \frac{-1^k * 6k*(6k-1)*(6k-2)*...*(3k+1)k}{(k!)^3 640320^{3k}}$$

$$\pi = \frac{1}{\frac{\sqrt{10005}}{4270934400}(13591409*A+545140134*B)}$$

Removing the fraction from the denominator and simplifying yields:

$$\pi = \frac{426880\sqrt{10005}}{13591409*A+545140134*B}$$

Students can observe that B's k^{th} term is the product of A's k^{th} term and k. Thus, B may be rewritten as follows:

$$\mathrm{B} = \Sigma_{k=0}^{\infty} A_k * k$$

It is explained to students that this means that terms of the B series can be computed at the same time as the terms of the A series.

It remains to be seen if the computation for A can be simplified. As mentioned, a recurrence relation for A desirable: compute the next term of the A series, A_k, using previously computed values. To do so, the following equation needs to be solved for x:

$$A_{k-1} * x = A_k$$

This informs us that:

$$x = \frac{(-1^k)6k(6k-1)(6k-2)...(3k+1) * ((k-1)!)^3 640320^{3(k-1)}}{(k!)^3 640320^{3k}(-1^{k-1})6(k-1)(6(k-1)-1)(6(k-1)-2)...(3(k-1)+1)}$$

The formula for x is quite complex and would be tedious and error-prone to program. It also requires the computation of several factorial values and arbitrarily long products. This is an opportunity for CS students to appreciate Mathematics as a powerful ally that can help simplify their work. Using high school algebra to perform simplifications yields:

$$x = -\frac{24*(6k-1)*(2k-1)*(6k-5)}{k^3 640320^3}$$

Thus, students have discovered a simpler way to compute the terms of the A series:

$$A_k = -\frac{24*(6k-1)*(2k-1)*(6k-5)}{k^3 640320^3} * A_{k-1}$$

It is highlighted to students that this is a remarkable result. They can observe that in the recurrence relation there is no need to compute factorials and the terms of the series are no longer complex as in the original formulation. Such a result illustrates a powerful reason for problem solvers and programmers to study Mathematics that students can appreciate. It also highlights that time invested developing a better design idea is time well invested.

4.2.2 Revisiting `computePi`. As before, this function takes as input a natural number and returns a high precision decimal. The purpose is to approximate the value of π. According to the formula developed, a constant needs to be divided by the sum of a multiple of the approximation for A and a multiple of the approximation

```
def computePi(iterations):
 """
 Signature: natnum -> decimal.Decimal
 Purpose: Approximate the value of pi
 Design Idea:
  Represent numbers using decimal.Decimal
  Use the following variables and function:
   numerator = 426880*sqrt(10005)
   afactor = 13591409
   bfactor = 545140134
   f=lambda k:(24*(6*k-5)*(2*k-1)*(6*k-1))/
              (640320**3 * k**3)
   AB = ChudnovskySeries(f,1,0,iterations) is the
    tuple for the approximations of A and B of
    the Chudnovsky series
 return numerator/(afactor*AB[0]+bfactor*AB[1])
 Do not locally define ChudnovskySeries
 """
```

(a) Prompt

```
numerator = decimal.Decimal(426880) * \
            decimal.Decimal(math.sqrt(10005))
afactor = decimal.Decimal(13591409)
bfactor = decimal.Decimal(545140134)
f = lambda k: (24*(6*k-5)*(2*k-1)*(6*k-1)) \
              (decimal.Decimal(640320**3)*\
               decimal.Decimal(k**3))
AB = ChudnovskySeries(f, \
                      decimal.Decimal(0), \
                      decimal.Decimal(0), \
                      iterations)
return numerator/(afactor*AB[0]+bfactor*AB[1])
```

(b) Suggested body

Figure 7: Prompt and suggested body for `computePi`.

for B. In the prompt's design idea, these factors may defined as variables. In addition, a function, f, may defined to compute the factor by which A_{k-1} is multiplied to obtain the value of A_k. The A and B series are computed simultaneously by, ChudnovskySeries, an auxiliary function that returns a tuple containing the approximation for each series. This auxiliary function processes iterations using structural recursion and needs as input f, the initial values for the accumulated approximations of the two series (intuitively, both initial values are 0), and iterations. Finally, the returned value is obtained by dividing the numerator variable and the sum of the multiples of the approximations for A and B.

Based on the above design idea, the prompt is written as displayed in Figure 7a. Observe that the prompt is written using the results obtained by developing a design idea that simplifies the computation of π. It is more detailed and longer than the prompt used for version 0 and the expectation is that it will produce the desired program. The prompt also asks for the auxiliary function to not be locally defined. This is simply an attempt to keep each function short and make both easier to read. Once an implementation is developed, students are free to locally define the auxiliary function. The body suggested by GitHub Copilot is displayed in Figure 7b. The suggested body implements the outlined design idea and, therefore, it is accepted.

4.2.3 Computing the A *and* B *Series.* The higher-order auxiliary function to approximate the values of the A and B series takes as input the function, f, to compute the factor for A's k^{th} term, a high precision decimal accumulator for each series, asum and bsum, and a natural number, k, for the number of iterations to perform. It returns a tuple containing the approximation for both series. The terms for both series are accumulated using structural recursion on the given natural number. If k is 0 then the tuple (asum,bsum) is returned. Otherwise, compute, f(k), A's k^{th} factor, recursively process k-1 to compute the tuple containing the approximations for the series using k-1 terms, and, according to the derived formulas above, return the tuple containing asum+f(k)*A_{k-1} and bsum+B_{k-1}ak. The accumulator invariant reflects that each tuple constructed contains the approximations obtained for each series by summing up the first k-1 terms: (asum, bsum)=$(\Sigma_{i=0}^{k-1}A_i, \Sigma_{i=0}^{k-1}B_i)$.

Based on the above design idea, the prompt is written as displayed in Figure 8a and the body suggested by GitHub Copilot is displayed in Figure 8b. Clearly, the proposed body is making the recursive call with the wrong values[5]. The coding assistant needs more guidance to generate the correct recursive call.

The design idea is refined as displayed in Figure 9a and the body suggested by the coding assistant is displayed in Figure 9b. The proposed body implements the design idea. However, the accumulator invariant is not used to document the code. This is important to communicate how the problem is solved and may be done by manually inserting a comment containing the invariant after setting prevtuple and, similarly, indicating the value of the accumulators in the base case. There is further motivation for insisting on such careful documentation. Students are being prepared to write Hoare Logic [16, 17] assertions when they study imperative programming.

4.2.4 Debugging the New Design. Following the steps of the design recipe for using a coding assistant, the next task is to develop a testing function. Observe that the development is a refactoring exercise for computePi. Therefore, the same testing function developed for version 0 is used. We include, however, 0 in the range used for testing. It is explained to students that 0 iterations using this improved implementation may yield a good approximation for π.

Running the tests throws the following error:

```
return numerator/(afactor * AB[0] + bfactor * AB[1])
   ~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
decimal.DivisionByZero:
  [<class 'decimal.DivisionByZero'>]
```

Alas! Despite all the design efforts, there is clearly a bug. The division by 0 bug means that both results for the A and B approximations are 0. As suggested by the design recipe, debugging is performed by refining the answers to the previous steps of the design recipe. The first step is to examine the design to compute A and B approximations.

[5]This is what many in the literature call a *hallucination*. This article purposely refrains from using this term in order not to foster anthropomorphism.

```
def ChudnovskySeries(f, asum, bsum, k):
 """
 Signature: (natnum -> decimal.Decimal)
             decimal.Decimal
             decimal.Decimal
             natnum
            -> (decimal.Decimal, decimal.Decimal)
 Purpose: Returns the approximations for the A
          and B series of the Chudnovsky
          algorithm to compute pi
 Design Idea:
  Represent numbers using decimal.Decimal
  Add the terms of the series produced by f
  Use structural recursion on k to compute sum
  Structure the code using the template for a
    function on a natnum
  Accumulator Invariants:
        (asum,bsum)=(Σ_{i=0}^{k-1} A_i, Σ_{i=0}^{k-1} B_i)
  if k is 0 return (asum, bsum)
  else set ak=f(k)
       set prevtuple=recursively process k-1
       return (asum+(ak*prevtuple[0]),
               bsum+(ak*prevtuple[1]))
 """
```

(a) Prompt

```
constant=decimal.Decimal(426880)*\
         decimal.Decimal(math.sqrt(10005))
if k == 0:
  return (asum, bsum)
else:
  ak = f(k)
  prevtuple=\
   ChudnovskySeries(f,asum+ak,bsum+(ak*k),k-1)
  return (prevtuple[0],prevtuple[1])
```

(b) Suggested body

Figure 8: `ChudnovskySeries`: first prompt and suggested body.

In the design idea for `computePi` (see Section 4.2.2), 0 is chosen as the initial value for both approximations. Could this be the source of the bug? Recall that:

$$A_k = f(k) * A_{k-1}$$

We examine the computation performed by `ChudnovskySeries` using a concrete value, 4, for `iterations` and 0 for the initial value of both series:

```
ChudnovskySeries(f,0,0,4)
  = f(4)*ChudnovskySeries(f,0,0,3)
  = f(4)*f(3)*ChudnovskySeries(f,0,0,2)
  = f(4)*f(3)*f(2)*ChudnovskySeries(f,0,0,1)
  = f(4)*f(3)*f(2)*f(1)*ChudnovskySeries(f,0,0,0)
  = f(4)*f(3)*f(2)*f(1)*0
```

Clearly, the design idea was too casual about the initial value for `asum`. Given the computation trace above, the initial value for `asum` is changed to 1. Upon making this change all tests pass and no errors are thrown. Student notice that running the tests is significantly faster now, thus, driving home the point that time invested in developing a design idea is time well invested.

```
"""
Design Idea:
  Represent numbers using decimal.Decimal
  Add the terms of the series produced by f
  Use structural recursion on iterations to
    compute the sum
  Structure the code using the template for a
    function on a natnum
  Accumulator Invariants:
        (asum,bsum)=(Σ_{i=0}^{k-1} A_i, Σ_{i=0}^{k-1} B_i)
  if k is 0 return (asum, bsum)
  else
   set ak to f(k)
   set prevtuple to recursively processing f,
     asum, bsum, and k-1
   return (asum+(ak*prevtuple[0]),
           bsum+(ak*prevtuple[1]))
"""
```

(a) Refined design idea for prompt

```
if k == 0:
   return (asum, bsum)
else:
   ak=f(k)
   prevtuple = ChudnovskySeries(f,asum,bsum,k-1)
   return (asum+(ak*prevtuple[0]), \
           bsum+(prevtuple[1]*ak))
```

(b) Suggested body

Figure 9: Refinement for `ChudnovskySeries`.

The final step of the design recipe for using a coding assistant asks for the program to be optimized. For brevity, we do not illustrate this step, but it is not difficult. In essence, the coding assistant refactors to use a `for`-loop instead of recursive calls to circumvent the artificial limit `Python` imposes on the number of recursive calls.

In closing, the program developed to approximate π still requires further refinements to improve its accuracy. For example, the program does not compute a high precision decimal for $\sqrt{10005}$. This introduces an error into the performed calculation. Unfortunately, `Python`'s `math.sqrt` does not accept high precision decimals as input. Thus, refining the program to approximate π requires writing a function to compute the square root of a number represented using a high precision decimal. Designing such a function means using, for example, Newton's method, which is based on generative recursion and beyond the scope of this article. In the classroom, students are advised that the next refinement shall occur while learning to design functions based on generative recursion.

5 Related Work

At the time of writing, there are very few textbooks that introduce students to AI-based programming. None focus on program design using a coding assistant.

Porter's and Zingaro's textbook, *Learn AI-Assisted Python Programming*, is an introduction to programming tailored for Python using GitHub Copilot. It introduces its readers to the concept of a prompt and the need to be precise. It presents what resembles steps for developing functions, but these fall short of being a design recipe by failing to be sufficiently systematic. For example, steps such as *Write a prompt that describes the function as clearly as possible* and *If the code is incorrect edit the prompt* provide no guidance as to how to complete such tasks. In contrast, the work presented in this article is specific about how to write prompts and how to perform debugging using the steps of the design recipe. The authors of this textbook advise to *keep functions short—something like 12–20 lines—to give us the best chance of getting good code from Copilot.* The natural question that arises is *how does a student know how long a function will be in advance?* In contrast, the work presented describes systematic steps with concrete answers that do not require the proverbial crystal ball to complete. The authors acknowledge the value of top-down design, but fail to provide concrete guidance of how to divide-and-conquer problems. Instead, they state that *knowing when to stop breaking a task into smaller subtasks is more art than science.* In contrast, the work presented uses a type-driven methodology to guide the decomposition of problems. Finally, debugging is focused on using a debugger and is also referred to as *an art more than a science.* In contrast, the work presented reduces the search space for resolving a bug by casting debugging in terms of revisiting the steps of the design recipe.

The book *Murach's AI-Assisted Programming with Copilot* walks its readers through the interface offered by CoPilot, which services a multitude of programming languages. The use of the interface is described without any regard for program design. It describes how to use features with vague advice about being specific, providing context, and specifying the output. Much of the book reads like the old programming books from yesteryears that defined programming as being equivalent to learning the syntax of a programming language. This book's final chapter does introduce the topic of unit testing. In this chapter's Python section unit testing is introduced in tandem with the pytest extension, which allows tests to be generated in a different file. In contrast, the work presented in this article outlines test development within the same file as the program under development. This is done for pedagogic reasons. It tightly-couples function and test development in the mind of students. The development of tests is not treated as an after thought once a problem is solved. It is an integral part of problem solving using programming. This is not to say that tests should not be placed in a separate file. There are valid and important reasons to do so, like not hindering performance every time a program is evaluated, but this is an optimization that should be pursued after successfully implementing a design.

Taulli's *AI-Assisted Programming: Better Planning, Coding, Testing, and Deployment* discusses the advantages and the pitfalls of using a coding assistant as done in this article. As the work presented, this book also discusses the implications of nondeterminism when using a coding assistant and the need for prompts to be precise. It also discusses the need for prompts to be provided with context, clear and succinct instructions, formatting requirements (as done for prompts to generate unit tests in this article), and the idea of how to divide and conquer a problem/prompt. In contrast to the work presented in this article, it does not outline a systematic methodology for AI-assisted software developing and claims that prompt engineering is both a science and an art.

There have also been efforts describing the delivery of courses using a coding assistant (for an extensive overview see [27]). For example, using Porter's and Zingaro's textbook, Vadaparty et al. describe how they integrate the use of a coding assistant into a course for beginners [32]. Their course is aimed at a general student population, not solely for CS majors. Among their goals is to encourage students to use coding assistants and prepare them to enter the workforce of the future. The course focuses on teaching variables, functions, conditionals, loops, strings, lists, and dictionaries in Python. As the work presented in this article, it recognizes the importance of top-down design and problem decomposition. In contrast, it does use a systematic type-driven design methodology. Furthermore, there is no recognition that prompts are programs that need to be systematically designed.

6 Concluding Remarks

This article presents a first effort to integrate systematic program design with AI-assisted programming. The work presented focuses on development using structural recursion and is only the first step in teaching students to design programs with a coding assistant. The presented approach uses the steps of the design recipe for functions to develop precise and succinct prompts to guide a coding assistant. The result is two-fold. On the one hand, it teaches students how to systematically use coding assistants to design and implement software. On the other hand, prompts also serve as documentation for programs to make them easier for others to understand. The development methodology using a coding assistant put forth clearly illustrates that the process of software development benefits from systematic design and squarely makes it a science and not an art as described by other authors.

Future work includes defining a systematic approach (i.e., design recipes) to develop functions using generative and accumulative recursion using a coding assistant. Generative recursion requires insights for problem decomposition beyond the structure of a type. This needs to be reflected in the design of prompts. Accumulative recursion requires the use of accumulators and accumulator invariants to describe them, which also needs to be reflected in prompts. Both of these forms of recursion, therefore, require new sets of skills that students need to be taught. Future work also includes extending the presented prompt engineering methodology to imperative and object-oriented programming. These programming approaches present their own unique set of challenges. Prompts need to capture, for example, how to safely sequence mutations during in-place operations. Finally, future work also includes human-factors study to determine student impressions and the how effective students are at using the proposed design recipes.

References

[1] 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan. 2024), 56–67. doi:10.1145/3624720

[2] Amani Alabed, Ana Javornik, and Diana Gregory-Smith. 2022. AI Anthropomorphism and its Effect on Users' Self-Congruence and Self–AI Integration: A Theoretical Framework and Research Agenda. *Technological Forecasting and Social Change* 182 (2022), 121786. doi:10.1016/j.techfore.2022.121786

[3] Sanaa M. Algaraibeh. 2022. Techniques for Enhancing Compiler Error Messages. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2* (Lugano and Virtual Event, Switzerland) *(ICER '22)*. Association for Computing Machinery, New York, NY, USA, 1–2. doi:10.1145/3501709.3544292

[4] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE, New York City, USA, 575–585. doi:10.1109/ICSE.2017.59

[5] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 500–506. doi:10.1145/3545945.3569759

[6] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. doi:10.1145/3344429.3372508

[7] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective Compiler Error Message Enhancement for Novice Programming Students. *Computer Science Education* 26, 2-3 (2016), 148–175. doi:10.1080/08993408.2016.1225464

[8] Idan A. Blank. 2023. What are Large Language Models Supposed to Model? *Trends in Cognitive Sciences* 27, 11 (2023), 987–989. doi:10.1016/j.tics.2023.08.006

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[10] D.V. Chudnovsky and G.V. Chudnovsky. 2000. Approximations and complex multiplication according to Ramanujan. In *Number Theory: Proceedings of the International Conference on Number Theory in Memory of Serge Lang (2000)*. Springer New York, 65–102. doi:10.1007/978-1-4757-3240-5_63

[11] David V. Chudnovsky and Gregory V. Chudnovsky. 1987. *Approximations and Complex Multiplication According to Ramanujan*. IBM Thomas J. Watson Research Division, Yorktown Heights, NY. Research report.

[12] Google Cloud. 2025. Prompt Engineering: Overview and Guide. https://shorturl.at/ZX8ki. Last Accessed: 8/2025.

[13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (Second ed.). MIT Press, Cambridge, MA, USA.

[14] Ben Garside. 2023. How anthropomorphism hinders AI education. https://www.raspberrypi.org/blog/ai-education-anthropomorphism/. Last Accessed: 8/2025.

[15] Ran He, Jie Cao, and Tieniu Tan. 2025. Generative Artificial Intelligence: A Historical Perspective. *National Science Review* 12, 5 (02 2025), nwaf050. doi:10.1093/nsr/nwaf050

[16] C.A.R. Hoare and H. Jifeng. 1998. *Unifying Theories of Programming*. Prentice Hall.

[17] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

[18] Matthew C. Jadud. 2006. *An Exploration of Novice Compilation Behaviour in BlueJ*. Ph.D. Thesis. University of Kent at Canterbury, Canterbury, UK.

[19] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using Large Language Models to Enhance Programming Error Messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 563–569. doi:10.1145/3545945.3569770

[20] Guinness World Records Limited. 2015. Most Decimal Places of Pi Memorized. https://www.guinnessworldrecords.com/world-records/most-pi-places-memorised. Last Accessed: 8/2025.

[21] Guinness World Records Limited. 2025. Most Accurate Value of pi. https://web.archive.org/web/20250508155903/https://www.guinnessworldrecords.com/world-records/66179-most-accurate-value-of-pi. Last Accessed: 8/2025.

[22] Marco T. Morazán. 2022. *Animated Problem Solving - An Introduction to Program Design Using Video Game Development*. Springer, Cham, Switzerland. doi:10.1007/978-3-030-85091-3

[23] Marco T. Morazán. 2022. *Animated Program Design - Intermediate Program Design Using Video Game Development*. Springer, Cham, Switzerland. doi:10.1007/978-3-031-04317-8

[24] Marco T. Morazán. 2024. *Programming-Based Formal Languages and Automata Theory - Design, Implement, Validate, and Prove*. Springer, Cham, Switzerland. doi:10.1007/978-3-031-43973-5

[25] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2025. A Comprehensive Overview of Large Language Models. *ACM Trans. Intell. Syst. Technol.* (June 2025). doi:10.1145/3744746

[26] Leo Porter and Daniel Zingaro. 2024. *Learn AI-Assisted Python Programming* (second ed.). Manning Publications, Shelter Island, NY.

[27] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots Are Here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education* (Turku, Finland) *(ITiCSE-WGR '23)*. Association for Computing Machinery, New York, NY, USA, 108–159. doi:10.1145/3623762.3633499

[28] Srinivasa Ramanujan. 1914. Modular equations and approximations to π. *Quarterly Journal of Mathematics* 45 (1914), 350–372.

[29] Sengar, Sandeep Singh and Hasan, Affan Bin and Sanjay Kumar and Fiona Carroll. 2024. Generative Artificial Intelligence: A Systematic Review and Applications. *Multimedia Tools and Applications* 84, 21 (aug 2024), 23661–23700. doi:10.1007/s11042-024-20016-1 Publisher Copyright: © Crown 2024..

[30] MIT Sloan Teaching & Learning Technologies. 2025. Effective Prompts for AI: The Essentials. https://shorturl.at/RxM8S. Last Accessed: 8/2025.

[31] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Adv. in Hum.-Comp. Int.* 2010, Article 3 (jan 2010), 26 pages. doi:10.1155/2010/602570

[32] Annapurna Vadaparty, Daniel Zingaro, David H. Smith IV, Mounika Padala, Christine Alvarado, Jamie Gorson Benario, and Leo Porter. 2024. CS1-LLM: Integrating LLMs into CS1 Instruction. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 297–303. doi:10.1145/3649217.3653584

[33] Wikipedia. 2025. Pi Day. https://en.wikipedia.org/wiki/Pi_Day. Last Accessed: 8/2025.

[34] Alexander J. Yee. 2025. y-cruncher - A Multi-Threaded Pi-Program. https://www.numberworld.org/y-cruncher/. Last Accessed: 8/2025.

Heuristics-based Type Error Diagnosis for Haskell

The case of type families (draft paper)

Niels Kwadijk
Dept. of Information and Computing Sciences
Utrecht University
The Netherlands
niels.kwadijk@gmail.com

Jurriaan Hage
Dept. of Computer Science
Heriot-Watt University
United Kingdom
Dept. of Information and Computing Sciences
Utrecht University
The Netherlands
J.Hage@hw.ac.uk

Abstract

Helium is a Haskell compiler designed to provide programmer friendly type error messages. It employs specially designed heuristics that work on a type graph representation of the type inference process.

In order to support (all flavours of) type families in Helium, we extend the type graphs of Helium in order to design and implement heuristics that can diagnose type errors that might be caused by mistakes and misunderstandings involving type families. To increase programmer understanding, we also include type family execution traces as part of the type error diagnostics.

Keywords: type error diagnosis, generalized algebraic data types, type graphs, Haskell

1 Introduction

Haskell is a pure, lazy, statically typed, functional programming language that is designed to be suitable for teaching, research, and applications [5]. Haskell is also an experimental testbed for type system level features and many of these features have been adopted by Haskell programmer's world wide. One such feature is *type families* [9] and its cousins and that exists in a number of flavours in Haskell's main implementation, Glasgow Haskell Compiler (GHC).

Helium [4] (Section **??**) is a Haskell compiler that is specialized in type error diagnosis. It uses a number of techniques to improve it. The compiler is not as extensive as GHC and only covers a subset of the features that it offers. Recently, the type error diagnosis techniques of Helium have been extended to work with one of the more advanced features of Haskell: GADTs [8]. A new type inference engine, Rhodium [2], has been designed to make the type error diagnosis techniques in Helium compatible with the more advanced features that GHC implements. In this paper we document our work extending Helium with type families, and its many flavours.

Consider the following very simple example:

```
type family Foo a
type instance Foo Float = Int

wut :: Foo Float
wut = 6
```

Here the type instance dictates that in Haskell types, occurrences of `Foo Float` can be compile-time evaluated to **Int**, and since that is the type of the body of `wut`, all is fine.

But what if we had mistakenly written:

```
type family Foo a
type instance Foo Int = Float

wut :: Foo Float
wut = 6
```

Notice that the type `Foo Float` is not reducible by the type family. We thus expect that `Foo Float` cannot be matched with the type **Int** from `6`. GHC, on the other hand, provides the following error message:

```
main.hs:179:7: error:
• No instance for (Num (Foo Float)) arising from the literal
• In the expression: 6
    In an equation for 'wut': wut = 6
    |
179 | wut = 6
    |
```

GHC expects an instance of **Num** for `Foo Float`, which obviously is not there.

Type families imply that certain computations are done at compile time. Since we are in effect evaluating types at compile time, diagnosing type errors that involve type families will need to show which types resulted from which type level computation. Moreover, the design of type families is such that they come with a number of not-so-obvious restrictions, which easily lead to type error messages that are hard to digest by proficient Haskell programmers as they take their first steps in type level programming.

One such rule is the *apartness rule* (Section **??**).

```
type family Loop a where
```

Conference'17, July 2017, Washington, DC, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
  Loop [a] = Loop a
  Loop a = a

loop :: Loop [c]
loop = 'X'
```

At first glance, a (novice) programmer may expect that `Loop [c]` would reduce to `c` by first applying the first instance and then applying the second. The second instance is, however, never applied and GHC gives the following error message:

```
  main.hs:206:8: error:
      * Couldn't match type 'Loop c' with 'Char'
        Expected type: Loop [c]
          Actual type: Char
      * In the expression: 'X'
        In an equation for 'loop': loop = 'X'
      * Relevant bindings include
          loop :: Loop [c] (bound at main.hs:206:1)
      |
  206 | loop = 'X'
      |
```

`Loop c` cannot be reduced because `c` can be instantiated so that it would match with both instances. In other words: the two instances are not apart. Note that the type error message does not explain that subtlety, leaving the programmer to guess what is going on.

This paper offers the following contributions:

- we have implemented all flavours of type families (closed, open, injective, and associated datatypes) in the front end of Helium[1]
- we have developed type family reduction traces to become part of the type error message in case a subject type of the type error report was a consequence of type family applications
- we have devised a number of heuristics to diagnose the cause of certain classes of type family errors.

We do so under the following basic premise: if we have to choose between blaming a Haskell expression for a mistake, or a type family, then we will always choose to blame the latter. This is not unreasonable: type families are often added to existing code to make the types of the code more precise. There will typically already be a reasonable amount of confidence that the value level code is type correct. Note that if we have to choose between blaming the definition of a type family and the use of a type family (as part of some explicit type), then we prefer to blame the latter, which is not any different from how error diagnosis works for type incorrect functions that employ algebraic data types.

[1]Details of the implemented static checks that we assume have been checked before type inferencing are given in the master's thesis of the first author [7]. The static checks are the same as the ones GHC implements.

2 Preliminaries

To be somewhat self-contained, we reiterate some of the details of the various flavours of type families that exist in Haskell and GHC. The feature is an improvement upon functional dependencies [6], that arises when written as an associated type synonym [3] for a given type class:

```
class Collects ce where
    type Elem ce
    empty :: ce
    insert :: Elem ce -> ce -> ce
    member :: Elem ce -> ce -> Bool

    instance Collects [a] where
    type Elem [a] = a
    empty = []
    insert x xs = (x:xs)
    member = elem
```

When we decouple associated type synonyms from type classes, we arrive at *open type families* [9]. A typical example is:

```
type family Sum n m
type instance Sum Zero x = x
type instance Sum (Succ x) y = Succ (Sum x y)
```

which is similar to what one can write on value level, but will now operate at the type level. We can use this in the definition of vector append as follows:

```
vconcat :: Vector e n -> e Vector e m -> Vector e (Sum n m)
vconcat Nil l = l
vconcat (Cons x xs) ys = Cons x (vconcat xs ys)
```

A type family is open when instances can be added freely at any time. In the case of a *closed type family*, all instances are given directly at the definition. The main advantage is that we can allow instances to overlap, and instances will be tried from top to bottom to find the applicable case. *Injective type families* [10] arise when a type family is marked as such *and* the type family behavior is injective. In the following example,

```
type family F a b = r | r -> b where
    F Int Int = Bool
    F Bool Bool = Int
    F Int Bool = Int
```

we say that type family `F` is injective in its second argument `b` as specified by the annotation `r -> b`. This means that the right-hand sides (rhs) of its instances determine only the second arguments on the left-hand side (lhs). `F` is not injective in its first argument as its rhs does not determine it. **Int** as result can be the result of **Int** or **Bool** as the first argument.

```
data TFDeclInfo = TDI {
    tfdName :: Name -- name of the type family
  , argNames :: Names -- names of the arguments
  , tfdType :: TFType
  , injective :: Bool
  , injNames :: Maybe Names -- names that are injective
  , classIdx :: Maybe [(Int, Int)] -- indices of tf args linked to class args (ATS)
  , classDName :: Maybe Name -- name of the accompanying class
  } deriving (Show, Eq)

data TFInstanceInfo = TII {
    tfiName :: Name
  , argTypes :: Types
  , defType :: Type
  , tfiType :: TFType
  , priority :: Maybe Int -- priority inside closed type family
  , classTypes :: Maybe Types -- class types in case of ATS
  , classIName :: Maybe Name -- class name of accompanying class
  , preCompat :: [Int] -- compatible precomputations.
  , tfiRange :: Range
  , closedDeclName :: Maybe Name
  , varNameMap :: Maybe (Map Int Name)
  } deriving (Show, Eq, Ord)

data TFType
  = Open -- open type family
  | Closed -- closed type family
  | ATS -- associated type synonym
  | TypeSyn -- type synonym
  deriving (Show, Eq, Ord)
```

Figure 1. Data structures for a Type Family declaration

3 Some implementation details

We store the family information in a convenient format so that we may access that information during static checks and type inferencing. We introduce two data structures. One in which we store Type Family *declarations* and one in which we store Type Family *instances*. The data structures are given in code snippet 1.

The name of the type family, the names of their arguments and the type of the type family are stored respectively in `tfdName`, `argNames` and `tfdType`. The type of `tfdType` is defined as visible in code snippet **??** and determines what type of type family we are dealing with. The type family may be open, closed, an associated type synonym (ATS), or a type synonym. Furthermore, we store whether the type family is injective and in which arguments. A type family does not have to be injective. Therefore, this attribute is a **Maybe** type. Last but not least, we store the potential name of a type class and the indices of those variable names that are linked to certain variable names of a class. This information is only relevant when we are dealing with an Associated Type Synonym and is empty otherwise. This information is stored in `classIdx` and `classDName`.

In `TFInstanceInfo`, we store information obtained from **type instance**'s. Again, we store the type family name to which the instance belongs. Next, we store the *argument types (lhs)* in `argTypes` and the *definition type (rhs)* in `defType`. We also store the type of type family. The `priority` is only relevant for Closed Type Families and determines the order in which the instances are placed under the declaration. For associated type synonyms, we want to determine if types match for those variables that are equal in the class and ATS definition. We therefore save the types assigned to the ATS instance. `preCompat` saves the indices of other instances under the same closed type family declaration with which the instance is *compatible*. This information is used during the type inference process to determine if an instance may be used to apply a certain type family. Last but not least, we employ a `varNameMap` that contains the names used in the Type Family declaration related to their index. We use this to retrieve the names when we build the *injectivity annotation hint*.

3.1 Extensions to the type inference system of Helium

This section discusses how the X for its type inference mechanism, Rhodium, in Helium is implemented and why certain decisions were made. Some parts of the implementation remained unchanged with respect to the definitions given by Burgers et al. [2]. This section first discusses the extension of the constraint environment of X and then discusses the implementation of X per simplification rule.

Constraint environment. The constraint environment, as described by Vytiniotis et al. [11] and implemented by Burgers et al. [2], is extended on two aspects: axioms and monotypes. Because type families are defined at top level, we extend the axiom definition Q with the following rule:

$$\begin{aligned} Q &::= \ldots \\ &\quad \mid \forall[\bar{\alpha}].\ \tau \sim F\ \bar{\xi} \end{aligned}$$

where $\bar{\alpha}$ denote the touchable type variables and τ denotes the type that represents the type instance body. This type is subject to the restrictions posed by the static checks. $F\ \bar{\xi}$ represents the type family name and its arguments. ξ denotes a *type family free* type. This agrees with the restrictions imposed on type family instance arguments. This axiom is spawned when the programmer defines a type instance in the program.

Furthermore, monotypes are extended to incorporate type family applications in types:

$$\tau ::= \ldots \mid F\ \bar{\tau}$$

where F is the type family name and $\bar{\tau}$ represent the types the F is applied to.

Canonicalization. For type families, the canonicalization rule has the task to deal with type family applications that contain type family applications in their arguments. In other words, *nested type families*. Furthermore, the canonicalization rule implements one of the simplifications that injective type families allow. In figure 2, the canon rules for type families are given. The first two rules handle nested type families. Nested type families are handled by the *unfamily* function. Vytiniotis et al [11] call this procedure *flattening* and it is similar to the flattening procedure described in section **??**. The rule is of the shape $\mathit{unfamily}(\tau) = (\alpha, \xi, Q)$, where τ may contain a type family application. Every type family application in τ is replaced by a fresh type variable β, henceforth known as a β-variable, while the shape of the constraints remains the same. The constraint Q contains equalities of the form $\beta \sim F\ \bar{\tau}$ with a fresh β for every type family $F\ \bar{\tau}$. Take as an example $\mathit{unfamily}(F\ \mathit{Int} \rightarrow G\ \mathit{Bool}) = (\{\alpha, \beta\}, \alpha \rightarrow \beta, \alpha \sim F\ \mathit{Int} \wedge \beta \sim G\ \mathit{Bool})$. The variables α and β are freshly introduced and replace the occurrences of the type families F and G.

$$\begin{array}{lcl}
\mathit{canon}(tv \sim \tau_1\tau_2) & & \\
\quad \text{where } \mathit{unfamily}(\tau_i) = (\bar{\alpha}_i, \xi_i, Q_i) & = & (\bar{\alpha}_1\bar{\alpha}_2, \{tv \sim \xi_1\xi_2\} \cup Q_1 \cup Q_2) \\
\mathit{canon}(F\ \bar{\tau} \sim \tau') & & \\
\quad \text{where } \mathit{unfamily}(\tau_i) = (\bar{\alpha}_i, \xi_i, Q_i) & = & (\bar{\alpha}_1..\bar{\alpha}_n, \{F\ \bar{\xi} \sim \tau'\} \cup (\bigcup_i Q_i)) \\
\mathit{canon}(F\ \bar{\tau} \sim F\ \bar{\sigma}) & & \\
\quad \text{where } \mathit{injArgs}(F\ \bar{\tau}) = (\bar{\tau}') & & \\
\quad \text{where } \mathit{injArgs}(F\ \bar{\sigma}) = (\bar{\sigma}') & = & (\epsilon, \{\tau_i \sim \sigma_i \mid \tau_i \leftarrow \bar{\tau}', \sigma_i \leftarrow \bar{\sigma}'\})
\end{array}$$

Family in application
Nested type families
Injectivity

Figure 2. Canonicalisation rules for type families (partly taken from [1])

Interaction. Figure 3 shows the interaction rules as given by Burgers et al. [2]. Notice that all types on which the interact rule works must be *type family free*, as denoted by the use of ξ. The flattening of type families in equations opens up more possible interactions between those equations and other constraints [11].

$$\begin{array}{lcl}
\mathit{interact}(\tau_1 \sim \tau_2, \tau_1 \sim \tau_2) & = & \tau_1 \sim \tau_2 \\
\mathit{interact}(tv \sim \xi_1, tv \sim \xi_2) & = & tv \sim \xi_1 \wedge \xi_1 \sim \xi_2 \\
\mathit{interact}(tv_1 \sim \xi_1, tv_2 \sim \xi_2) & & \\
\quad \text{where } tv_1 \in fv(\xi_2) & = & tv_1 \sim \xi_1 \wedge tv_2 \sim [tv_1 \rightarrow \xi_1]\xi \\
\mathit{interact}(tv \sim \xi, D\ \bar{\xi}) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) & = & tv \sim \xi \wedge D\ [tv \rightarrow \xi]\bar{\xi} \\
\mathit{interact}(D\ \bar{\xi}, D\ \bar{\xi}) & = & D\ \bar{\xi} \\
\mathit{interact}(tv \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) \text{ or } tv \in fv(\xi_2) & = & tv \sim \xi_1 \wedge F\ [tv \rightarrow \xi_1]\bar{\xi} \sim [tv \\
\mathit{interact}(F\ \bar{\xi} \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & = & (F\ \bar{\xi} \sim \xi_1) \wedge (\xi_1 \sim \xi_2)
\end{array}$$

Figure 3. Interaction rules (taken from [1])

The interaction rules introduce two new rules that involve (flattened) type families. The first interacts $tv \sim \xi_1$ and $F\ \bar{\xi} \sim$ The rule allows the type variable tv to be substituted inside the type family arguments and definitions ξ and ξ_2. The second new rule and the last rule in figure 3 allows the interaction of two type family constraints that are applied equally. This should result in the fact that their definitions ξ_1 and ξ_2 should be equal, which is why the new constraint $(\xi_1 \sim \xi_2)$ is created.

We introduce one additional rule for injective type families. If the last rule in figure 3 fails, we may check if ξ_1 and ξ_2 are equal. If this is the case, we may introduce evidence that the injective arguments to F are equal. We may thus introduce those new equalities. We formalize this new rule as follows:

$$\begin{array}{l}
\mathit{interact}(F\ \bar{\xi}_1 \sim \xi_3, F\ \bar{\xi}_2 \sim \xi_4) = (F\ \bar{\xi}_1 \sim \xi_3) \wedge \{\xi_1 \sim \xi_2 \mid \xi_1 \leftarrow \bar{\xi}_1, \xi_2 \leftarrow \bar{\xi}_2 \\
\quad \text{where } \mathit{equal}(\xi_3, \xi_4)
\end{array} \tag{1}$$

Simplification. The simplification rules are very similar to the interaction rules as explained in section **??**. We therefore shortly focus on the two rules that this simplification step introduces. These are visible in figure 4.

$$\begin{array}{lcl}
\mathit{simplify}(F\ \bar{\xi} \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & = & \xi_1 \sim \xi_2 \\
\mathit{simplify}(tv \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) \text{ or } tv \in fv(\xi_2) & = & F\ [tv \rightarrow \xi_1]\bar{\xi} \sim [tv \rightarrow \xi_1]\xi_2
\end{array}$$

Figure 4. Simplify rules for type families (taken from [1])

The first rule simplifies two type family applications with the exact same arguments. The only difference with the interact version of this rule is that the given, left-most constraint is not returned as it is not needed further in this phase of the simplification process. The same goes for the second rule.

Top-level react. Next to the canonicalization rule, the top-level react is the most interesting rule when introducing type

families. As mentioned, type families introduce a new type of axiom in the form $\forall[\bar{\alpha}].\ \tau_Q \sim F_Q\ \bar{\xi}_Q$ which represents a `type instance`. This rule is used to react with type family applications in the constraint system. The top-level react rule for type families performs the following steps, given a constraint of the form $F\ \bar{\xi} \sim \xi$. Note that all types must have been made *family free*.

1. We loop over all axioms that are defined in the constraint system and check if, for axiom $\forall[\bar{\alpha}].\ \tau_Q \sim F_Q\ \bar{\xi}_Q$ and type family application $F\ \bar{\sigma}$, $F = F_Q$. If not, we consider the next axiom.
2. If $F = F_Q$, we will try to unify $\bar{\xi}_Q \sim \bar{\xi}$, where the variables $\bar{\alpha}$ are considered touchable and may thus be unified to another type. This allows us to apply a type application on a more general type instance.
3. If step 2 is successful, the resulting substitution θ is returned. The result of the top-level react call is then $\theta\tau_Q \sim \xi$ where we apply the substitution θ to the definition of the type family axiom ξ_Q. We thus replace $F\ \bar{\xi}$ with its definition.
4. If step 2 is unsuccessful, we return to step 1 and loop further over the known axioms. When no matching axiom is found after all have been considered, the constraint is left residual.

The above steps work well for open type families, as there is no order implied. For closed type families, we utilize the *Axiom Closed Group* as explained in section **??**. Because type family names are unique, we can extend the axiom loop function to also search for axiom groups. When such a group is found, we only need to loop over the axioms inside it. Furthermore, because we know it represents an ordered set of axioms from a closed type family, we can check the axioms in order. This also allows us to perform the *compatibility* and *apartness* checks over the complete group as explained in section **??**. Compatibility is computed before the type inference process.

Top-level improvement. When the type family that we want to simplify is injective, and the above process fails, we may attempt to perform a *top-level improvement*. When, again, given an axiom of the form $\forall[\bar{\alpha}].\ \tau_Q \sim F_Q\ \bar{\xi}_Q$ and a contraint of the form $F\ \bar{\xi} \sim \xi$, we perform the following steps:

1. We attempt to *pre-match* ξ and τ_Q. Pre matching describes the combined process of matching and pre unification. In short, we use the pre unification algorithm (figure **??**) while only keeping the types in one order. If pre matching fails, we return. Else, we obtain the resulting substitution *subst* and advance to the next step.
2. We apply *subst* on the axiom family $F_Q\ \bar{\xi}_Q$ and check if it matches with its original version. This is important because it ensures that injectivity is indeed correct [10]. Again, if it fails, we return.
3. At this point, we may create new evidence based on the injective properties of the type family. We apply *subst* over the injective arguments in $\bar{\xi}_Q$ to create $\bar{\xi}'_Q$. We then build a set of new constraints as follows:

$$\text{injConstr} = \{\xi'_Q \sim \xi \mid \xi'_Q \leftarrow \bar{\xi}'_Q, \xi \leftarrow \bar{\xi}\} \tag{2}$$

4. We return the set of constraints *injConstr* and $F\ \bar{\xi} \sim \xi$. This constraint may be improved using the constraints in *injConstr* but may also be left residual in some form after type inference is complete. This, however, introduces a subtlety. It may the case that $F\ \bar{\xi} \sim \xi$ may not be able to be improved at all. In this case we do not want to perform above steps on it again as this will not provide new information. We, therefore, annotate the constraint with the notion that top-level improvement was already applied. This prevents the type inference system from looping.

4 The Reduction Trace

Type family applications may introduce new type families. Furthermore, type families may occur as arguments to other type families in annotated types. It may therefore take several type family applications before a base type is reached. During these applications, several intermediate types are created that are not present in the source code. This is intentional but poses a problem if such an intermediate type may not be reduced further. The resulting type error message will then contain a type that is not present in the source code, which may confuse the programmer. Furthermore, when a type family is fully reduced and the resulting base type creates an error, it may also be hard to find the original type as the previous steps are lost after type inference terminates.

In this section, we discuss a new feature of Helium that traces changes concerning to types that contain type families, including changes to type family arguments as a result of injective properties. In the design we aimed to achieve the following requirements for this feature, a so-called *Reduction Trace*:

- The Reduction Trace should be complete, that is from start to finish, omitting no steps.
- The trace should be presented in a type error clearly. The steps that are part of the trace should be presented one by one with possibly extra information that explains why the step was taken. For example, what instance was applied? How often was it applied and why?
- The trace should be able to distinguish between different kinds of reductions, e.g., left-to-right applications, but also argument injections where argument information is obtained because of the injective properties of the type family.

- The trace should be able to compact. Sometimes a particular instance is left-to-right applied many times in succession, and it would be helpful to describe these many steps in a single trace step
- The Reduction Trace should be an extension of the existing system. The trace should be gathered during the steps of the type inference system. We impose this requirement as we do not want to adapt the existing type inference system, which would be risky undertaking.

Our starting point are monotypes, defined as follows:

Monotypes	τ	::=	α	variables
		|	$\tau_1\tau_2$	type variable application
		|	A	constants
		|	$F\ \bar{\tau}$	type families

Figure 5. The definition of monotypes in Helium

To build a trace of type family applications, we extend the monotypes with a *reduction step*. The goal of this step is to save a potential application and thus reduction, of a type family. We add this step to monotypes, because every monotype may potentially be the result of a type family application. Furthermore, it allows us to quickly see if a type is the result from a type family reduction. This has its use when checking if a type family may be to blame for a type inconsistency.

We define *ReductionType* (what kind of reduction is it, the location where the applied instance can be found in the source code), *ReductionStep* (basically the in and output type of the reduction, and *ReductionTrace*, which is a step annotated with the number of times the step was applied:

ReductionType	*rt*	::=	LeftToRight (τ, τ') *location*
		|	ArgInjection (τ, τ')
ReductionStep	*rs*	::=	Step τ τ' *rt*
ReductionTrace	*trace*	::=	[(*rs*, *Int*)]

Reduction steps can take place at two points during type inference: during a top-level reaction of a type family and at the interaction of a variable constraint with a type family constraint. The first leads to a LeftToRight reduction step and the second to an ArgInjection.

For a LeftToRight reduction step, given the constraint $F\ \bar{\xi} \sim \tau$ and *axiom* $\forall[\bar{\alpha}].\ \tau_Q \sim F_Q\ \bar{\xi}_Q$, we first check that *map unify* $\{\xi \sim \xi_Q \mid \xi \leftarrow \bar{\xi}, \xi_Q \leftarrow \bar{\xi}_Q\}$ succeeds. If so, then we create the new constraint $\tau'_Q \sim \tau$ where τ'_Q is the substituted version of τ_Q, and we extend τ'_Q with the following step (note that we only save τ'_Q at this point as a matter of convenience) :

$$\text{Step } \tau'_Q\ (F\ \bar{\xi})\ (\text{LeftToRight } (F_Q\ \bar{\xi}_Q, \tau_Q)\ \text{loc}(\textit{axiom})).$$

An ArgInjection reduction step is the result of applying rule in figure 6: This rule *inserts* new information obtained for arguments of a Type Family. As explained, this may be due to the injective properties of the type family. Whether this is the case or not does not matter as we always want to show any changes to the type family to the programmer when an error occurs. When we insert the reduction step, however, is quite subtle. First of all, we only do so if $tv \in fv(\bar{\xi})$ because we will only substitute an argument of F when this is the case. Furthermore, we do not add a reduction step when tv is a β-variable. As explained in section 3.1, such variables are only introduced during the *flattening* process which allows the type inference process to consider nested type family separately. When we encounter a β-variable in this rule, we are only undoing a flattening step which is introduced by the type inference system and should thus not be shown to the programmer. To conclude, we introduce the following reduction step to the type $F\ [tv \rightarrow \xi_1]\bar{\xi}$ when $tv \in fv(\bar{\xi})$ and $tv \neq \beta$:

$$\text{Step } (F\ [tv \rightarrow \xi_1]\bar{\xi})\ (F\ \bar{\xi})\ (\text{ArgInjection } (tv, \xi_1)).$$

The after type is the same type family as the before type with its arguments substituted by the first constraint given to the interact rule. The ArgInjection reduction type saves this constraint to be able to show it in a possible error later on.

4.1 Building a Reduction Trace

The above information is collected along with the type inference process. If we find a type inconsistency, and a type, say τ, that will be reported on in the type error message was the result of type family rules being applied, then we can use that information to construct a trace. Our implementation uses two phases.

Phase 1: following reduction steps. We apply the reduction steps, in reverse order, starting with τ and working our way backwards. At some point, a type is reached without a reduction step. In this case, we inspect the type and check if it is a type application or a type family. If it does and the type contains nested monotypes, we proceed to phase 2, otherwise we are done.

Phase 2: diving deeper. It is important to note that the reduction steps of recursive monotypes do *not* know that they were part of another type family or type application. This is due to the flattening that happens during type inference. For the nested monotypes, we first perform Phase 1, to obtain all their reduction steps, and this may again lead to recursing on these types. When we are done tracing back all the type family steps, we need to insert the steps back into the higher-level type of which the nested monotype was a part. Let us show this situation with an example. Let us have the following type families

$$interact(tv \sim \xi_1, F\ \bar{\xi} \sim \xi_2) \\ \text{where } tv \in fv(\bar{\xi}) \text{ or } tv \in fv(\xi_2) \quad = \quad tv \sim \xi_1 \wedge F\ [tv \rightarrow \xi_1]\bar{\xi} \sim [tv \rightarrow \xi_1]\xi_2$$

Figure 6. Interaction rule

```
type family H a b where
    H Int Int = Float

type family J c where
    J Int = Int
```

and the type haskellH (J Int) (J Int). During type inference, this type is flattened to haskellH beta1 beta2 with constraints `beta1 ~ J Int` and `beta2 ~ J Int`. haskellJ Int reduces to haskellInt but does not know that it was initially part of H. Fortunately, inserting a trace back into a higher level type is not difficult: we loop over every step in the trace and encapsulate the *after* and *before* types in the higher level type. To be able to do this, we need to know what the argument was that the type belonged to. For type families, we therefore keep track of which arguments were already processed. We therefore also choose to present the traces within these arguments from left to right.

As a last step we compact multiple applications of the same reduction rule to reduce the trace. Consider for example

```
type family Loop a where
    Loop [a] = Loop a
    Loop a = a
```

and the type haskellLoop [[[[Int]]]]. It is clear that the result type after type inference is haskellInt. However, it takes *five* top-level react phases to reach this conclusion of which *four* use the exact same instance. Showing these four steps separately is unnecessary so we squash these four steps and instead mention in the error how often the instance was used. The *after* and *before* types of the new Step become the after step of the first step and the before step of the last step in the sequence. In a later section, we shall show some examples of Reduction Traces.

4.2 Difficulties with the current approach

There are cases in which our reduction traces currently fail to build a trace from start to finish. In some cases, it even fails to build a trace at all. It unveils a problem with the reduction traces that arise from the fact that we construct our traces during the standard type inference process, which of itself is not necessarily a linear, predictable process.

Problem 1: the order of phases. The phases of building a reduction trace as described in section 4.1 are executed in a fixed order. Only when a type does not contain a reduction step, do we consider the possible traces in its recursive arguments. However, a type family instance may contain a type family on its right-hand side. After application, we may thus have a new type family inside a type which may be reduced to another type. As a result, we have a type that has a reduction step and also contains a recursive type with a reduction step. In this situation, the current implementation falls short. The ordering of phases should become more subtle.

Problem 2: order of applications of a nested type family. Consider

```
type family Const a b where
    Const a b = b

type family Id c where
    Id c = c

tfconst :: a -> b -> Const a (Id b)
tfconst x y = x
```

Here, the function `tfconst` contains a type error because `Const a (Id b)` reduces to b and x has type a. So how do the type family reductions take place:

1. `Const a (Id b)` is flattened, leading to `Const a beta1` and the new constraint `beta1 ~ Id b`.
2. `Id b` is reduced to b and we obtain the constraint `beta1 ~ b`.
3. `Const a beta1` is reduced `beta1`.
4. `beta1` is substituted with b in the constraint `beta1 ~ a` to create the constraint `a ~ b` where we return an error.

Steps two and three violate the order in which most reductions take place. The top-level type family is reduced before the reduced nested type family is reinserted. As a result, b only contains the step that it was reduced from `Id b`. This situation only happens in case we have very general type families like `Const` and `Id` that may be applied to any type and thus also with type variables. In situations like this, the trace becomes incomplete as the reduction from `Const` will be omitted from the trace.

We designed a fix by restricting top-level reaction that a type family may only attempt a reaction if all its arguments are *beta free*. In other words, all possible nested type families inside a type family must be considered first before the top-level family is reduced. In case a nested type family cannot be reduced, the top-level family will also stay unreduced. We argue that this does not change the type system: we do not allow fewer type families to be type-checked, and if a type family is fully reducible, the type inference system will still do so, albeit in a different order. In case a type family is not

(fully) reducible, the type inference system will still return an error.

We did find, however, that in some cases, the error is a bit too general. Consider the following somewhat less general example:

```
type family Const a b where
    Const a b = b

type family Id c where
    Id Int = Int

tfconst :: a -> b -> Const (Id a) b
tfconst x y = x
```

`Id` is now less generic and only allows **Int** as an argument. As a result `Id a` in the type signature will not be reducible. In case we still allowed β-variables to be in the type family arguments during the top-level react phase, `Const` would still be reducible, and we would get an error that tells us that `Id a` is not reducible. In the new situation, `Const` will not be reduced, as it will remain in the form `Const beta1 b` after flattening. After substitution, the type inference system will tell us that `Const (Id a) b` in total is not reducible. We argue that this is not a problem as the situation is very specific. Furthermore, as we will see when we discuss our heuristics later in this paper: the type to blame is still part of the type and can be blamed.

4.3 Problem 3: looping trace construction

During the creation of the trace, we try to substitute some type variables to check if the result type has a trace that we may potentially explore. This turned out to be dangerous because it can cause looping. Especially when the substitution resulting from type inference is certainly not correct. If the type variable we substitute is also present in the before type, we end up in a loop because we eventually consider the type variable again. In the current implementation, we built in a safety measure that stops the creation of the trace after a certain amount of going into recursion. This amount is currently set to 50. See [7] for an example.

5 Heuristics for type families

Will be in the final version. And will be discussed in the presentation

6 Related work

Will be in the final version. It will discuss more or less the same work as does [2].

7 Conclusion

Will be in the final version.

Acknowledgments

We thank for Wouter Swierstra, Alejandro Serrano and Joris Burgers for help and advice.

References

[1] Joris Burgers. 2019. *Type error diagnosis for OutsideIn(X) in Helium.* Master's thesis. Utrecht University.

[2] Joris Burgers, Jurriaan Hage, and Alejandro Serrano. 2020. Heuristics-based Type Error Diagnosis for Haskell: The case of GADTs and local reasoning. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*. 33–43.

[3] Manuel MT Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type synonyms. *ACM SIGPLAN Notices* 40, 9 (2005), 241–253.

[4] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. 62–71.

[5] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. 1992. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices* 27, 5 (1992), 1–164.

[6] Mark P Jones. 2000. Type classes with functional dependencies. In *European Symposium on Programming*. Springer, 230–244.

[7] N. Kwadijk. 2022. Implementing Type Families in Helium. A study on improving Type Error Diagnosis for Type Families in Haskell. (2022). TBA.

[8] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. *ACM SIGPLAN Notices* 41, 9 (2006), 50–61.

[9] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 51–62.

[10] Jan Stolarek, Simon Peyton Jones, and Richard A Eisenberg. 2015. Injective type families for Haskell. *ACM SIGPLAN Notices* 50, 12 (2015), 118–128.

[11] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn (X) Modular type inference with local assumptions. *Journal of functional programming* 21, 4-5 (2011), 333–412.

Higher-ranked region inference for polymorphic, lazy languages

Ivo Gabe de Wolff
i.g.dewolff@uu.nl
Utrecht University
Utrecht, Netherlands

Jurriaan Hage
j.hage@hw.ac.uk
Heriot-Watt University
Edinburgh, Scotland

ABSTRACT

Region based memory management is a compile-time alternative to garbage collection, where allocations are placed in lexically scoped regions, either manually by the programmer or by a static program analysis: region inference. The precision of such an analysis—that is, determining in which region an object should be allocated, and when regions can be deallocated—has a significant impact on the run-time performance of the program. To increase the precision of region inference for higher-order programs, we adapt higher-ranked type and effect systems to region inference. In this work we discuss the problems of extending prior work on higher-ranked program analysis to polymorphic languages, and present our novel region inference analysis for a lazy, polymorphic language with higher-order types.

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

memory management, regions, type-based program analysis, higher-ranked polymorphism, Haskell

ACM Reference Format:
Ivo Gabe de Wolff and Jurriaan Hage. 2025. Higher-ranked region inference for polymorphic, lazy languages. In *IFL '22: 34th Symposium on Implementation and Application of Functional Languages.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Region based memory management is a compile-time alternative to garbage collection, where the heap is split into lexically-scoped regions. Each allocation is placed in a region, and at the end of the lexical scope of a region, all objects in the region may be deallocated. The programmer or the compiler has to decide where regions are introduced, and for each allocation the region it is placed in.

We can support region based memory management in the lambda calculus by extending it with two constructs: `letregion` to introduce a new region, and `@` to link each allocation with a region. Regions are referred to by region variables, denoted by ρ.

IFL '22, August 31—September 2, 2022, Copenhagen, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{array}{lll}
e ::= \ldots & & \\
\quad \mid \texttt{letregion}\ \rho\ \texttt{in}\ e & \textit{Region introduction} & (1) \\
\quad \mid e\ @\ \rho & \textit{Allocation} &
\end{array}
$$

The construct `letregion` ρ `in` e introduces a new region ρ which can be used in the body e. This region may be used to store intermediate values in the computation e and all objects in that region will be deallocated when the computation finishes. The result of that computation should thus not be stored in region ρ.

Region inference, introduced by [20], converts a program without regions to a program that uses regions. This has been extensively studied in the MLKit compiler [24, 25]. Vejlstrup [26] presented a static program analysis to find the memory requirements of the regions (as they can in general be unbounded) to efficiently allocate many regions on the stack instead of the heap [4]. For some programs region inference gives imprecise results, where many allocations are placed in a region with a long lifetime; either the program should then be rewritten to be more region-friendly, which is often difficult to do for the programmer, or the compile-time region inference can be combined with run-time garbage collection [9, 11]. Recently, regions have been popularized under the name lifetimes in Rust, a programming language that enforces memory-safety at compile time using regions [17]. Contrary to MLKit with region inference, Rust requires the user to place regions in the program.

1.1 Region inference

Regions can be inferred with a static program analysis [3, 24] via a type and effect system [18]. In a type and effect system, the type system of the source language is annotated with analysis-specific annotations. For region inference, a function type is annotated with a discription of how the function may use regions. The precision of such an analysis will impact the run-time performance of the program.

In this paper, we introduce a new region inference algorithm for a polymorphic language with higher precision for higher order functions. We illustrate the general ideas of the analysis with some examples. First, consider the `inc` function, which increments its integer argument by one:

```
inc x = x + 1
```

If we assume that integers are boxed, then the evaluation of this function will lead to allocating a new integer object. Hence there is no relation between the region of the argument `x` and the region where the result is stored, and we can thus store them in different regions. Now consider the identity function:

```
id x = x
```

In this case there is a relation between the region of the argument and the result, as we do not allocate a new integer object, but reuse the existing object which is already placed in a region. Instead of enforcing that the those regions are the same, as done in [24], we only require that the region of the argument *lives at least as long as* the region of the result. Then we can pretend that the result is actually stored in the output region, as the input region cannot be deallocated when the output region is still needed.

The components of a compound value may be stored in multiple regions; for instance in case of a list, the list objects (the nil `[]` and cons `:` objects forming the linked list structure) may be stored in a different region than the elements in the list. For the function `map`, the analysis must decide on the regions for both the list objects and the elements.

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

The list objects of the result are allocated in the function, similar to the integer objects in `inc`, so no relation between the input region and the output region is enforced (assuming strict evaluation). How the regions for the elements should be handled depends on function `f`, which is an argument to `map`. In our analysis, we let annotations also be functions: the annotation of `map` is a function taking the annotation of `f` as an argument and then returning the relation between the regions of the input and the regions of the output. The annotations of higher order functions are thus higher order annotation functions. The analysis can now find that the input and output regions of `map inc` are unrelated. In case of `map id`, we do find that the region of the elements in the list of the input should live at least as long as the region of the elements in the output. By handling higher order functions this way (with higher-ranked polyvariance, that is polymorphism on annotation variables), and letting annotations be functions as well, we can get higher precision.

Lazy values are also treated as compound values: we assign them one region for the thunk, the run-time representation of a lazy value, and one region for the computed value. The free variables of a lazy value must outlive the thunk as those values need to be available when the thunk needs to be evaluated. This may give them a long lifetime and cause high memory usage. Our analysis makes use of strictness information, inferred by a strictness analysis [5, 13] or given by the user. This may reduce the number of lazy values or shorten their lifetime. To illustrate the latter, the region of the thunk may be deallocated before the region of the values, which for instance happens with the regions of the elements of an array in the function `head`: this function takes a list of lazy values, and returns a strict value. The region of the thunks could be deallocated after calling `head`.

1.2 Contributions

We introduce a region inference algorithm with higher-ranked polyvariance for a polymorphic, lazy language. We extend on previous work on analyses with higher-ranked polyvariance, as applied to flow analysis [14] and dependency analysis [22], by using a lambda calculus as annotation language, and adapt this to region inference. Whereas previous work on higher-ranked polyvariance was applied to monomorphic languages, we extend this to polymorphic languages and show that the precision of the type system is not orthogonal to analysis precision: by the inclusion of polymorphism in the source language, previous approaches of higher-ranked analysis do not work any more. We show that the main problem is that fixed-point iteration isn't guaranteed to terminate and that some fixed-points are undecidable, and work towards possible solutions.

We have an implementation of the analysis in the Helium compiler as described in section 7.1.

1.3 Analysis overview

Our region analysis splits the heap in regions and links each allocation to a region. As the components of a compound value may be stored in multiple regions, for instance a list whose elements are stored in a different region than the list objects, we assign regions to values by defining a mapping $P_\Delta : \tau \to \hat{P}$ from types of the source language τ to the shape of the region variables $\hat{P}$ called the *region sort*, which is a structure of tuples of region variables.

To represent the "lives at least as long as" relations between those variables, we use annotations. The language of annotations is based on the lambda calculus; annotations can be functions taking annotations and regions as arguments. Such an approach for higher-ranked analysis has been applied to other analyses [14, 23]. When analysing a lambda function, the annotation function takes the regions where the arguments and the returned value of a function are stored and returns the constraints between these regions.

Besides taking the regions of arguments of the source function, an annotation function also takes annotations of the arguments as arguments in an annotation function. For instance, the annotation of `map` takes the annotation of the function argument as an argument in the lambda calculus. This annotation language is introduced in section 3.

In the analysis we can use the region variables of arguments and the return value and we can allocate regions local to a function. This is not always sufficient; sometimes we need to introduce regions not bound by an argument or return value of a function, and not bound locally in the function. In such case we can add additional region arguments to a function.

To generate those annotation terms, we introduce the escape check in section 4, which figures out for each region whether it escapes the lifetime of the function and/or can be unified with another region. If it does escape, and cannot be unified, then it becomes an additional region argument. To handle (mutual) recursive declarations, we use fixed-point iteration (section 5). Whereas previous work has shown that fixed-points in a higher-ranked analysis are guaranteed to exist and be reached with fixed-point iteration in a language without polymorphism, we show that by the inclusion of polymorphism, some functions do not have a fixed-point. Finally we show the mapping from source terms to annotation terms in section 6. At the end, we briefly mention how data types are supported in the analysis in section 6.4.

1.4 Source language

We introduce the analysis on the source language based on System F [10] with laziness [2] whose types and syntax are shown in figure 1. We write $\tau_1 \to \tau_2$ to denote the type $(\to)\ \tau_1\ \tau_2$, (τ_1, τ_2) for a tuple type $(,)\ \tau_1\ \tau_2$ and similar for tuple types with more elements. A strict type $!\tau$ may only be used as the argument type of a function or

Figure 1: Source language

Types

$$\begin{array}{lll} \tau ::= (\rightarrow) \mid D & & \textit{Function \& Data type} \\ \quad \mid \tau\ \tau \mid \forall\alpha.\ \tau & & \textit{Application \& quantification} \\ \quad \mid \alpha \mid\ !\tau & & \textit{Type variable, Strict type} \\ \quad \mid () \mid (\ ,\ ) \mid (\ ,,\ ) \mid \ \ldots & & \textit{Unit, Tuples} \end{array}$$

Program

$$p ::= \left\{ \overline{x : \tau = f} \right\}$$

Toplevel bindings

$$\begin{array}{lll} f ::= \backslash x : \tau \rightarrow f \mid \forall\alpha.\ f & & \textit{Lambda, Quantification} \\ \quad \mid e & & \textit{Function body} \end{array}$$

Patterns

$$p ::= (\overline{x : \tau}) \mid C\ \overline{@\tau}\ \overline{(x : \tau)} \qquad \textit{Tuple, Constructor}$$

Expressions

$$\begin{array}{lll} e ::= c & & \textit{Call} \\ \quad \mid \texttt{let!}\ x : \tau\ =\ e\ \texttt{in}\ e & & \textit{Strict let} \\ \quad \mid \texttt{let}\ x : \tau\ =\ c\ \texttt{in}\ e & & \textit{Lazy let} \\ \quad \mid \texttt{let rec}\ \overline{x : \tau\ =\ c}\ \texttt{in}\ e & & \textit{Recursive let} \\ \quad \mid \texttt{match}\ x\ \left\{ \overline{p \rightarrow e} \right\} & & \textit{Pattern match} \end{array}$$

Call

$$\begin{array}{lll} c ::= x \mid C & & \textit{Variable, Constructor} \\ \quad \mid () \mid (\ ,\ ) \mid (\ ,,\ ) \mid \ \ldots & & \textit{Unit, Tuple} \\ \quad \mid c\ x \mid c\ @\tau & & \textit{Value \& type application} \end{array}$$

as the type in a variable binding. We write $\tau'[\alpha := \tau]$ to denote the instantiation of α with τ in τ' and $(\forall\alpha.\ \tau')\ \tau$ reduces to $\tau'[\alpha := \tau]$. We treat $!(\forall\alpha.\ \tau)$ and $\forall\alpha.\ !\tau$ as equal types.

A program p consists of a list of bindings $x : \tau = f$. We may omit types in examples if they are trivial or redundant. Lambdas and quantification may only occur at the root of a binding, which is assured in the language by the separation between toplevel bindings f and expressions e. The bound value of a lazy and recursive bind must be a call c, which is a chain of type and value arguments applied to a function, tuple or constructor. The reason for this restrictions is that region analysis needs concrete information on the runtime behaviour of closures and this representation makes that explicit and corresponds to how a functional program can be executed. A program in a language as Haskell [16] can be converted into this format with lambda lifting, by moving non-trivial subterms to let bindings and/or new toplevel declarations with arguments for the free variables [15]. Strictness can be inferred automatically [5, 13].

Expressions are always strict: when an expression is executed, then its value is evaluated to a value of a strict type. Constructor calls must be saturated (all arguments must be given) and are also strict. Hence a strict let and a let whose binding is a constructor call must have a strict type.

We define the arity of a function as the number of lambdas in the definition.

$$\begin{aligned} \text{arity}(\backslash x : \tau \rightarrow f) &= 1 + \text{arity}(f) \\ \text{arity}(\forall\alpha.\ f) &= \text{arity}(f) \\ \text{arity}(e) &= 0 \end{aligned} \tag{2}$$

The arity is thus not a property of the type; a definition $x_1 = \backslash x \rightarrow x$ has arity 1 but $x_2 = x_1$ has arity 0, whereas they have the same type. Calls to a function may provide more or fewer arguments than the arity of the function. If a function returns a function, then it can be called with more arguments than its arity (currying).

A thunk is the runtime representation of a lazy computation. It is represented as a linked list of arguments and a function pointer. A lazy computation is, by the syntax in figure 1, always a function call and can thus be represented in this thunk form. A call providing too few arguments results in a thunk where the remaining arguments can be passed later on. Each item of the linked list corresponds with one argument to the call and the last element of this list points at the callee. After evaluating a thunk, we store the resulting value in the thunk, to avoid recomputing the same expression.

We assume that all values are stored as pointers to objects. The analysis could however be adapted to not infer regions for unboxed values.

2 REGION VARIABLES

Region inference decides for each object, in which region it will be allocated. We reference those regions using region variables, which are denoted by ρ, the Greek letter rho. A value in the source language may need multiple region variables, as nested fields of an object may be allocated in different regions. Instead of annotating the types with those regions, for instance as $\texttt{!Integer}^{\rho_1}$, we store the regions separately in a structure of tuples of region variables. The reasons for this change are that this allows us to have a simpler annotation language and makes the analysis more precise for some polymorphic functions. This will become clear when we introduce the annotation language in section 3. We write $\hat{\rho}$ to denote a tuple or tree-structure of region variables and use tuples of regions to structure multiple region variables.

$$\hat{\rho} ::= \rho \mid\ (\hat{\rho}, \hat{\rho}, \ \ldots) \tag{3}$$

Note that the tuple may also be empty (a unit), which we use for types that do not require additional region variables, such as integers, and we also support tuples of one element. We have two special regions, the global region ρ_{global} and the bottom region $\rho_\perp$. The lifetime of the global region is the full execution of the program. This region is used for global variables and for code that cannot be analysed. The bottom region has no meaning at runtime. All regions outlive the bottom region, but no other region is outlived by the bottom region. It is used in the analysis for regions which do not escape out of a function. We call the "type" of region variables the *region sort*, denoted by $\hat{P}$ (P is the Greek capital letter rho).

$$\begin{array}{lll} \hat{P} ::= P & & \textit{Monomorphic region sort} \\ \quad \mid P\langle\ \alpha\ \overline{\tau}\ \rangle & & \textit{Polymorphic region sort} \\ \quad \mid \left(\hat{P}, \hat{P}, \ \ldots\right) & & \textit{Tuple} \end{array} \tag{4}$$

The tuples allow us to create a tree of region variables. A leaf of that tree may be a monomorphic region sort, which is just a single region variable, or a polymorphic region sort, which is used for values of a polymorphic type. When instantiating such a polymorphic type, the polymorphic region sorts will be replaced by the region sorts of the instantiated type. Throughout this paper, we use the hat to denote a tuple-structure.

We write $\hat{\rho} : \hat{P}$ to denote that regions $\hat{\rho}$ have region sort $\hat{P}$. For clarity, we will sometimes interleave both structures and for instance write $(\rho_1 : P,\ \rho_2 : P)$ instead of $(\rho_1, \rho_2) : (P, P)$.

2.1 Region assignment

We define a mapping $P_\Delta : \tau \to \hat{P}$ from types of kind $*$ to region sorts. For strict values, we need one region to store the value and for lazy values we need one region for the thunk and one for the evaluated value. As the thunk and the evaluated value are stored in different regions, the thunk may be deallocated earlier than the evaluated value in some programs. Depending on the type, we may also need regions for nested values, for instance the elements of a list, which are found in P°_Δ. The subscript Δ (Delta) represents the data type environment, which we briefly discuss in section 6.4.

$$
\begin{aligned}
P_\Delta(!\tau) &= (P, P^\circ_\Delta(\tau)) \\
P_\Delta(\tau) &= (P, P, P^\circ_\Delta(\tau)) \text{ if } \tau \text{ is not strict}
\end{aligned}
$$

$$
\begin{aligned}
P^\circ_\Delta(\tau_1 \to \tau_2) &= () \\
P^\circ_\Delta(()) &= () \\
P^\circ_\Delta((\tau_1, \tau_2)) &= (P, P, P^\circ_\Delta(\tau_1), P, P, P^\circ_\Delta(\tau_2)) \\
P^\circ_\Delta(\alpha\ \tau_1 \ldots \tau_n) &= P\langle \alpha\ \tau_1 \ldots \tau_n \rangle \\
P^\circ_\Delta(\forall \alpha.\ \tau) &= \text{Substitute } P\langle\ \alpha\ \overline{\tau}\ \rangle \text{ with } P \text{ in } P^\circ_\Delta(\tau)
\end{aligned}
\tag{5}
$$

Function P°_Δ gives the region sort for the fields of a type. Functions do not have additional region parameters, as the annotation on the function is an annotation function taking the region arguments of the input and output. For each element of a tuple, we need a region to store the thunk for that element, a region for the evaluated value and a region for fields of the element.

2.2 Polymorphism

To illustrate why we need polymorphic region variables, we will consider the following example and explain why other approaches have less precision.

```
ifThenElsePair :: Bool -> a -> a -> a -> (a, a)
ifThenElsePair True x y z = (x, y)
ifThenElsePair False x y z = (y, z)
```

In a classic type and effect system, we would, in the annotated type system, leave the type variable a, to be instantiated later at the uses of this function. This does mean that the types (including their regions) of arguments `x` and `z` should be equal, which causes poisoning. If the function was not polymorphic, but for instance specialized to `[Int]`, then the regions may be different. Now we have two annotated types and neither of them is strictly better than the other. In this type system we would not have principal types: there is no guarantee that a most general type for a function exists, from which all other types can be derived.

By storing the region variables of a polymorphic value separately, we can avoid this form of poisoning. As the number of regions needed for this type variable depends on the type it is instantiated with (it could be a compound value as a list), we have special syntax to denote the regions of a type variable, $P\langle\ \alpha\ \overline{\tau}\ \rangle$. Depending on the kind of the type variable, the type variable may also be applied with other types, which for instance happens with type variable m in the type of `return :: Monad m => a -> m a` for monads in Haskell. When a type variable is instantiated, we will substitute this polymorphic region sort, which are the leaves in the tree of region variables, with the region sort of that type, by substituting it with the subtree of that region sort.

2.3 Instantiation

Region sorts may be instantiated, by substituting a type variable α with a type τ denoted by $\hat{P}[\alpha := \tau]_\Delta$. The substitution requires environment Δ to instantiate the type variable of a polymorphic region sort ($P\langle\ \alpha\ \overline{\tau}\ \rangle$) and expand it to the region sort of the instantiated type.

$$
\begin{aligned}
P[\alpha := \tau]_\Delta &= P \\
P\langle \alpha'\ \tau_1\ \ldots\ \tau_n \rangle[\alpha := \tau]_\Delta &= \begin{cases} P^\circ_\Delta(\tau\ \tau'_1\ \ldots\ \tau'_n) & \text{if } \alpha = \alpha' \\ P\langle \alpha'\ \tau'_1\ \ldots\ \tau'_n] \rangle & \text{otherwise} \end{cases} \\
&\text{where } \tau'_i = \tau_i[\alpha := \tau] \\
\left(\hat{P_1}, \ldots, \hat{P_n}\right)[\alpha := \tau]_\Delta &= \left(\hat{P_1}[\alpha := \tau]_\Delta,\ \ldots,\ \hat{P_n}[\alpha := \tau]_\Delta\right)
\end{aligned}
\tag{6}
$$

2.4 Lifetime relation

The analysis computes a relation on the lifetimes of region variables in a program. We express this relation using $\geq$, which means "lives at least as long as". For brevity we will however pronounce $\rho_1 \geq \rho_2$ as "ρ_1 outlives ρ_2" instead of "ρ_1 lives at least as long as ρ_2".

Given a relation $\mathcal{R}$, we write $x \leq_\mathcal{R} y$ for $(x, y) \in \mathcal{R}$, $x \geq_\mathcal{R} y$ for $(y, x) \in \mathcal{R}$ and $x \equiv_\mathcal{R} y$ for $(x, y) \in \mathcal{R} \ \wedge\ (y, x) \in \mathcal{R}$.

Definition 2.1 (Lifetime relation). A relation $\mathcal{R} \subset P' \times P'$ is a lifetime relation on set P' with $\{\rho_\bot, \rho_\text{global}\} \subset P' \subset P$ if the following holds:

- $\mathcal{R}$ is reflexive and transitive (i.e. a preorder [21])
- For all $\rho \in P'$, $\rho_\bot \leq_\mathcal{R} \rho$ and $\rho \leq_\mathcal{R} \rho_\text{global}$

Definition 2.2 ($\sqsupseteq$, partial order on lifetime relations). We say that a lifetime relation $\mathcal{R}_1$ is at least as precise (or constrained) as $\mathcal{R}_2$ and write $\mathcal{R}_1 \sqsupseteq \mathcal{R}_2$ (and $\mathcal{R}_2 \sqsubseteq \mathcal{R}_1$) if for all ρ_1, ρ_2, $\rho_1 \geq_{\mathcal{R}_2} \rho_2$ implies $\rho_1 \geq_{\mathcal{R}_1} \rho_2$.

We write $[\![a \geq b, c \geq d, \ldots]\!]^*$ for the smallest lifetime relation satisfying the given constraints, namely the transitive closure [21] of these constraints with ρ_global added as top element and $\rho_\bot$ as bottom. We write $\mathcal{R}_1 \sqcup \mathcal{R}_2$ for the smallest lifetime relation $\mathcal{R}$ such that $\mathcal{R} \sqsupseteq \mathcal{R}_1$ and $\mathcal{R} \sqsupseteq \mathcal{R}_2$. This is the transitive closure of $\mathcal{R}_1 \cup \mathcal{R}_2$.

2.5 Containment

Containment means intuitively that the fields of an object outlive the object. For instance for a tuple, the elements should live at least

as long as the tuple. We formalize this with a containment function $C_\Delta : \tau \times \hat{P} \to R$, taking the type and regions of a value and returning a lifetime relation on those regions. The type should be of kind $*$ and the region variables should have sort $P_\Delta(\tau)$.

$$\begin{aligned} C_\Delta(!\tau, (\rho_2, \hat{\rho}_3)) &= C^\circ_\Delta(\tau, \rho_2, \hat{\rho}_3) \\ C_\Delta(\tau, (\rho_1, \rho_2, \hat{\rho}_3)) &= [\![\rho_2 \geq \rho_1]\!] \sqcup C^\circ_\Delta(\tau, \rho_2, \hat{\rho}_3) \end{aligned} \tag{7}$$

For non-strict values, we require that the region in which the value is stored outlives the region containing the thunk. This prevents that an evaluated thunk points at a deallocated value.

Note that $\hat{\rho}_3$ may not just be a single region, but may represent an empty list (for types without additional region variables) or a list of region variables. The constraints between these regions and the region of the value (ρ_2) are specified in $C^\circ_\Delta : \tau \times P \times \hat{P} \to R$. First we consider tuples. A tuple has three region variables for each field, namely a region to store the thunk, one for the value and one for nested fields of the element. The last of these can itself be a list of multiple region variables. We require that the region containing the thunk outlives the region of the tuple itself and that the region of the value outlives the thunk. Furthermore, the value in the thunk may have more region variables in ρ_3, which can give more constraints.

$$\begin{aligned} &C^\circ_\Delta((\tau_1, \tau_2), \rho, (\rho_{1t}, \rho_{1v}, \hat{\rho}_1, \rho_{2t}, \rho_{2v}, \hat{\rho}_2)) \\ &\quad = [\![\rho_{1v} \geq \rho_{1t}, \rho_{1t} \geq \rho, \rho_{2v} \geq \rho_{2t}, \rho_{2t} \geq \rho]\!]^* \\ &\qquad \sqcup C^\circ_\Delta(\tau_1, \rho_{1v}, \hat{\rho}_1) \sqcup C^\circ_\Delta(\tau_2, \rho_{2v}, \hat{\rho}_2) \end{aligned} \tag{8}$$

$$C^\circ_\Delta(\alpha\ \tau_1 \ldots \tau_2, \rho, \rho_1) = [\![\rho_1 \geq \rho]\!] \tag{9}$$

$$C^\circ_\Delta(\tau, \rho, ()) = \bot \tag{10}$$

Containment is needed to enforce that there are no pointers to deallocated objects. When combining regions with garbage collection, this is required to allow the garbage collector to trace the heap [11]. If one wouldn't enforce containment, you could for instance allow that the items of a list are deallocated if the list is only used to compute its length.

3 ANNOTATIONS

Besides regions, we must now also assign annotations to variables. The language of annotations is a functional language based on the typed lambda calculus. An annotation of a function will take the regions of the arguments and return value and give the relation between those regions. In such an annotation, the region variables of arguments and the returned value are all distinct and the relations between those regions are stored in a lifetime relation (section 2.4).

As an example, consider the following functions, both of type `!Int → Int`:

```
id x = x
inc x = x + 1
```

Assume that the argument has region ρ_1 and the result ρ_2. Function `id` returns its argument and hence you may expect that regions ρ_1 and ρ_2 should be the same. We however only require that the region of the argument ρ_1 lives at least as long as the return region ρ_2, resulting in the lifetime relation $[\![\rho_1 \geq \rho_2]\!]$. In the second example, `inc`, we return a new object and hence there are no constraints between ρ_1 and ρ_2.

Similar to region variables of the argument, the annotation function also takes the annotations of the argument. Whereas region arguments are concrete in the sense that they directly represent the region in which a value is stored at runtime, annotation variables are more abstract as they describe the relations between region variables when a higher order argument is applied. To be more precise, annotations describe relations between the regions of the arguments and the return value of a function.

3.1 Sorts

As the annotation language is a typed functional language, we will start by introducing those types. To avoid confusion with types in the source language, we will call the types of annotations *sorts*.

$$\begin{aligned} s &::= R \mid (\ \overline{s}\ ) && \textit{Relation, Tuple} \\ &\mid \left[s; \hat{P}\right]_l \to s && \textit{Function} \\ &\mid \forall\alpha.\ s \mid \Psi\langle\ \alpha\ \overline{\tau}\ \rangle && \textit{Quantification, Polymorphic sort} \\ l &::= \mathbin{\uparrow} \mid \emptyset \end{aligned} \tag{11}$$

A relation, R, is a lifetime relation on regions. A tuple sort is used for types with multiple annotations, similar to the tuple used in region sorts.

The function sort is used to retrieve the regions and annotations of an argument or return value of a function (in the source language). We may limit how region arguments are used in a function. We annotate region arguments with a lifetime context, denoted by l. The lifetime context $\mathbin{\uparrow}$, pronounced *local bottom*, denotes that those argument may only be used on the right hand side of an outlives constraint ($\geq$) or in applications to functions with lifetime context $\mathbin{\uparrow}$. Thus, the function may not extend on the lifetime of those region variables, but these region variable can be used to extend the lifetimes of other region variables. When we do not annotate the region argument, it may be used on both sides in the lifetime relation. We write l for the lifetime context, $l ::= \mathbin{\uparrow} \mid \emptyset$. This lifetime context only applies to the region arguments of an annotation function.

The quantification sort is used for polymorphic values. Note that this quantifies over types instead of sorts. Similar to region sorts, we also have a polymorphic sort here. This sort will be replaced when its type variable is instantiated. We write $s\ \tau$ to denote the instantiation of a quantification, which we will later formalize in section 3.5.5, and $s.n$ to denote a one-based projection, where s must be a tuple sort of at least n elements. A projection $(s_1, s_2, \ldots, s_n).k$ simplifies to s_k.

3.2 Sort assignment

Similarly to region sorts, we define a mapping $\Psi_\Delta : \tau \to s$ from types to sorts. We annotate a function type $\tau_1 \to \tau_2$ with a function taking the annotations and regions of τ_1 and returning a pair containing the effect of a call to this function and the annotation of the returned value. The first has sort $[(), P] \to [(), P_\Delta(!\tau_2)]_{\mathbin{\uparrow}} \to R$ and is a function taking the region of the previous thunk, used for partial applications as described below, and the region variables of the return value. The function returns the constraints on those return regions in terms of the other regions.

$$\Psi_\Delta(\tau_1 \to \tau_2) = [\Psi_\Delta(\tau_1); P_\Delta(\tau_1)] \to ([(); P] \to [(); P_\Delta(!\tau_2)]_{\uparrow} \to R,\ \Psi_\Delta(\tau_2)) \tag{12}$$

For other types, we propagate the annotations on nested functions. For instance, for a tuple type, we create a tuple with the annotations of the elements.

$$\begin{aligned} \Psi_\Delta(()) &= () & \Psi_\Delta((\tau_1, \ldots, \tau_n)) &= (\Psi_\Delta(\tau_1), \ldots, \Psi_\Delta(\tau_n)) \\ \Psi_\Delta(!\tau) &= \Psi_\Delta(\tau) & \Psi_\Delta(\alpha\ \tau_1 \ldots \tau_n) &= \Psi\langle\ \alpha\ \tau_1 \ldots \tau_n\ \rangle \\ & & \Psi_\Delta(\forall\alpha.\ \tau) &= \forall\alpha.\ \Psi_\Delta(\tau) \end{aligned} \tag{13}$$

3.2.1 Partial applications. When partially applying a function, a thunk object is constructed containing a pointer to the function and its arguments. Thunks may form a linked list when applying a variable containing a partially applied function. As an example, consider the following code:

```
f x y z = x + y + z
a = f 1
b = a 2
```

Variable b contains a thunk object, pointing at a. We thus need a containment constraint, saying that the region of thunk a outlives the region of thunk b. The annotation function thus needs to get the region of the previous thunk as an argument. We pass this region variable in an additional lambda, with as argument $[(), P]$, as seen in the sort of a function in equation 12.

Note that the argument for the region of the previous thunk is only used for partial applications. When partially applying a function with the first argument, as we do in the definition of a, it points at a global function.

3.3 Annotation language

The analysis works with annotations from a language based on the typed lambda calculus. The syntax of the annotation language is shown in figure 2. Bottom, $\bot$, means that there are no constraints on the argument and return regions of some function and $\top$ implies that everything is stored in the global region.

A lambda or an abstraction is written as $\lambda\left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a$. It introduces both region variables ($\hat{\rho}$) and an annotation variable ψ in body a. The abstraction is annotated by a lifetime context, the same as the lifetime context on a function sort. Lambdas can be applied with an application, written as $a\ [a; \hat{\rho}]_l$.

A relation, written as a list of outlive constraints within brackets, describes a relation on the lifetimes of region variables. The constraints in the list must be unique and transitively closed. Reflexivity and constraints of the form $\rho_{\text{global}} \geq \rho$ or $\rho \geq \rho_\bot$ are implicit and should not be listed. For brevity, we may also write $\hat{\rho}_1 \geq \hat{\rho}_2$ to denote a pairwise outlives relation, e.g. $[\![(\rho_1, \rho_2) \geq (\rho_3, \rho_4)]\!]$ means $[\![\rho_1 \geq \rho_3, \rho_2 \geq \rho_4]\!]$.

To accommodate polymorphism, we need quantification ($\forall\alpha.\ a$) and instantiation $a\ \{\tau\}$. Instantiating an annotation will cause that polymorphic region sorts and polymorphic annotation sorts of the instantiated type variable are instantiated with the given type.

The language has tuples, as some types need multiple annotations. Tuples are represented as a comma-separated list of annotations. To extract elements out of a tuple, we have (one-based) projection, $a.n$.

Furthermore, we have a join operator, which is the least annotation (with respect to the partial order defined below) greater than both of its arguments.

In section 2.4, we defined a partial order on lifetime relations, e.g. annotations of sort R. We extend the definition to annotations of arbitrary sorts with the usual extension [8, 22], written as $a \sqsubseteq a'$. We write $a = a'$ when $a \sqsubseteq a'$ and $a' \sqsubseteq a$.

3.4 Sorting rules

The sorting rules are the "typing rules" of the annotation language and describe the sorts of annotations. We write $\Delta; \Sigma \vdash a : s$ to denote that annotation a has sort s in data type environment Δ and sort environment Σ, which contains the sorts of the annotation and region variables in scope. It is a set with elements of the form $\psi : s$, $\rho : P^l$ or $\rho : P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle^l$, respectively bindings of annotation variables, monomorphic region variables and polymorphic region variables. The l denotes the lifetime context of the lambda which introduced those region variables. We can only introduce region variables of a monomorphic sort or a polymorphic sort; region variables of a tuple sort are not allowed; tuples should be bound to a multiple region variables. We will however write $\{\hat{\rho} : \hat{P}^l\}$ which should be read pairwise, e.g. $\{(\rho_1, \rho_2) : (P_1, P_2)\}$ denotes $\{\rho_1 : \hat{P}, \rho_2 : P\}$. We assume there is no shadowing of variables.

The sorting rules are given in figure 3. In the rule for quantification, $\text{ftv}(\Sigma)$ denotes the free type variables in Σ. Most rules are as you would expect for a functional language, we highlight some nonstandard rules here. For region variables, we have subeffecting. It is allowed to use a variable with any lifetime context as having lifetime context local bottom, and we may use a monomorphic region variable ($\rho : P^l$) as a polymorpic variable or as a tuple ($\rho : \hat{P}^l$). Note that in the notation that we use, P is syntax (the monomorphic region sort) and $\hat{P}$ is variable (denoting any region sort).

A lifetime relation consists of outlive constraints of the form $\rho \geq \rho'$. For each outlive constraint we require that if both operands are polymorphic, then they should be polymorphic on the same type. Furthermore, the left hand side must have lifetime context any and cannot be local bottom. The sorting rule for a lifetime relation with multiple constraints says that all constraints on their own should be valid, and it should be a lifetime relation according to definition 2.1.

Figure 2: Language of annotations

$a ::= \bot \mid \top$	*Bottom, Top*
$\mid \lambda\left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a$	*Abstraction*
$\mid \psi \mid a\ [a; \hat{\rho}]_l$	*Variable, Application*
$\mid a \sqcup a \mid [\![\ \overline{\rho \geq \rho}\ ]\!]$	*Join, Relation*
$\mid \forall\alpha.\ a \mid a\ \{\tau\}$	*Quantification, Instantiation*
$\mid (\ \overline{a}\ ) \mid a.n$	*Tuple, Projection*

Figure 3: Sorting rules of annotations

$$\frac{\psi : s \in \Sigma}{\Delta;\Sigma \vdash \psi : s} \quad \frac{\rho : \hat{P}^l \in \Sigma}{\Delta;\Sigma \vdash \rho : \hat{P}^l} \quad \frac{\Delta;\Sigma \vdash \rho : \hat{P}}{\Delta;\Sigma \vdash \rho : \hat{P}^{\uparrow}} \quad \frac{\Delta;\Sigma \vdash \rho : P^l}{\Delta;\Sigma \vdash \rho : \hat{P}^l} \quad \frac{}{\Delta;\Sigma \vdash \bot : s} \quad \frac{}{\Delta;\Sigma \vdash \top : s} \quad \frac{\Delta;\Sigma \vdash a : s \quad \alpha \notin \mathrm{ftv}(\Sigma)}{\Delta;\Sigma \vdash \forall\alpha.\, a : \forall\alpha.\, s}$$

$$\frac{\Delta;\Sigma \vdash \hat{\rho}_1 : \hat{P}_1 \quad \ldots \quad \Delta;\Sigma \vdash \hat{\rho}_n : \hat{P}_n}{\Delta;\Sigma \vdash (\hat{\rho}_1, \ldots, \hat{\rho}_n) : \left(\hat{P}_1, \ldots, \hat{P}_n\right)} \quad \frac{\Delta;\Sigma \cup \{\psi : s_1\} \cup \{\hat{\rho} : \hat{P}^l\} \vdash a : s_2}{\Delta;\Sigma \vdash \lambda\left[\psi : s_1; \hat{\rho} : \hat{P}\right]_l \mapsto a : \left[s_1; \hat{P}\right] \to s_2} \quad \frac{\Delta;\Sigma \vdash a_1 : \left[s_1; \hat{P}\right]_l \to s_2 \quad \Delta;\Sigma \vdash a_2 : s_1 \quad \Delta;\Sigma \vdash \hat{\rho} : \hat{P}^l}{\Delta;\Sigma \vdash a_1\ [a_2; \hat{\rho}]_l : s_2}$$

$$\frac{\Delta;\Sigma \vdash \rho : P \quad \Delta;\Sigma \vdash \rho' : P^l}{\Delta;\Sigma \vdash [\![\rho \geq \rho']\!] : R} \quad \frac{\Delta;\Sigma \vdash [\![\rho_1 \geq \rho_1']\!] : R \quad \ldots \quad \Delta;\Sigma \vdash [\![\rho_n \geq \rho_n']\!] : R \quad \text{The relation is a lifetime relation}}{\Delta;\Sigma \vdash [\![\rho_1 \geq \rho_1', \ldots, \rho_n \geq \rho_n']\!] : R}$$

$$\frac{\Delta;\Sigma \vdash \rho : P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle \quad \Delta;\Sigma \vdash \rho' : P^l}{\Delta;\Sigma \vdash [\![\rho \geq \rho']\!] : R} \quad \frac{\Delta;\Sigma \vdash \rho : P \quad \Delta;\Sigma \vdash \rho' : P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle^l}{\Delta;\Sigma \vdash [\![\rho \geq \rho']\!] : R} \quad \frac{\Delta;\Sigma \vdash \rho : P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle \quad \Delta;\Sigma \vdash \rho' : P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle^l}{\Delta;\Sigma \vdash [\![\rho \geq \rho']\!] : R}$$

$$\frac{\Delta;\Sigma \vdash a : \forall\alpha.\, s}{\Delta;\Sigma \vdash a\ \{\ \tau\ \} : s[\alpha := \tau]_\Delta} \quad \frac{\Delta;\Sigma \vdash a_1 : s_1 \quad \Delta;\Sigma \vdash a_n : s_n}{\Delta;\Sigma \vdash (a_1, \ldots, a_n) : (s_1, \ldots, s_n)} \quad \frac{\Delta;\Sigma \vdash a : (s_1, \ldots, s_n) \quad 1 \leq k \leq n}{\Delta;\Sigma \vdash a.k : s_k} \quad \frac{\Delta;\Sigma \vdash a_1 : s \quad \Delta;\Sigma \vdash a_2 : s}{\Delta;\Sigma \vdash a_1 \sqcup a_2 : s}$$

3.5 Evaluation rules

We will introduce the evaluation rules of the annotation language in this section, ignoring name collisions and alpha conversion. We write $a \longrightarrow a'$ to denote that a well sorted annotation a evaluates to a'. The "evaluates to" relation is reflexive and transitive.

$$\frac{}{a \longrightarrow a} \quad \frac{a_1 \longrightarrow a_2 \quad a_2 \longrightarrow a_3}{a_1 \longrightarrow a_3}$$

3.5.1 Sub-annotations. The subterms of annotations may be evaluated.

$$\frac{a \longrightarrow a'}{\lambda\left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a \longrightarrow \lambda\left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a'}$$

$$\frac{a_1 \longrightarrow a_1' \quad a_2 \longrightarrow a_2'}{a_1\ [a_2; \hat{\rho}]_l \longrightarrow a_1'\ \left[a_2'; \hat{\rho}\right]_l} \quad \frac{a \longrightarrow a'}{\forall\alpha.\, a \longrightarrow \forall\alpha.\, a'}$$

$$\frac{a_1 \longrightarrow a_1' \quad a_n \longrightarrow a_n'}{(a_1, \ldots, a_n) \longrightarrow (a_1', \ldots, a_n')} \quad \frac{a \longrightarrow a'}{a\ \{\tau\} \longrightarrow a'\ \{\tau\}}$$

$$\frac{a \longrightarrow a'}{a.i \longrightarrow a'.i} \quad \frac{a_1 \longrightarrow a_1' \quad a_2 \longrightarrow a_2'}{a_1 \sqcup a_2 \longrightarrow a_1' \sqcup a_2'}$$

3.5.2 Tuples. A projection extracts an element from a tuple.

$$\frac{}{(a_1, \ldots, a_n).i) \longrightarrow a_i}$$

3.5.3 Join. A join of two lifetime relations evaluates to the smallest lifetime relation containing both operands.

$$\frac{\mathcal{R} = [\![\rho_1 \geq \rho_1', \ldots, \rho_n \geq \rho_n', \rho_{n+1} \geq \rho_{n+1}', \ldots, \rho_m \geq \rho_m']\!]^*}{[\![\rho_1 \geq \rho_1', \ldots, \rho_n \geq \rho_n']\!] \ \sqcup\ [\![\rho_{n+1} \geq \rho_{n+1}', \ldots, \rho_m \geq \rho_m']\!] \longrightarrow \mathcal{R}}$$

The join has bottom as identity and is associative, commutative and idempotent.

$$\frac{}{\bot \sqcup a \longrightarrow a} \quad \frac{}{(a_1 \sqcup a_2) \sqcup a_3 \longrightarrow a_1 \sqcup (a_2 \sqcup a_3)}$$

$$\frac{}{\top \sqcup a \longrightarrow \top} \quad \frac{}{a_1 \sqcup a_2 \longrightarrow a_2 \sqcup a_1} \quad \frac{}{a \sqcup a \longrightarrow a}$$

Furthermore, we have some rules on how joins and bottoms distribute over other annotations. For quantifications and lambdas, we move the join inward. With alpha conversion, we can assure that the type variable, region variables and annotation variable have the same name on both sides.

$$\frac{}{(\forall\alpha.\, a_1) \sqcup (\forall\alpha.\, a_2) \longrightarrow \forall\alpha.\, a_1 \sqcup a_2}$$

$$\frac{}{(\lambda\left[\psi; \hat{\rho}\right]_l \mapsto a_1) \sqcup (\lambda\left[\psi; \hat{\rho}\right]_l \mapsto a_2) \longrightarrow \lambda\left[\psi; \hat{\rho}\right]_l \mapsto a_1 \sqcup a_2}$$

$$\frac{}{\forall\alpha.\, \bot \longrightarrow \bot} \quad \frac{}{\lambda\left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto \bot \longrightarrow \bot}$$

For applications and instantiations, we move the joins outward.

$$\frac{}{(a_1 \sqcup a_2)\ [a, \hat{\rho}]_l \longrightarrow a_1\ [a, \hat{\rho}]_l\ \sqcup\ a_2\ [a, \hat{\rho}]_l} \quad \frac{}{\bot\ [a, \hat{\rho}]_l \longrightarrow \bot}$$

$$\frac{}{(a_1 \sqcup a_2)\ \{\tau\} \longrightarrow a_1\ \{\tau\}\ \sqcup\ a_2\ \{\tau\}} \quad \frac{}{\bot\ \{\tau\} \longrightarrow \bot}$$

3.5.4 Application. An application can be evaluated if it targets an abstraction, with a substitution.

$$\frac{}{\left(\lambda\left[\psi : s; \hat{\rho}_1 : \hat{P}\right]_l \mapsto a_1\right)\ [a_2; \hat{\rho}_2]_l \longrightarrow a_1[\psi := a_2][\hat{\rho}_1 := \hat{\rho}_2]}$$

The substitution of the annotation argument ψ is a syntactic substitution (ignoring name collisions). The substitution of region variables is slightly different as a lambda takes a tree of region variables. As noted before, a single region ρ may also be passed to a tree of region variables. The constraints in a lifetime relation must after substitution be converted to a lifetime relation again, as there might be new transitively implied constraints after the substitution.

3.5.5 Instantiation. An instantiation of a quantification, $(\forall\alpha.\, a)\ \{\tau\}$ and its sort $(\forall\alpha.\, s)\ \tau$, can be evaluated by substituting type variable α with type τ in a and s. A polymorphic annotation sort $\Psi\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle$ is substituted with $\Psi_\Delta(\alpha\ \tau_1\ \ldots\ \tau_n)$ and $P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle$ with $P^\circ_\Delta(\alpha\ \tau_1\ \ldots\ \tau_n)$. A polymorphic region variable, bound by a lambda, may be replaced by a structure of region variables.

3.5.6 Normal form. If an annotation cannot be evaluated further, we say that it is in normal form. As the join is commutative and associative, we enforce an arbitrary order on annotations and order the annotations in a join accordingly. Furthermore variables must be saturated, which we ensure by eta expansion.

4 BINDING REGIONS WITH ESCAPE CHECK

So far we have described how analysis values can be represented in our setup, but we haven't shown how those terms are computed. To decide where regions are bound, we add the *escape check*, which does the actual analysis-specific work by detecting which regions can be allocated within the function and which regions may be unified with others. After this section, we explain how the constructs in the source language are mapped to annotation terms in the annotation language.

4.1 Additional region arguments

When analysing a toplevel definition, we get access to the regions of arguments and the return value and the global region, and we can allocate new local regions. In some higher order functions, this is not sufficient. In these cases, we add additional region arguments to the function. For example, in the `(.)` operator in Haskell for function composition, we add an additional region argument for the intermediate value (the result of one function, to be passed to the second). In case of `(inc . id) x`, that additional region argument may be instantiated with the same region as the input `x`, and in case of `(id . inc) x` it can be instantiated with the same region as the result of the entire expression.

The analysis starts by assigning region variables to the local bindings of the function. The *escape check* will then decide for each of those regions, whether it can be unified with another region or can be allocated within the function. If neither of those options happens, then that region becomes an additional region argument.

We must adapt the run-time representation of thunks to support additional region arguments. A thunk is a linked list of arguments, where the last item points at the function being called. We now must change this to be an object containing both the function pointer and the additional region arguments. For partial applications we would already allocate an object for the thunk, but when using a toplevel function with additional region arguments without passing any arguments this change does have a larger impact: without region based memory management, using a toplevel function would not result in any allocation, but now we must allocate an object containing the function pointer and additional region arguments. This is a tradeoff; using additional region arguments we can instantiate the regions differently for the different uses of the function. Instead we could also remove the additional region arguments and replace them by the global region, which eliminates the need for this additional allocation, but reduces the precision of the analysis.

4.2 Escape check

The analysis performs the *escape check* to decide which regions may be allocated within a function or may be unified with others. It operates on an annotation, which we expect to be a function originating from a source function of a certain arity n. We take the lifetime relation of the subannotation corresponding with the effect after being applied with n arguments (i.e., we drop the subannotations corresponding with applications of fewer than n arguments) and also check whether region variables occur elsewhere in the effect or in the annotation of the returned value. If a region variable is used elsewhere, except in an application with lifetime context local bottom, we call it a higher order region variable as this is caused by a usage of an function argument of a higher order function. Occurrences of region variables in applications with lifetime context local bottom ($\updownarrow$) may be ignored, as such applications cannot extend the lifetime of those regions; those applications can only only extend the lifetime of other regions in terms of those. We cannot default higher order region variables, as we may get more constraints on them if the annotation of the higher order function argument is given.

Before we introduce the defaulting rules which we use in the escape check, we must first introduce the term *directly outlives*.

Definition 4.1 (Outlive set). The outlive set $outlive(\rho_u)$ of a region ρ_u in a lifetime relation $\mathcal{R}$ is the set of all regions $\rho_v \neq \rho_u$ such that $\rho_u \geq \rho_v$.

Definition 4.2 (Directly outlives). We say that region ρ_u directly outlives region ρ_v if $outlive(\rho_u) = outlive(\rho_v) \cup \{v\}$.

A region directly outlives at most one region. In first order functions, we can find regions which directly outlive an other region by looking at the relation specified in the normalized annotation. For higher order functions, we do not yet know all constraints, as some regions, the higher order region variables, may be influenced by the annotations of the arguments. The escape check performs the following rules:

- If an additional region argument ρ does not occur in the form $\rho \geq \rho'$ and is not a higher order region variable, then no constraints require this region to be live outside of this function. We allocate this region within the function, substitute it with $\rho_\perp$ in the remaining constraints and keep applying this rule until no further regions can be allocated within the function.
- If two regions ρ_1 and ρ_2 exist with the constraints $\rho_1 \geq \rho_2$ and $\rho_2 \geq \rho_1$, at least one of them is an additional region argument and they have the same region sort, then we can unify them.
- Finally we substitute a region ρ_1 with ρ_2 if ρ_1 is an additional region argument, not a higher order region variable, they have the same region sort and ρ_1 directly outlives ρ_2. This rule is applied repeatedly.

Note that the second rule can be applied to higher order region variables, as additional constraints cannot cause the constraints $\rho_1 \geq \rho_2$ and $\rho_2 \geq \rho_1$ to disappear. Also note that because we don't need to consider applications with lifetime context local bottom, we can detect that the `Maybe` objects in `mapMaybe :: (a -> Maybe b) -> [a] -> [b]` can be allocated within the function: even though we do not know what the function does to construct those `Maybe` values, we still know that they cannot escape `mapMaybe`.

5 FIXED POINT ITERATION FOR (MUTUAL) RECURSION

As the source language contains recursive definitions, we must support them in the analysis. In type inference, also in inference for type and effect systems, we would assign an type/annotation variable to the recursively used declaration and refine that using unifications. As our annotation language is richer, we cannot use unifications. We use fixed-point iteration, where we initially assign bottom (the lowest value in the lattice) as the annotation value to the recursively used declaration. We iterate by using the resulting annotation as the annotation of that declaration for the next iteration, until we reach a fixed-point.

The source language has two ways to introduce (mutual) recursive declarations: recursive functions and recursive let-bindings. Our implementation actually has a third form, as we apply the analysis in our implementation to an intermediate language with SSA [1, 7], which may contain loops after the tail call optimization pass. Those recursive bindings are handled the same as the recursive let-bindings.

In our domain, region inference, we need to perform the escape check which will bind and unify regions. Instead of introducing the escape check as a separate construct in the annotation language, we need to combine it with fixed-point iteration: The escape check is not monotone, whereas fixed-point iteration requires a monotonicity. Furthermore, the escape check may find that some regions do not escape the lifetime of a function, and by performing the escape check within the fixed-point iteration (as opposed to first finding a fixed-point and then running the escape check) we allow the region to be allocated within the function and not have any constraints with the recursive calls. We extend the annotation language with two fixed-point combinators:

$$a ::= \ldots \mid \text{fix } s.\ a \qquad \textit{Fixed-point}$$

$$\mid \text{fix escape } \{\hat{P}\} : \left(\overline{[n, s] : a}\right) \qquad \textit{Fixed-point with escape check}$$

The first fixed-point combinator, fix $s.\ a$, evaluates to the least annotation a' of sort s such that $a\ a' = a'$, if we can reach that fixed-point. We will later discuss how we can handle functions for which we do not reach a fixed-point (or cannot detect that we are in a fixed-point) in a certain number of iterations.

The second fixed-point combinator includes the escapes check. In contrary to previous higher ranked analyses, we must now couple the fixed-point iteration with the escape check to unify and remove region variables, as the transformation in the escape check is not monotone. A fixed-point with escape check, written as fix *escape* $\{\hat{P}\} : ([n_1, s_1] : a_1, \ldots, [n_k, s_k] : a_k)$, has sort $(\hat{P} \to s_1, \ldots, \hat{P} \to s_n)$. Annotation a_i should have sort $[(s_1, \ldots, s_n), ()] \to [(); \hat{P}] \to s_i$ and should correspond with a function type of arity at least n_i. The fixed-point evaluates to a tuple $(a'_1.0, \ldots, a'_k.0)$ such that for all $i \in \{1, \ldots, k\}$, a'_i is equal to the result of the escape check applied to $a_i(a'_i)$, if we can find such a value.

We use this to analyse a group of k functions where function i has annotation a_i, arity n_i and sort s_i. The annotation and sort are a tuple: to handle recursive let bindings, we must store an annotation per variable together with the actual annotation of the analysed function in the tuple.

5.1 Evaluation

A fixed-point fix $s.\ a$, where $a : [s; ()] \to s$. is evaluated by repeatedly applying a starting with $\bot$, until a fixed-point is reached. The evaluation of a fixed-point with escape check is similar, but we will try to remove additional region arguments there, as described in section 4.2. A fixed-point with escape check can only be evaluated if the body has no free variables, as free variables could cause the escape check to be inconsistent.

A fixed-point can sometimes be detected by checking syntactic equivalence after evaluating the terms. As shown previously in [22], this is not sufficient as there exist pairs of normalized functions with the same behaviour but a different notation. If the source language is monomorphic, and we thus do not have a polymorphic sort, we can detect equivalence of two functions by applying them with all possible inputs of the correct sort. The set of possible inputs is finite; it may be very large but it is at least possible. If this source language does have polymorphism, then this set can become infinite: there might be infinitely many types with which we can instantiate a polymorphic annotation. The detection of a fixed-point is not the only problem, some functions also do not have a fixed-point.

As an example, consider the function `iterate :: (a -> a) -> a -> [a]` which repeatedly applies the function and returns the values in an infinite list. If the function passed to `iterate` rotates a tuple of size n, i.e. shifts all elements of a tuple one place and places the last at the beginning, then we would need to perform at least n iterations in fixed-point iteration to make sure that the annotation of the first element of the tuple is propagated to all positions in the tuple. The function is however analysed before we know how it is used and how type argument `a` is instantiated, and we can choose an arbitrarily large n. This function does not have a least fixed-point, and we have considered various options to handle those functions. We could simply use $\top$ as annotation if a function does not have a fixed-point or if the fixed-point cannot be found. We could also perform fixed-point iteration starting at top: the annotation after any iteration is admissible, and hence we can stop iterating at any time. This does cause that the analysis loses a lot of precision.

Another option would be to just keep the fixed-point combinator in the annotation, to be evaluated later when more information about the use of the function is known. When the function is used, we may know the type with which the type argument is instantiated or the function passed to the higher order function, and be able to evaluate the fixed-point. We experimented with this, but it becomes difficult to make use of the information later on. Furthermore, we do need to perform code generation on the higher order function, before we know it's used: we can keep the fixed-point combinator in the annotation, but we still need to know which regions need to be placed where. Hence we do need to evaluate a fixed-point with escape check to a normal fixed-point combinator, which can be done by basing the escape check on some steps of fixed-point iteration starting at top. It is helpful to implement additional evaluation rules, for instance detecting whether an argument is invariant (always passed the same value in calls in the fixed-point) and removing the invariant argument, to evaluate more fixed-points.

6 INFERENCE

We write $\Delta, \Gamma, \hat{\rho} \vdash e : a_{\text{effect}}, a_{\text{result}}$ to denote that if expression e in data type environment Δ and annotation environment Γ needs to store its result in regions $\hat{\rho}$, then it causes effect a_{effect} and returns a value with annotation a_{result}.

We introduce an environment Γ mapping variables to the annotations and regions of their values. A top-level function (a toplevel binding whose arity is at least one) $x : \tau = f$ is present as $x : (\hat{P}, a)_T$, where $\hat{P}$ is the sort of the additional region arguments and $a : [(); \hat{P}] \to \Psi_\Delta(\tau)$ is the annotation of the function. A top-level constant or a local variable x of type τ is present as $x : (a, \hat{\rho})$ where $a : \Psi_\Delta(\tau)$ and $\hat{\rho} : P_\Delta(\tau)$ are its annotation and regions.

6.1 Top-level definitions

We write $\Delta, \Gamma \vdash f : a$ to denote that a top-level declaration f has annotation a, in data type environment Δ and annotation environment Γ. Top-level definitions are always stored in the global region. In the analysis we distinguish top-level functions and top-level constants. Recall that lambdas only occur in top-level definitions, not in expressions. This format isn't restrictive, as a program can for instance be converted into it with lambda lifting [15]. We call a top-level definition with lambdas, i.e. arity$(f) \geq 1$, a function, and one without a constant.

6.1.1 Functions. Rule Function annotates a toplevel function $x : \tau = f$ with an annotation $a : \Psi_\Delta(\tau)$, written as $\Delta, \Gamma \vdash_{\text{function}} f : a$. As defined in Ψ_Δ, the annotation on a function type contains the effect of calling the lambda and the annotation on the result. We call the rightmost lambda in a function the *saturated* lambda and the others *unsaturated*. The effect and annotation of a saturated lambda follow directly from the expression in the body (rule Saturated Lambda). Rule Unsaturated Lambda places the containment of the thunk object (or closure) that would be created for an unsaturated call as the effect: the thunk refers to the previous thunk (the applied function) and the arguments of the application. We show the inference rules for functions in figure 4.

6.1.2 Constants. A top-level constant $x : \tau = f$ (where arity$(f) = 0$) is annotated with both an effect $a_{\text{effect}} : R$ and an annotation on the value $a_{\text{result}} : \Psi_\Delta(\tau)$, written as $\Delta, \Gamma \vdash_{\text{constant}} f : a_{\text{effect}}, a_{\text{result}}$. The effect is not needed for usages of this toplevel definition, but it is necessary to analyse and compile the code for the top-level as well: Can local variables of the declaration for instance be allocated locally or should they be placed in the global region? Later on, we will strip the effect away in function $\mathcal{N}$.

$$\frac{\Delta, \Gamma \vdash_{\text{constant}} f : a_{\text{effect}}, a_{\text{result}}}{\Delta, \Gamma \vdash f : (a_{\text{effect}}, a_{\text{result}})} \text{ [Constant]}$$

$$\frac{\Delta, \Gamma \vdash_{\text{constant}} f : a_{\text{effect}}, a_{\text{result}}}{\Delta, \Gamma \vdash_{\text{constant}} \forall\alpha.\, f : a_{\text{effect}}, \forall\alpha.\, a_{\text{result}})} \text{ [C. Quantification]}$$

$$\frac{\Delta, \Gamma, \rho_{\text{global}} \vdash e : a_{\text{effect}}, a_{\text{result}}}{\Delta, \Gamma \vdash_{\text{constant}} e : a_{\text{effect}}, a_{\text{result}}} \text{ [Constant Body]}$$

6.2 Binding groups

We introduce a judgement $\Delta, \Gamma \vdash \{x_1 : \tau_1 = f_1; \ldots\ x_n : \tau_n = f_n\} : \Gamma'$ to denote that if the program with bindings $x_1, \ldots, x_n$ with all free variables present in Γ is analysed, then $\Gamma' \supset \Gamma$ adds the annotations of $x_1, \ldots, x_n$ to Γ. Note contrary to other type (and effect) systems, Γ does not contain bindings for the introduced indentifiers $x_1, \ldots, x_n$. Instead, this judgement tells how those bindings extend the environment Γ to Γ'. Rule Split can split a program in binding groups, clusters of (mutually) recursive functions. Non-recursive functions are alone in such a group. Splitting in binding groups improves both the performance and precision of the analysis.

$$\frac{\begin{array}{c} k + m = n \quad \{b_1, \ldots, b_k\} \cup \{r_1, \ldots, r_m\} = \{1, \ldots, n\} \\ \Delta, \Gamma \vdash \{x_{b_1} : \tau_{b_1} = f_{b_1}; \ldots\ x_{b_k} : \tau_{b_k} = f_{b_k}\} : \Gamma'_b \\ \Delta, \Gamma \cup \Gamma'_b \vdash \{x_{r_1} : \tau_{r_1} = f_{r_1}; \ldots\ x_{r_k} : \tau_{r_k} = f_{r_k}\} : \Gamma' \end{array}}{\Delta, \Gamma \vdash \{x_1 : \tau_1 = f_1; \ldots\ x_n : \tau_n = f_n\} : \Gamma'} \text{ [Split]}$$

Rule Program analyses a a binding group by introducing a fixed-point with escape check. All free region variables of the annotations are bound as additional region arguments. The escape check will detect whether those regions do not escape and can be allocated locally. Free polymorphic region variables become monomorphic additional region arguments, as they may need to be passed at call sites before the type arguments are known. If the escape check unifies such a variable with another region variable, then it may become polymorphic again and we wouldn't lose precision.

For recursive calls we use the same additional region arguments: there is no polymorphic recursion on additional region arguments. If the escape check would detect that a region doesn't escape, then all recursive calls will get their own local region.

Figure 4: Inference of toplevel functions

$$\frac{\Delta, \Gamma \vdash_{\text{function}} f : a}{\Delta, \Gamma \vdash f : a} \text{ [Function]} \qquad \frac{\Delta, \Gamma \vdash_{\text{function}} f : a}{\Delta, \Gamma \vdash_{\text{function}} \forall\alpha.\, f : \forall\alpha.\, a)} \text{ [Function Quantification]}$$

$$\frac{\begin{array}{c} \psi_1, \psi_2, \psi_3, \rho_{\text{previous}}, \hat{\rho} : P_\Delta(\tau_1), \hat{\rho}_{\text{result}} : P_\Delta(\tau_2) \text{ fresh} \qquad \Delta, \Gamma \cup \{x : (\psi_1, \hat{\rho})\}, \rho_{\text{result}} \vdash e : a'_{\text{effect}}, a_{\text{result}} \\ a_{\text{effect}} = \lambda[\psi_2 : (), \rho_{\text{previous}} : P] \mapsto \lambda[\psi_3 : (), \hat{\rho}_{\text{result}} : P_\Delta(\tau_2)]_\uparrow \mapsto C_\Delta(\tau_1, \hat{\rho}) \sqcup a'_{\text{effect}} \end{array}}{\Delta, \Gamma \vdash_{\text{function}} \backslash x : \tau_1.\, e : \lambda[\psi_1 : \Psi_\Delta(\tau_1), \hat{\rho} : P_\Delta(\tau_1)] \mapsto (a_{\text{effect}}, a_{\text{result}})} \text{ [Saturated Lambda]}$$

$$\frac{\begin{array}{c} \psi_1, \psi_2, \psi_3, \rho_{\text{previous}}, \hat{\rho} : P_\Delta(\tau_1), \hat{\rho}_{\text{result}} : P_\Delta(!\tau_2) \text{ fresh} \qquad \Delta, \Gamma \cup \{x : (\psi_1, \hat{\rho})\} \vdash_{\text{function}} () : f a_{\text{result}} \\ a_{\text{effect}} = \lambda[\psi_2 : (), \rho_{\text{previous}} : P] \mapsto \lambda[\psi_3 : (), \hat{\rho}_{\text{result}} : P_\Delta(!\tau_2)]_\uparrow \mapsto C_\Delta(\tau_1, \hat{\rho}) \sqcup [\![\hat{\rho}_{\text{previous}} \geq \hat{\rho}_{\text{result}}.0]\!] \end{array}}{\Delta, \Gamma \vdash_{\text{function}} \backslash x : \tau_1.\, f : \lambda[\psi_1 : \Psi_\Delta(\tau_1), \hat{\rho} : P_\Delta(\tau_1)] \mapsto (a_{\text{effect}}, a_{\text{result}})} \text{ [Unsaturated Lambda]}$$

In case a binding is a toplevel constant, then we must convert its annotation to an annotation without the effect and without additional region arguments. This conversion happens in functions $\mathcal{N}$ and n. The formalisation is shown in figure 5.

$$\mathcal{N}_{\text{fix}}(f, \hat{\rho}_{\text{add.}}, a) = \begin{cases} ((a\ [(); \rho_{\text{global}}]).2, \rho_{\text{global}}) & \text{if arity}(f) = 0 \\ (a\ [(); \hat{\rho}_{\text{add.}}], \rho_{\text{global}}) & \text{otherwise} \end{cases}$$

$$\mathcal{N}(f, \hat{P}, a) = \begin{cases} ((a\ [(); \rho_{\text{global}}]).2, \rho_{\text{global}}) & \text{if arity}(f) = 0 \\ (\hat{P}, a)_T & \text{otherwise} \end{cases}$$

$$n(f, s) = \begin{cases} (R, s) & \text{if arity}(f) = 0 \\ s & \text{otherwise} \end{cases}$$

6.3 Expressions and calls

The rules for pattern matching on lists and non-recursive let bindings bring the new variables in scope and propagate the effects of the subterms. Together with the rule for recursive let bindings, they are shown in figure 6. We use a similar notation as for expressions for calls: $\Delta, \Gamma, \hat{\rho} \vdash_{\text{call}} c : a_{\text{effect}}, a_{\text{result}}$, but $\hat{\rho}$ may here correspond to a lazy type instead of a strict type; in contrast to expressions, calls may have a lazy type. When a call is used strictly, either in an expression or when calling a constructor (including tuple), we can forget about the thunk region with respectively Expr Call or Strict Call.

$$\frac{\Delta, \Gamma, (\rho_t, \rho_v, \hat{\rho}) \vdash_{\text{call}} c : a_{\text{effect}}, a_{\text{result}}}{\Delta, \Gamma, (\rho_v, \hat{\rho}) \vdash c : a_{\text{effect}}, a_{\text{result}}}\ [\text{Expr Call}]$$

$$\frac{\Delta, \Gamma, (\rho_t, \rho_v, \hat{\rho}) \vdash_{\text{call}} c : a_{\text{effect}}, a_{\text{result}}}{\Delta, \Gamma, (\rho_v, \hat{\rho}) \vdash_{\text{call}} c : a_{\text{effect}}, a_{\text{result}}}\ [\text{Strict Call}]$$

The constraints in rule Var1 for local variables assure that the regions of the variable outlive the destination regions. For a call to a toplevel declaration (Var2), we don't need to generate those constraints, as it is already stored in the global region. We do have to apply the annotation with additional region arguments there.

$$\frac{x : (a, \hat{\rho}_2) \in \Gamma}{\Delta, \Gamma, \hat{\rho}_1 \vdash_{\text{call}} x : [\![\hat{\rho}_2 \geq \hat{\rho}_1]\!]^*, a}\ [\text{Var1}]$$

$$\frac{x : (\hat{P}, a)_T \in \Gamma}{\Delta, \Gamma, (\rho_t, \rho_v, \hat{\rho}) \vdash_{\text{call}} x : \hat{\rho}_{\text{add}} \geq \rho_t, a\ [(); \hat{\rho}_{\text{add}}]}\ [\text{Var2}]$$

Rule Apply for applications passes the regions and annotation of the annotation, the regions for the output of the call and the region of the previous thunk to the annotation of the function. The resulting effect ($a.1$) is combined with the effect of the function expression, containment of the value and containment of the thunk,

$$\frac{\begin{array}{c} \Delta, \Gamma, \hat{\rho}_{\text{function}} \vdash_{\text{call}} c : a^{\text{function}}_{\text{effect}}, a^{\text{function}}_{\text{result}} \\ x : (a_{\text{argument}}, \hat{\rho}_{\text{argument}}) \in \Gamma \\ a = a^{\text{function}}_{\text{result}}\ [a_{\text{argument}}; \hat{\rho}_{\text{argument}}] \\ a_{\text{thunk}} = [\![(\rho_{\text{function}}.1, \hat{\rho}_{\text{argument}}) \geq \rho_t]\!] \\ \tau \text{ is the type of } c \text{ and } \hat{\rho}_{\text{function}} : P_\Gamma(\tau) \text{ fresh} \end{array}}{\begin{array}{c} \Delta, \Gamma, (\rho_t, \rho_v, \hat{\rho}) \vdash_{\text{call}} c\ x : \\ C_\Delta(\tau, \hat{\rho}_{\text{function}}) \sqcup a^{\text{function}}_{\text{effect}} \sqcup a_{\text{thunk}} \\ \sqcup\ a.1\ [(), \rho_{\text{function}}.2]\ [(), (\rho_v, \hat{\rho})]_{\uparrow}, a.2 \end{array}}\ [\text{Apply}]$$

Rule Instantiate for type applications in calls uses the same regions as the subexpression to store the value, as it is allowed to use a monomorphic region variable $\rho : P$ for any region sort $\hat{P}$.

$$\frac{\Delta, \Gamma, \hat{\rho} \vdash_{\text{call}} c : a_{\text{effect}}, a_{\text{result}}}{\Delta, \Gamma, \hat{\rho} \vdash_{\text{call}} c\ @\tau : a_{\text{effect}}, a_{\text{result}}\ \{\tau\}}\ [\text{Instantiate}]$$

Unit has no effect nor annotations. A tuple has an annotation containing all the annotations of the fields. The effect of constructing a tuple ensures that the region variables of the values stored in the tuple outlive the corresponding regions of the tuple.

$$\frac{}{\Delta, \Gamma, \hat{\rho} \vdash_{\text{call}} () : \bot, ()}\ [\text{Unit}]$$

$$\frac{}{\begin{array}{l} \Delta, \Gamma, \hat{\rho}_{\text{constructor}} \vdash_{\text{call}} (\ ,\ ) : \bot, \\ \forall \alpha_1.\ \forall \alpha_2. \\ \lambda[\psi_1 : \Psi\langle\alpha_1\rangle, (\rho^t_1, \rho^v_1, \hat{\rho}_1) : (P, P, P\langle\alpha_1\rangle)] \mapsto \\ (\bot, \lambda[\psi_2 : \Psi\langle\alpha_2\rangle, (\rho^t_2, \rho^v_2, \hat{\rho}_2) : (P, P, P\langle\alpha_2\rangle)] \mapsto \\ (\lambda[\_ : (), \_ : P] \mapsto \lambda[\_ : (), (\_, \hat{\rho}) : P_\Delta(!(\alpha_1, \alpha_2))] \\ \quad \mapsto [\![(\rho^t_1, \rho^v_1, \hat{\rho}_1, \rho^t_2, \rho^v_2, \hat{\rho}_2) \geq \hat{\rho}]\!], \\ (\psi_1, \psi_2))) \end{array}}\ [\text{Tuple-2}]$$

6.4 Data types

To add support for data types in the analysis, we must handle data types in the assignment of region variables (P°_Δ), the assignment of annotations (Ψ_Δ), containment (C) and the annotations and effects of constructing and pattern matching and store the required information in data type environment Δ. We support algebraic data types as seen in Haskell, where the type variables of recursive positions are invariant (we support no polymorphic recursion; finger trees [12] for instance cannot be handled), and recursion does not occur in contravariant positions. Unsupported data types should

Figure 5: Inference of a program

$$\frac{\begin{array}{c} \psi, \psi' \text{ fresh} \qquad \hat{\rho} : \hat{P} \text{ is a list of (monomorphic) free region variables of } a_1, \ldots, a_n \\ \Gamma_{\text{fix}} = \Gamma \cup \{x_1 : \mathcal{N}_{\text{fix}}(f_1, \hat{\rho}, \psi.1), \ldots, x_n : \mathcal{N}_{\text{fix}}(f_n, \hat{\rho}, \psi.n)\} \qquad s = (n(f_1, \Psi_\Delta(\tau_1)), \ldots, n(f_n, \Psi_\Delta(\tau_n))) \\ \Delta, \Gamma_{\text{fix}} \vdash f_1 : a_1 \quad \ldots \quad \Delta, \Gamma_{\text{fix}} \vdash f_n : a_n \\ \text{fix escape } \{\hat{P}\} : \left(\begin{array}{l} [\text{arity}(f_1), \Psi_\Delta(\tau_1)] : \lambda[\psi : s, () : ()] \mapsto \lambda[\psi' : (); \hat{\rho} : \hat{P}] \mapsto a_1, \\ \ldots \\ [\text{arity}(f_n), \Psi_\Delta(\tau_n)] : \lambda[\psi : s, () : ()] \mapsto \lambda[\psi' : (); \hat{\rho} : \hat{P}] \mapsto a_n \end{array} \right) \longrightarrow a' \end{array}}{\Delta, \Gamma \vdash \{x_1 : \tau_1 = f_1; \ \ldots\ x_n : \tau_n = f_n\} : \Gamma \cup \{x_1 : \mathcal{N}(f_1, \hat{P}, a'.1), \ldots x_n : \mathcal{N}(f_n, \hat{P}, a'.n)\}}\ [\text{Program}]$$

Figure 6: Inference of pattern matching and let bindings

$$\frac{x : ((a_1, \ldots, a_n), (\rho, (\rho_1^t, \rho_1^v, \hat{\rho}_1, \ldots, \rho_n^t, \rho_n^v, \hat{\rho}_n))) \in \Gamma \qquad \Delta, \Gamma \cup \{x_i : ((\rho_i^t, \rho_i^v, \hat{\rho}_i), a_i) \mid i \in \{1 \ldots n\}\}, \hat{\rho} \vdash e \ : \ a_{\text{effect}}, \ a_{\text{result}}}{\Delta, \Gamma, \hat{\rho} \vdash \texttt{match}\ x\ \{(x_1 : \tau_1, \ldots, x_n : \tau_n) \to e\} \ : \ a_{\text{effect}}, \ a_{\text{result}}} \ [\textsc{Match}]$$

$$\frac{\hat{\rho}_{\text{bound}} : P_\Delta(\tau) \text{ fresh} \qquad \Delta, \Gamma, \hat{\rho}_{\text{bound}} \vdash e_{\text{bound}} \ : \ a_{\text{effect}}^{\text{bound}}, \ a_{\text{result}}^{\text{bound}} \qquad \Delta, \Gamma \cup \{x : (a_{\text{result}}^{\text{bound}}, \hat{\rho}_{\text{bound}})\}, \hat{\rho} \vdash e_{\text{body}} \ : \ a_{\text{effect}}^{\text{body}}, \ a_{\text{result}}^{\text{body}}}{\Delta, \Gamma, \hat{\rho} \vdash \texttt{let!}\ x : \tau = e_{\text{bound}}\ \texttt{in}\ e_{\text{body}} \ : \ C_\Delta(\tau, \hat{\rho}_{\text{bound}}) \sqcup a_{\text{effect}}^{\text{bound}} \sqcup a_{\text{effect}}^{\text{body}}, \ a_{\text{result}}^{\text{body}}} \ [\textsc{Strict Let}]$$

$$\frac{\hat{\rho}_{\text{bound}} : P_\Delta(\tau) \text{ fresh} \qquad \Delta, \Gamma, \hat{\rho}_{\text{bound}} \vdash_{\text{call}} c_{\text{bound}} \ : \ a_{\text{effect}}^{\text{bound}}, \ a_{\text{result}}^{\text{bound}} \qquad \Delta, \Gamma \cup \{x : (a_{\text{result}}^{\text{bound}}, \hat{\rho}_{\text{bound}}), \hat{\rho} \vdash e_{\text{body}} \ : \ a_{\text{effect}}^{\text{body}}, \ a_{\text{result}}^{\text{body}}}{\Delta, \Gamma, \hat{\rho} \vdash \texttt{let}\ x : \tau = c_{\text{bound}}\ \texttt{in}\ e_{\text{body}} \ : \ C_\Delta(\tau, \hat{\rho}_{\text{bound}}) \sqcup a_{\text{effect}}^{\text{bound}} \sqcup a_{\text{effect}}^{\text{body}}, \ a_{\text{result}}^{\text{body}}} \ [\textsc{Lazy Let}]$$

$$\frac{\begin{array}{c}\psi \text{ fresh} \qquad \hat{\rho}_1 : P_\Delta(\tau_1), \ldots, \hat{\rho}_n : P_\Delta(\tau_n) \text{ fresh} \qquad \Gamma_{\text{bound}} = \Gamma \cup \{x_1 : (\psi.2.1, \hat{\rho}_1), \ldots, x_n : (\psi.2.n, \hat{\rho}_n)\} \\ \Delta, \Gamma_{\text{bound}}, \hat{\rho}_1 \vdash_{\text{call}} c_1 \ : \ a_{\text{effect}}^1, \ a_{\text{result}}^1 \quad \ldots \quad \Delta, \Gamma_{\text{bound}}, \hat{\rho}_n \vdash_{\text{call}} c_n \ : \ a_{\text{effect}}^n, \ a_{\text{result}}^n \\ s = (R, (\Psi_\Gamma(\tau_1), \ldots, \Psi_\Gamma(\tau_n))) \qquad a = \text{fix}\ s.\ \lambda[\psi : s, () : ()] \mapsto (a_{\text{effect}}^1 \sqcup \ldots a_{\text{effect}}^n, (a_{\text{result}}^1, \ldots, a_{\text{result}}^n)) \\ \Delta, \Gamma \cup \{x_1 : (a.2.1, \hat{\rho}_1), \ldots, x_n : (a.2.n, \hat{\rho}_n)\}, \hat{\rho} \vdash e_{\text{body}} \ : \ a_{\text{effect}}^{\text{body}}, \ a_{\text{result}}^{\text{body}}\end{array}}{\Delta, \Gamma, \hat{\rho} \vdash \texttt{let rec}\ x_1 : \tau_1 = c_1, \ldots, x_n : \tau_n = c_n\ \texttt{in}\ e_{\text{body}} \ : \ C_\Delta(\tau_1, \hat{\rho}_1) \sqcup \ldots C_\Delta(\tau_n, \hat{\rho}_n) \sqcup a.1 \sqcup a_{\text{effect}}^{\text{body}}, \ a_{\text{result}}^{\text{body}}} \ [\textsc{Recursive Let}]$$

use $\top$ as annotation. We assign regions and annotations per mutual recursive group of data type, where we assign regions and annotations per field, per constructor, per data type. Recursive positions of the data type are given the same regions and annotations. When constructing and pattern matching, we propagate the appropriate regions and annotations. When assigning annotations, we group data types in mutual recursive groups, by looking at all the types it uses. For regions, we may ignore types in function types, as P_Δ° doesn't use them. We refer to section 3.4 of [8] for more details.

7 CONCLUSION

To improve the precision of automatic region based memory management, we presented a higher-ranked region inference algorithm. The analysis gathers outlive-constraints as relations between region variables and binds the regions using the escape check. The annotations of higher order functions are higher order annotation functions: such an annotations is a function that takes the annotation of the argument.

We extended previous work on higher-ranked analyses to region inference and a polymorphic source language. The latter caused problems with fixed-point iteration: it is more difficult to detect a fixed-point, and some polymorphic higher-order functions do not have a fixed-point. We have shown various directions for solutions to this problem.

7.1 Implementation

We have an implementation of the analysis available online[1]. This implementation is based on an experimental LLVM backend [7], so there may be bugs unrelated to region inference. The implementation differs slightly to the formalisation in this paper. We use De Bruijn indices [6] to avoid name collisions. We add containment on the regions for the return value of a function. This conflicts with the lifetime context $\uparrow$, but improves the results of the escape check. The formalisation of $\uparrow$ should be refined to allow this. The handling of let-bindings is different in our implementation, as the analysis is implemented on an SSA-based intermediate representation. Hence we don't create nested fixed-points, but instead create one large fixed-point [8, 19].

[1] https://github.com/Helium4Haskell/helium/tree/regions/src/Helium/CodeGeneration/Iridium/Region

7.2 Future work

To evaluate the full potential of higher-ranked region inference in Haskell, the run-time part needs to be implemented. We have experimented with a higher-ranked analysis to infer the memory size of a region [19], as previously done in a non-higher-ranked analysis [26]. This should be integrated with a run-time system to allocate bounded regions on the stack and unbounded regions on the heap [4], and to handle programs where region inference behaves poorly, a runtime garbage collector could, selectively for some regions, reduce memory usage [11].

The evaluation of fixed-points requires more attention. If the analysis can easier detect that a fixed-point is reached, or does not exist, then the analysis can become faster. Better strategies to handle functions that do not have a fixed-point will improve the precision of the analysis.

REFERENCES

[1] Andrew W Appel. Ssa is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.
[2] Zena M Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of functional programming*, 7(3):265–301, 1997.

[3] Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258(1-2):299–392, 2001.
[4] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM, 1996.
[5] Geoffrey L Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of computer programming*, 7:249–278, 1986.
[6] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.
[7] Ivo Gabe de Wolff. The helium haskell compiler and its new llvm backend. *EuroLLVM 2019, available online https://youtu.be/x6CBks1paF8*, 2019. Accessed: 2019-08-06.
[8] Ivo Gabe de Wolff. Higher ranked region inference for compile-time garbage collection. Master's thesis, 2019.
[9] Martin Elsman. Garbage collection safety for region-based memory management. *ACM SIGPLAN Notices*, 38(3):123–134, 2003.
[10] Jean-Yves Girard. The system f of variable types, fifteen years later. *Theoretical computer science*, 45:159–192, 1986.
[11] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. *ACM SIGPLAN Notices*, 37(5):141–152, 2002.
[12] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of functional programming*, 16(2):197–217, 2006.
[13] Stefan Holdermans and Jurriaan Hage. Making "stricterness" more relevant. *Higher-Order and Symbolic Computation*, 23(3):315–335, 2010.
[14] Stefan Holdermans and Jurriaan Hage. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *ACM Sigplan Notices*, volume 45, pages 63–74. ACM, 2010.
[15] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture*, pages 190–203. Springer, 1985.
[16] Simon Marlow et al. Haskell 2010 language report. *Available online https://www.haskell.org/definition/haskell2010.pdf*, 2010.
[17] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.
[18] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Type and effect systems. In *Principles of Program Analysis*, pages 283–363. Springer, 1999.
[19] HP Ottens. Higher ranked region bound inference for region based memory management in helium. Master's thesis, 2021.
[20] Cristina Ruggieri and Thomas P Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–293, 1988.
[21] Gunther Schmidt. *Relational mathematics*, volume 132. Cambridge University Press, 2011.
[22] Fabian Thorand and Jurriaan Hage. Higher-ranked annotation polymorphic dependency analysis. In *ESOP*, pages 656–683, 2020.
[23] Fabian K Thorand. Higher-ranked polymorphism in dependency analyses. Master's thesis, 2017. *Available online, https://studenttheses.uu.nl/bitstream/handle/20.500.12932/26370/thesis.pdf*.
[24] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):724–767, 1998.
[25] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.
[26] Magnus Vejlstrup. Multiplicity inference. *Master's thesis, Dept. of Computer Science, Univ. of Copenhagen*, 1994.

Closures in a Higher-Order Polymorphic DSL for GPU programming

André Rauber Du Bois, Henrique Gabriel Rodrigues, and Gerson Cavalheiro

Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas - RS - Brazil
{dubois,henrique.grdr,gerson.cavalheiro}@inf.ufpel.edu.br

Abstract. This paper describes the design and implementation of the support for closures, i.e., anonymous functions that can capture non-local variables from surrounding scopes, in PolyHok, a higher-order polymorphic DSL for GPU computing embedded in the Elixir functional language. The main abstraction of PolyHok are *higher-order GPU kernels*, i.e., GPU kernels that can take device functions as arguments before being launched. There are many challenges in supporting closures in such a language. CPU code must be able to reference GPU code, like functions and closures, at host code, so that kernels can be configured at launch time. Closures, in a dynamic language like Elixir, are functions generated at run time, and in the case of PolyHok, they must have their types inferred in order to be JIT compiled into GPU code. Variable capturing is also a challenge as device code is capturing variables that are located in the CPU, and not in the device. The implementation of closures presented in this paper is completely based on Metaprogramming, without modifying Elixir's compiler.

Keywords: closures, DSL, GPU, Metaprogramming

1 The PolyHok DSL

The PolyHok DSL for writing GPU kernels embedded in the Elixir functional language [1]. It is a higher-order, dynamically typed, imperative language supporting loops, CUDA grid and block constants, and in-place update of arrays, as can be seen in the example of Figure 1, where a kernel for a simple parallel map is presented. In the Figure, we see the definition of a `PolyHok` module (line 3). A `PolyHok` module is just like a regular Elixir module, but it also allows the definition of kernels, using the `defk` keyword (line 4), and device functions, using the `defd` keyword (line 13). Kernels are implemented using the `PolyHok` DSL (lines 4 to 12). The `map_ker` kernel showcases the main characteristics of the DSL: it is imperative, providing loops (line 9) and in-place update (line 10), it gives access to CUDA grid and block constants (lines 5-7), it is higher-order, i.e., function `f` is an argument to the kernel (line 4) and is applied to all positions of the array argument `a1` (line 10), and it is dynamically typed, i.e., the programmer does not type variables and types are inferred at runtime.

```
1  require PolyHok
2
3  PolyHok.defmodule PMap do
4    defk map_ker(a1,a2,size,f) do
5      ind = blockIdx.x * blockDim.x
6              + threadIdx.x
7      str = blockDim.x * gridDim.x
8
9      for i in range(ind,size,str) do
10           a2[i] = f(a1[i])
11     end
12   end
13   defd inc(x) do
14     x+1
15   end
16 end
```

A GPU kernel for a `map` skeleton

```
1  def map(input, f) do
2   shape = PolyHok.get_shape(input)
3   type = PolyHok.get_type(input)
4   result_gpu =PolyHok.new_gnx(shape,type)
5   size = Tuple.product(shape)
6   threadsPerBlock = 128;
7   numberOfBlocks = div(size +
8                        threadsPerBlock - 1,
9                        threadsPerBlock)
10
11  PolyHok.spawn(&PMap.map_ker/4,
12            {numberOfBlocks,1,1},
13            {threadsPerBlock,1,1},
14            [input,result_gpu,size, f])
15  result_gpu
16 end
```

Pure Elixir `map` skeleton

Fig. 1. GPU kernel and Elixir implementation for a `map` skeleton

The PolyHok DSL can be seen as a low-level language that programmers can use to construct high-level Elixir abstractions. For example, the pure Elixir function defined in Figure 1, implements a parallel `map` using the `map_ske` kernel. The `map` skeleton takes as an input a GNx array, i.e., an array that is located in the GPU memory (`input`) and a device function (`f`). It first creates a GNx array to hold the result of the map computation, with the same size and type as the input array (line 4). Then, after computing the number of threads and blocks to be used, the `map_ske` kernel is launched with the `spawn` primitive (line 11). PolyHok's `spawn` takes as arguments a kernel to be launched, two tuples configuring the GPU's grid, and a list of arguments to be passed to the kernel. Finally, once the kernel has been executed, the `map` skeleton returns the GNx array containing the `map`'s result (line 15).

The reader should notice that in-place updates are limited to PolyHok kernels, and the purely functional part of Elixir cannot offload a GNx from the GPU memory while it is being modified by kernels, since a call to get_gnx is blocked while the GNx is in use on the GPU. A PolyHok kernel can be seen as an isolated Elixir process that does not communicate. However, a kernel can still be executed by a regular Elixir process that communicates and runs PolyHok kernels as needed.

We can use the `map` skeleton and the `inc` device function from Figure 1, on different array types, as can be seen in the script of Figure 2. First, three host arrays of different types (integer, float, and double) are created using the `Nx` library (lines 3 to 4). In the example, we employ Elixir's composition operator (`|>`), which uses the expression on its left as the first argument of the expression on its right. In the first two compositions (lines 6 to 9 and lines 11 to 14), both `map` and `inc` are applied to arrays of different types (i.e., integer and float). On the third composition (lines 16 to 19), we perform the same computation on an array of doubles, but instead of using the `inc` function, a device anonymous function is used. Device anonymous functions are defined using `PolyHok.hok` primitive that does not allow variable capture. As with device functions, PolyHok anonymous

functions can be written using a subset of Elixir or PolyHok. Also, pure Elixir functions can not be invoked on device anonymous functions, only PolyHok or CUDA functions can be called.

```
n = 10000000
arr1 = Nx.tensor([Enum.to_list(1..n)],type: {:s, 32})
arr2 = Nx.tensor([Enum.to_list(1..n)],type: {:f, 32})
arr3 = Nx.tensor([Enum.to_list(1..n)],type: {:f, 64})

host_res1 = arr1
    |> PolyHok.new_gnx
    |> PMap.map(&PMap.inc/1)
    |> PolyHok.get_gnx

host_res2 = arr2
    |> PolyHok.new_gnx
    |> PMap.map(&PMap.inc/1)
    |> PolyHok.get_gnx

host_res3 = arr3
    |> PolyHok.new_gnx
    |> PMap.map(PolyHok.hok fn x -> x + 1 end)
    |> PolyHok.get_gnx
```

Fig. 2. Using the `map` skeleton with arguments of different types

2 Closures in PolyHok

We extend PolyHok with the `PolyHok.clo` primitive, that allows the definition of an anonymous device function that can capture non-local variables from surrounding scopes at run time, as in the following example:

```
1 dev_arr = PolyHok.new_gnx(Nx.tensor(Enum.to_list(1..1000), type: :s32))
2
3 x = 10
4
5 fun = PolyHok.clo fn y -> x + y end
6
7 host_arr = dev_arr
8  |> Ske.map(fun)
9  |> PolyHok.get_gnx
```

In the example, we define a closure (line 5) that captures the value of variable `x` (defined in line 3). Then, the closure is applied to all elements of a device array

(previously defined in line 1), using a `map` skeleton (line 8). Hence, all elements of the final `host_array` are incremented by 10.

Closures in PolyHok can capture numerical values and also GNx arrays. Numerical values are captured by value, i.e., converted into device values and passed as extra arguments to the device function, and GNx arrays are captured by reference, i.e., the anonymous function receives a pointer to the device array.

Providing closures in a higher-order language for GPU programming allows two main advantages. First, it allows to simplify the interface of skeletons. For example, the function argument of a map, usually relies on other values besides the ones that come from the input array. For example, in the `NBodies` program (see Section 5), that simulates how physical forces influence a dynamic system of particles, for each body in the system we have to check all other bodies to calculate the force impact. Hence, to implement it as a map, the function argument needs not only the current body, but a reference to array of bodies and its size. We could hard code values inside anonymous functions, but then we would need different versions fo the program for different sizes. We could also provide different implementations of a map that receive extra arguments to be passed to the anonymous functions. Closures solve this problem as they can capture the specific values for the computation being performed at run time.

Closures also allow the implementation of other high-level abstractions, as for example, *array comprehensions*, as described in the next Section.

3 Array Comprehensions

List comprehension is another abstraction that can be implemented using higher-order functions. Using *List Comprehensions*, programmers can use notation similar to set theory to describe new lists based on existing ones. Using Elixir's macros, it is possible to implement simple array comprehensions, which are translated to calls to skeletons configured with closures. For example, the following array comprehension:

```
1  host_array =  Nx.tensor(...)
2  x = 10
3  host_resp = PolyHok.gpu_for n <- host_array,  do: x * n
```

which multiplies the elements of an input array by 10, can be translated by the macro on Figure 3, to the following code:

```
1  PolyHok.new_gnx(host_array)
2    |> PMap.map(PolyHok.phok(fn n -> x * n end))
3    |> PolyHok.get_gnx
```

Closures are crucial for the implementation of array comprehensions, since, otherwise, the expressions allowed in the comprehensions would be very limited. Since GNx arrays are captured by reference, they can also be used inside comprehensions as in this implementation of *saxpy*:

```
1  host_resp = PolyHok.gpu_for i <- 0..99999,  do:  2 * a[i] + b[i]
```

```
defmacro gpu_for({:<-, _ ,[var,tensor]},do: b)  do
 quote do:
   PolyHok.new_gnx(unquote(tensor))
   |> PMap.map(PolyHok.phok(fn (unquote(var))->(unquote b) end))
   |> PolyHok.get_gnx
end
```

Fig. 3. Macro for compiling a simple Array Comprehension

4 Implementation

The implementation of closures in PolyHok relies on compilation time support, through macros, and execution time support, for the JIT compilation of closures. These issues will be detailed in the extended version of the paper.

5 Experiments

In the extended version of this paper we plan to include experiments comparing programs using PolyHok with closures with programs written in pure CUDA, in order to understand the impact of using the high-level abstractions provided by PolyHok.

6 Conclusions and Future Work

This paper described the support for closures implemented in the PolyHok DSL, a higher-order polymorphic DSL for writing GPU kernels. In the extended version of this paper we plan to discuss how the PolyHok system was modified to support closures, compare our design with related works, and present experiments comparing PolyHok with pure CUDA, with the object of understanding the overhead imposed by the high-level of abstraction provided by PolyHok.

References

1. Du Bois, A.R., Cavalheiro, G.: Polymorphic Higher-Order GPU Kernels. In: Euro-Par 2025: European Conference on Parallel and Distributed Processing (2025)

Automatic Testing for Finite-State Machines

Marco T. Morazán
Seton Hall University
South Orange, NJ, USA
morazanm@shu.edu

Sophia G. Turano
Seton Hall University
South Orange, NJ, USA
turanoso@shu.edu

Andrés M. Garced
Seton Hall University
South Orange, NJ, USA
maldona2@shu.edu

David Anthony K. Fields
Seton Hall University
South Orange, NJ, USA
fieldsda@shu.edu

Abstract

A modern trend in Formal Languages and Automata Theory education uses a design- and programming-based approach to teach students about finite-state machines and regular languages. Following the steps of a design recipe, students use FSM, a domain-specific language embedded in Racket, to design, implement, validate, and verify deterministic and nondeterministic finite-state machines. As part of the design process, students implement state invariant predicates to validate the role of machine states. Writing thorough unit tests for state invariant predicates, however, is notoriously difficult as test suites grow large very quickly. This article describes a tool to automatically and thoroughly test all such predicates for a given machine. The tool uses the given machine's transition relation to provide good test coverage by finding all paths to intermediate states between the machine's starting state and any of its final states. The result is an elegant way to test state invariants using a single expression. Despite operating in a search space that grows exponentially, empirical measurements suggest that the use of the tool is feasible for typical problems studied in a Formal Languages and Automata Theory course.

CCS Concepts

• **Theory of computation → Regular languages**; • **Applied computing → Education**; • **Software and its engineering**;

Keywords

Finite-state machines, Machine design, Invariants, Testing

ACM Reference Format:
Marco T. Morazán, Sophia G. Turano, Andrés M. Garced, and David Anthony K. Fields. 2025. Automatic Testing for Finite-State Machines. In . ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

A modern approach in Formal Languages and Automata Theory (FLAT) education introduces students to the systematic design of finite-state machines [29]. Unlike most approaches to FLAT education, where students implement machines by pencil and paper [14, 21, 24, 38, 41] or by using a GUI-based system like JFLAP [22, 39, 40] or OpenFLAP [26], this new approach makes students' interest and experience in programming an integral part of the learning process. To this end, students are not expected to learn how to implement finite-state machines solely by looking at examples. Instead, they are presented with a design recipe that provides scaffolding for the development and implementation of machines from a problem statement to a verified solution.

A design recipe is a series of steps, each with a concrete outcome, that defines a systematic process to design and implement programs. It is a teaching methodology first pioneered by Felleisen et al. for a one-semester programming course for beginners [10] and later expanded by Morazán into a 2-semester sequence for beginners [27, 28]. In the FLAT context relevant to this article, design recipes are used to implement deterministic (dfa) and nondeterministic (ndfa) finite-state machines in FSM (Functional State Machines)–a domain-specific language, embedded in Racket [11], for the FLAT classroom [30]. By following the steps of the design recipe, a student reasons about each component needed to build a machine. For instance, students attach meaning to each state before developing a transition relation. In turn, invariant predicates are written to validate the role of states.

When a machine applies a transition rule, the input word may be partitioned: the consumed input (ci) and the unconsumed input. We can, therefore, speak of an invariant property the ci needs to satisfy for the machine to be in a given state. Invariant properties may be tested by implementing a *state invariant predicate*. An invariant predicate is given the ci and determines if a condition holds. This condition defines the role of the state. For example, the role of a state may be that the ci is empty or that the ci has an even number of bs. Invariant predicates, of course, must be thoroughly tested to validate that they work properly. This task sounds deceptively simple, but is (for good reason) difficult to perform. The major stumbling block is that thorough testing does not only depend on the invariant-predicate's implementation. It also depends on the machine's transition relation. If there is at least one sequence of transitions that take the machine from its starting state through a state R to a final state (equivalently, there exists at least one path from the starting state through R to a final state in the

Figure 1 Machine and invariant test failure highlighting.

```
(check-accept? M '() '(a a a) '(a a) '(a a b a a a))
(check-reject? M '(b) '(a b) '(a a a) '(a a b a a))
```

(a) Machine test failure highlighting.

```
(check-inv-holds? A-INV '(a a a) '(a a b a b) '(a b a a))
(check-inv-fails? A-INV '(b b) '(a a b) '(b a))
```

(b) Invariant test failure highlighting.

transition diagram), then the testing suite for R needs to include words obtained by using all transition rules on paths that take the machine from its starting state to R. To do so for all states in a machine is tantamount to counting/finding all paths in a directed graph, a problem that is known to be #P [45]. That is, there is no known polynomial time algorithm to find a thorough testing suite for a machine's state invariant predicates. To intuitively understand this, consider the number of possible paths without repeating a state in a transition diagram that is as a complete graph with n nodes. There are n*(n-1) paths of length 2, n*(n-1)*(n-2) paths of length 3, and so on until n! paths of length n. Thus, the total number of paths is $\Sigma_{i=2}^{n}\Pi_{j=n-i+1}^{n} j$. The complexity is worse for invariant testing, because paths cannot be limited to those without repeated nodes.

This article describes a tool integrated into FSM to automatically test state invariant predicates for dfas and ndfas using a single expression:

```
(sm-test-invs <machine> (list state <inv>)+)
```

Given a machine and a list of pairs containing a state and its invariant predicate, the tool performs a traversal of the transition relation to identify the words that must be used to test each state invariant predicate. If none of state predicate invariants fail, the tool returns an empty list of failures. Otherwise, it returns a list of pairs containing a state and a list of words for which the state's invariant predicate failed. Given that the machine invariant testing is NP, the goal for the tool is not to be efficient for arbitrary machines. Instead, the goal is for the tool to be a useful and practical for typical problems studied in a FLAT classroom. The article is organized as follows. Section 2 briefly describes the FSM interface for dfas and ndfas. Section 3 presents an overview of the design recipe for state machines and an illustrative example. Section 4 describes the developed tool to automatically test state invariant predicates and presents empirical data that suggests classroom use of the tool is feasible for problems typically studied in a FLAT course. Section 5 illustrates how the tool is used for debugging. Section 6 compares and contrasts with related work. Finally, Section 7 presents concluding remarks and directions for future work.

2 Finite-State Machines in FSM

2.1 Constructors and Observers

FSM provides support for both dfas and ndfas. The constructors have the following signatures:

```
make-dfa: K Σ S F δ → dfa    make-ndfa: K Σ S F Δ → ndfa
```

Each takes as input the machine's states (K), input alphabet (Σ), starting state (S∈K), final states (F⊆K), and a transition relation represented as a list of transition rules. For a dfa, the transition relation (δ) must be a function. For an ndfa, the transition relation (Δ) does not have to be a function. Each transition in a relation has the form: (state symbol state). Each state must be a member of K. For a dfa transition rule, the symbol must be in Σ. For an ndfa transition rule, the symbol must be in $\Sigma \cup \{\epsilon\}$, where ϵ denotes that no input is read. A dfa accepts an input word, w, if after reading all of w's elements it ends in a final state. Otherwise, w is rejected. An ndfa accepts w if there is a computation that ends in a final state after reading all of w's elements. If no such computation exists, then the ndfa rejects w.

The primitive sm-showtransitions returns a trace of a single accepting computation. For example, the following is the trace of an ndfa for the language L=a(ab)*aa+ applied to '(a a b a a a):

```
(((a a b a a a) S)
 ((a b a a a)   A)
 ((b a a a)     B)
 ((a a a)       A)
 ((a a)         C)
 ((a)           D)
 (()            D)
 accept)
```

A computation is a list of configurations. Each configuration is a pair containing the unconsumed input and the machine's current state. In any given configuration, the part of the input word that is missing, ci, is the consumed input.

2.2 Unit Testing

FSM provides a DSL to write unit tests, which is tightly-coupled with the design recipe for state machines (discussed in Section 3.1). It allows programmers to write succinct tests for machines and invariant predicates.

Finite-state machines are designed to accept words in a given language and to reject words not in the given language. To test a machine, M, the following syntax is used:

```
(check-accept? M <word>+)    (check-reject? M <word>+)
```

Each given word is tested by applying the machine to it. If all the words are correctly accepted/rejected the tests pass. Consider the following test suite for, M, a machine deciding L=a(ab)*aa+:

Figure 2 The state invariant predicate for A.

```
(define (A-INV ci)
  (and
   (eq? (first ci) 'a)
   (even? (length (rest ci)))
   (andmap (λ (p) (equal? p '(a b)))
           (foldl
            (λ (s acc)
             (if (or (empty? acc)
                     (= (length (first acc)) 2))
                 (cons (list s) acc)
                 (cons (list (first (first acc)) s)
                       (rest acc))))
            '()
            (rest ci)))))
```

```
(check-accept? M '() '(a a a) '(a a) '(a a b a a a))
(check-reject? M '(b) '(a b) '(a a a) '(a a b a a))
```

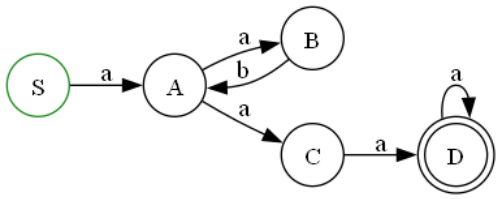
The generated fail test reports are recipe-based errors:

```
Step 6 of the design recipe has not been
successfully completed. The constructed machine,
M, does not accept the following words: () (a a)

Step 6 of the design recipe has not been
successfully completed. The constructed machine,
M, does not reject the following words: (a a a)
(a a b a a)
```

In addition, the program window highlights the words that cause the test to fail. The words that cause `check-reject?` to fail, for example, are displayed in Figure 1a.

As mentioned, students write an invariant predicate for each state. For instance, the transition diagram for M is:



Observe that whenever the machine transitions into A, the consumed input that led the machine to A must be in $a(ab)^*$. A's invariant predicate may be implemented as displayed in Figure 2. To test A-INV, the following test suite may be proposed:

```
(check-inv-holds? A-INV '(a a a) '(a a b a a)
                        '(a b a a))
(check-inv-fails? A-INV '(b b) '(a a b) '(b a))
```

Upon running the tests, the generated fail test reports are:

```
Step 6 of the design recipe has not been
successfully completed. The following test cases
for the invariant, A-INV, fail when they should
hold: (a a a) (a b a a)
Step 6 of the design recipe has not been
successfully completed. The following test case
for the invariant, A-INV, holds when it should
fail: (a a b)
```

Upon clicking on the first error, Figure 1b displays the highlighting of the words that cause `check-inv-holds?` to fail. Clicking on the second error causes the words that cause the second test to fail to be highlighted (in this case, the second word given to `check-inv-fails?`).

Figure 3 The State Machine Design Recipe

(1) Name the machine and specify the input alphabets
(2) Write unit tests
(3) Associate a state with tracked conditions, and identify the start and final states.
(4) Formulate the transition relation
(5) Implement the machine
(6) Run the tests and, if necessary, redesign
(7) Design, implement, and test an invariant predicate for each state
(8) Prove L = L(M)

2.3 Visualizations

Finally, FSM integrates a suite of static and dynamic visualization tools to help students understand FLAT concepts. The static visualization tools include those to generate a transition diagram [36] (e.g., used to generate the graphic for M) and to generate a computation graph given a machine and a word [32, 33]–a computation graph summarizes all the paths in a computation tree to provide visual proof that a word is accepted or rejected. In such static visualizations, every node represents a state and every edge represents a transition rule.

The dynamic visualization tools include machine execution [31] and an array of transformation algorithms. During machine execution visualization, the value of the state invariant predicate may be observed for the state transitioned into (i.e., green when it holds and red when it fails). The transformation algorithm tools include, for example, the `ndfa` to `dfa` transformation [35], the transformations to and from finite-state machines and regular expressions [34], and construction algorithms for closure properties of regular languages [25].

3 Machine Design

At the heart of the finite-state machine design process is the meaning of states. States represent conditions that need to be clearly defined. A state's condition is a property of the consumed input that must hold when the machine transitions into said state in an accepting computation. This section outlines the design recipe for state machines and presents an illustrative example.

3.1 The State Machine Recipe

The 8-Step design recipe for state machines is displayed in Figure 3 [29], where L is the target language the machine is designed to accept. Step 1 requires students to select a descriptive name for

the machine and, for finite-state machines, specify the input alphabet[1]. Step 2 requires students to write a thorough suite of unit tests. For finite-state machines, this means writing tests for words in the machine's language and for words not in the machine language. The `check-accept?` and `check-reject?` syntax described in Section 2.2 is used.

Step 3 asks for informal descriptions of the conditions states represent. Students are taught to start by specifying the conditions the starting state and the final states represent. Following this, students reason about how the machine transitions between states as a word is consumed. Each time a new condition is encountered, a new state is needed. Based on the result of Step 3, Step 4 asks students to formulate the transition relation. Note that "guessing work" is eliminated if Step 3 is successfully completed before formulating an answer for Step 4. This is important, because it allows for students and instructors to discuss finite-state machine design using a common language.

Step 5 requires the collection of answers from Steps 1-4 to implement the machine using the constructors described in Section 2.1. Step 6 has students run the tests. If any errors are thrown or if any tests fail, students are taught to revisit the steps of the design recipe to fix bugs. In addition to the instructors encouraging this practice, the `FSM` error-messaging system [8] and the `FSM` fail test reports encourage this practice by including the step of the design recipe that has been unsuccessfully completed. This is important, because the steps have meaning for students from lectures and the textbook. Therefore, their inclusion encourages them to think in terms of design and recall or look up how the step is satisfied in the solutions of previous problems.

Step 7 asks students to formalize the answer to Step 3. This is the step in which informal descriptions are turned into state invariant predicates. This is a difficult step for two reasons. The first is that informal descriptions may be too informal and, therefore, may require a revision step to clarify the condition a state represents. This challenge is something that advanced Computer Science students usually handle well. The second is that state invariant predicates must be tested and the testing suite must be thorough. The latter part poses a new challenge for students. Consider testing the invariant predicate for a state `N`. To be thorough, words generated from all possible transitions used in computations paths that may lead the machine to accept and that take the machine from its starting state to `N` must be tested. That is, thoroughness is dependent on the machine's transition relation and not just on the implementation of the invariant predicate. In part, this challenge has motivated the work presented in this article.

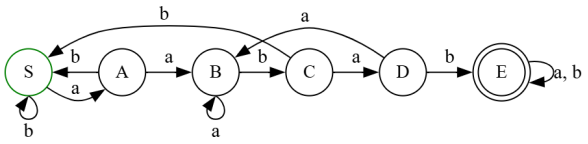
Finally, Step 8 asks for establishing the equivalence between, `L`, the language the machine is designed to decide and, `L(M)`, the language of, `M`, the machine implemented. This is done in two steps. First, students develop a proof by induction on the number of steps taken by an accepting computation to prove that invariant predicates always hold. To aid students in this step the dynamic visualization tool for machine execution accepts invariant predicates as optional arguments to observe if they hold. Invariably, every semester, students realize that their invariant predicates do not always hold despite unit tests passing. That is, they realize that their invariant predicate testing is not thorough enough. Second, based on invariant predicates always holding, they prove the equivalence of the stated languages.

[1]The emphasis on finite-state machines follows from noting that the design recipe is also used to develop pushdown automata, which require a stack alphabet to be defined.

Figure 4 The `dfa` for `L={w|w contains aabab}`.

```
1 ;; State Documentation
2 ;; S: ci suffix != prefix of aabab, aabab∉ci, starting state
3 ;; A: ci ends with a and no other aabab prefix, aabab∉ci
4 ;; B: ci ends with aa and no other aabab prefix, aabab∉ci
5 ;; C: ci ends with aab and no other aabab prefix, aabab∉ci
6 ;; D: ci ends with aaba and no other aabab prefix, aabab∉ci
7 ;; E: aabab∈ci, final state
8 (define CONTAINS-aabab
9   (make-dfa '(S A B C D E)
10             '(a b)
11             'S
12             '(E)
13             '((S a A) (S b S) (A a B) (A b S)
14               (B a B) (B b C) (C a D) (C b S)
15               (D a A) (D b E) (E a E) (E b E))))
16 (check-accept? CONTAINS-aabab
17                '(a a b a b) '(b b a a a b a b b))
18 (check-reject? CONTAINS-aabab
19                '() '(b b a a a b a) '(a a b a a))
```

(a) `FSM` implementation.



(b) Transition diagram.

3.2 Illustrative Example

To illustrate the design process, consider a classical application of finite-state machines that is the foundation of the `KMP` algorithm [19]: detecting a pattern. Our sample problem is to design a `dfa` to decide `L={w|w contains aabab}`. The result of following the first 6 Steps of the design recipe are displayed in Figure 4a and the transition diagram for the implemented machine is displayed in Figure 4b. The backbone of the machine from `S` to `E` detects progressively larger consecutive portions of the pattern. When a letter is read that does not match the next element in the pattern, the machine transitions to the state whose role matches the largest possible prefix of the pattern to the suffix of the consumed input. For example, if the machine is in state `D` then the largest suffix of `ci` that matches the pattern (i.e., `aabab`) is `aaba`. Upon reading an `a` the largest suffix of `ci` that matches the pattern is `aa` and, therefore, the machine transitions to, `B`, the state whose role is satisfied. The role of each state is documented on lines 1–7 indicating the part of the pattern detected.

Figure 5 CONTAINS-aabab's predicates and auxiliary functions.

```
;; word → Boolean
;; Purpose: Determine if word contains aabab
(define (contains-aabab? w)
  (and (>= (length w) 5)
       (or (equal? (take w 5) '(a a b a b))
           (contains-aabab? (rest w)))))
;; word word → Boolean
;; Purpose: Determine if second word ends with first word
(define (end-with? suffix w)
  (and
   (>= (length w) (length suffix))
   (equal? suffix (take-right w (length suffix)))))
;; word → Boolean
;; Purpose: Determine that none of aabab is detected
(define (S-INV ci)
  (and (not (end-with? '(a) ci))
       (not (end-with? '(a a) ci))
       (not (end-with? '(a a b) ci))
       (not (end-with? '(a a b a) ci))
       (not (contains-aabab? ci))))
;; word → Boolean
;; Purpose: Determine that only a is detected
(define (A-INV ci)
  (and (end-with? '(a) ci)
       (not (end-with? '(a a) ci))
       (not (end-with? '(a a b a) ci))
       (not (contains-aabab? ci))))
;; word → Boolean
;; Purpose: Determine that only aa is detected
(define (B-INV ci)
  (and (end-with? '(a a) ci)
       (not (contains-aabab? ci))))
;; word → Boolean
;; Purpose: Determine that only aab is detected
(define (C-INV ci)
  (and (end-with? '(a a b) ci)
       (not (contains-aabab? ci))))
;; word → Boolean
;; Purpose: Determine that only aaba is detected
(define (D-INV ci)
  (and (end-with? '(a a b a) ci)
       (not (contains-aabab? ci))))
;; word → Boolean
;; Purpose: Determine that only aabab is detected
(define E-INV contains-aabab?)
```

The needed state invariant predicates to satisfy Step 7 of the design recipe are displayed in Figure 5[2]. Each predicate tests that `ci` does not contain the pattern and that the largest suffix of `ci` that matches any prefix of the pattern is the prefix represented by the predicate's state. For instance, consider `A-INV`. This predicate tests that `ci` ends with an `a`, that no other prefixes of the pattern that end with a match `ci`'s suffix, and that `ci` does not contain the pattern.

Observe that unit tests for the state predicate invariants have not been included. This done to drive home the point that writing such tests is not straightforward. How do you determine a thorough set of tests for each invariant predicate? For instance, consider writing the following tests for `A-INV`:

```
(check-inv-holds? A-INV '(a) '(a a b b a))
(check-inv-fails? A-INV '(b) '(a b b) '())
```

Is this a thorough enough suite of tests? Even a cursory look at the transition diagram in Figure 4b reveals that the testing is lacking. At the very least, the set of words in the testing suite ought to traverse every transition that may be used to reach `A` in an accepting computation. This follows from observing that it does not matter how `A` is reached in an accepting computation. The invariant must hold, because once the machine is in `A`, the unconsumed input may transition the machine to `E` where the word is (eventually) accepted. Under this light, the test suite for `check-inv-holds?` ought to include, for example, `(b a)` and `(a a b a a b b a)`. In fact, it ought to cover all transitions used in accepting computations from `S` to `A`. To guarantee such testing coverage, all paths from `S` to `A` need to be computed. We note that for the machine in Figure 4, the number of words in a thorough test suite for all invariant predicates is 50^3. Clearly, it is not reasonable nor practical for any programmer to write that many unit tests.

We also note that the test suite for `check-inv-fails?` is also lacking. It ought to include words that end with an `a` that should not take the machine to `A` such as `(a a b a)` (which ought to take the machine to `D`) and `(a a)` (which ought to take the machine to `B`).

4 Testing-Word Generation

Consider testing `B-INV`, the invariant predicate for a state `B`. Determining a thorough test suite requires traversing all paths to `B`, including transitioning through any loops. This is hard, because a `dfa` or `ndfa` may be arbitrarily complex and contain arbitrarily nested transition loops. Even for a simple machine, like `CONTAINS-aabab` displayed in Figure 4b, it is not immediately obvious to an `FSM` programmer how to determine the set of words for each state that provides the necessary coverage of transitions.

Despite machine and invariant validation making machine verification easier, there are two problems with forcing `FSM` programmers to determine on their own a thorough test suite. The first is that it is a time-consuming and error-prone process. Untangling nested loops, for example, is always a challenge. The second is that it distracts attention away from machine verification. In fact, determining a thorough testing suite can be harder than verifying a machine.

4.1 Implementation

Our solution, therefore, is two-fold. We encourage the writing of unit tests given that it is important for students to reason about their design. We also adopt the solution put forth by QuickCheck

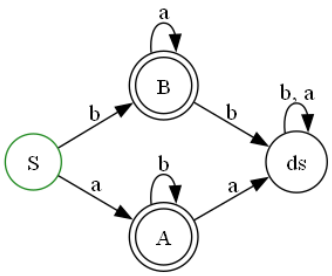
[2]The predicates are not optimized to eliminate code duplication. Such an optimization may be pursued after completing a machine's design

[3]This number is obtained using the word generation software described in the next section.

[2, 15]: generate tests [17]. It is now possible to automatically test a machine's state invariant predicates in `FSM`. A test suite of words for each state, `C`, is generated that provides the necessary (minimum) coverage to include every transition that any accepting computation may use to reach `C`. The implemented solution operates in 3 steps:

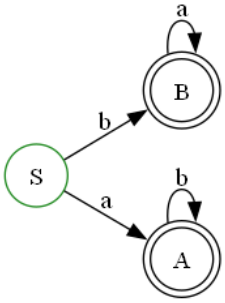
(1) Machine refactoring
(2) Word generation
(3) Invariant testing

4.1.1 Machine Refactoring. The goal is to obtain a test suite for each state that covers all possible transitions on accepting computations. It is possible, of course, for a machine to contain transition rules that never lead to accept. For instance, consider the following `dfa`:



This machine has, `ds`, a dead/trap state. Any computation that reaches this state never leads to accept. Therefore, there is no need to generate a test suite for `ds`. That is, a testing suite for the invariant predicate for `ds` does not need to be generated nor does the testing suite for any state invariant predicate need to provide coverage for the transitions in and out of `ds`. We note that an arbitrary finite-state machine may have an arbitrary number of dead/trap states.

The first step, therefore, is to refactor the given machine by removing all dead/trap states and the transitions into and out of them, which does not change the language of the machine. This is accomplished by determining the states reachable from the starting state and then filtering out those states from which a final state is not reachable. Both of these subtasks are performed using a breadth-first traversal of the transition relation (or, equivalently, the graph representing the machine's transition diagram). The result of this process for the machine above yields:



Test suites are only generated for `S`, `A`, and `B`.

4.1.2 Word Generation. To generate a test suite for each state in a refactored machine, a breadth-first traversal of its transition relation is performed. The traversal accumulates two values: a queue of computations to explore (each computation represented as the list of transition rules used) and a list of computations found. The queue is used to search for longer computations by only applying transition rules that have not been used in a given computation. In this manner, the needed coverage of transitions is obtained. A consequence of this search policy is that a loop is traversed at most once, thus, reducing the amount of work done when, for example, a loop contains a nested inner loop. The list of computations found is used to generate test suites for each state invariant predicate. For each state, `K`, in an accepting computation, the (sub)word that leads to `K` is made part of `K`'s testing suite.

Initially, the list of found computations is empty and the queue contains a number of computations equal to the number of transitions out of the given machine's starting state. Therefore, initially, each computation contains a single rule. At each step, the first computation in the queue is removed and extended by applying all applicable rules not used in the computation to its current state. The new computations are added to the queue and the computation removed from the queue is added to the list of found computations. When the queue becomes empty the accumulated list of found computations is returned.

To illustrate the process, consider the search for computation using the refactored machine above. The search starts with the following values:

```
        Queue: (((S b B))
                ((S a A)))
Computations: ()
```

There are two computations to explore: one for each transition rule out of `S`. The first step removes the first computation from the queue, adds a new computation to the queue, and adds the first computation to the list of found computations. The resulting state of the accumulators is:

```
        Queue: (((S a A))
                ((S b B) (B a B)))
Computations: (((S b B)))
```

Note that the computation `((S b B))` will be used to generate `(b)` as part of the test suite for `B`'s invariant. The process is repeated to yield the following state for the accumulators:

```
        Queue: (((S b B) (B a B))
                ((S a A) (A b A)))
Computations: (((S a A))
                ((S b B)))
```

Note that the computation `((S a A))` will be used to generate `(a)` as part of the test suite for `A`'s invariant. The process is repeated and there are no unused transitions that apply to `B`. Therefore, the first computation is removed from the queue and and added to the list of found computations. No new computations are added to the queue:

```
        Queue: (((S a A) (A b A)))
Computations: (((S b B) (B a B))
                ((S a A))
                ((S b B)))
```

Observe that `(b a)` shall be part of the testing suite for `B`'s invariant. A similar step is taken again to yield:

```
        Queue: ()
 Computations: (((S a A) (A b A))
                ((S b B) (B a B))
                ((S a A))
                ((S b B)))
```

Observe that `(a b)` shall be part of the testing suite for `A`'s invariant. Given that the queue is empty, the process stops and the list containing the 4 found computations is returned.

The returned list of computations is traversed to generate the needed test suite for each state. For this example, the test suite for each state is:

```
((S (()))
 (A ((a) (a b)))
 (B ((b) (b a))))
```

The empty word is always added to the test suite for the machine's starting state.

4.1.3 Invariant Testing. The function that tests state invariant predicates takes as input a machine and a list of pairs. Each pair contains a state and the state's invariant predicate. The function returns a list of pairs. Each pair contains a state and a word for which the state's invariant predicate does not hold. If the returned list is empty, then the invariant predicates hold for every word in the test suite.

Testing is performed by traversing the returned list of test suites. For each state, the words in the test suite for which the state's invariant predicate does not hold are accumulated and used to construct the list of pairs returned by the testing function.

4.2 Empirical Measurements

Given that thorough invariant testing is in NP, the presented empirical study does not measure performance on arbitrary finite-state machines to determine if the algorithm is practical in general–we know from the start that it is not and that arbitrarily large machines will not be successfully processed. Instead, empirical measurements are taken using machines that are typically studied in a FLAT course to determine if the developed algorithm makes the tool's use in a FLAT classroom practical/feasible.

Empirical data is presented for 17 finite-state machines used in an introduction to FLAT course at Seton Hall University in the Spring 2025 semester. Table 1 outlines the characteristics of the 17 machines: number of states, number of transition rules, alphabet size, number of final states, machine type, number of dead states, and number of dead transitions (i.e., transitions in or out of a dead state). The majority of the machines have 7 or more states. A majority of the machines also have over 10 edges. Thus, the majority of the machines are nontrivial, when compared to typical examples found in FLAT textbooks. We also note that machines M1, M14, M15, and M17, each with over 30 transitions rules each, are more complex that the average machine studied in a FLAT textbook. These machines are the result of large course projects at our institution (e.g., searching for a given long pattern in a DNA strand). Nonetheless, the data collected is presented in the spirit of thoroughness. It is, indeed, good to know, as the data below suggests, that some machines with large number of edges can be automatically tested in a reasonable amount of time.

For each machine, the study includes four experiments:

Machine	|K|	|R|	|Σ|	|F|	Type	Dead K	Dead R
M1	10	40	4	5	dfa	1	16
M2	7	14	2	6	dfa	1	3
M3	4	9	3	3	ndfa	0	0
M4	8	17	4	4	ndfa	0	0
M5	7	18	3	3	ndfa	0	0
M6	6	8	2	4	ndfa	0	0
M7	7	10	3	1	ndfa	0	0
M8	7	21	3	1	dfa	1	11
M9	6	10	3	1	ndfa	0	0
M10	4	8	2	3	dfa	1	3
M11	2	4	2	1	dfa	0	0
M12	3	4	2	2	ndfa	0	0
M13	4	6	3	1	ndfa	0	0
M14	9	35	5	3	ndfa	0	0
M15	10	40	4	5	dfa	0	0
M16	6	12	2	1	dfa	0	0
M17	7	35	5	1	ndfa	0	0

Table 1: Machine characteristics.

(1) Refactored using words generated by using any transition at most once in a computation
(2) Unrefactored using words generated by using any transition at most once in a computation
(3) Refactored using words generated by using any transition at most once twice
(4) Unrefactored using words generated by using any transition at most twice

Refactored and unrefactored machines are used to measure the benefit/penalty of refactoring. The goal of permitting words generated by using any transition at most once twice is to test words that traverse nested loops in the transition relation. For each experiment, automatic testing execution time (in ms) is collected 50 times for a total of 3400 experiments. Data is collected using the laptop issued to students at our institution (processor/OS).

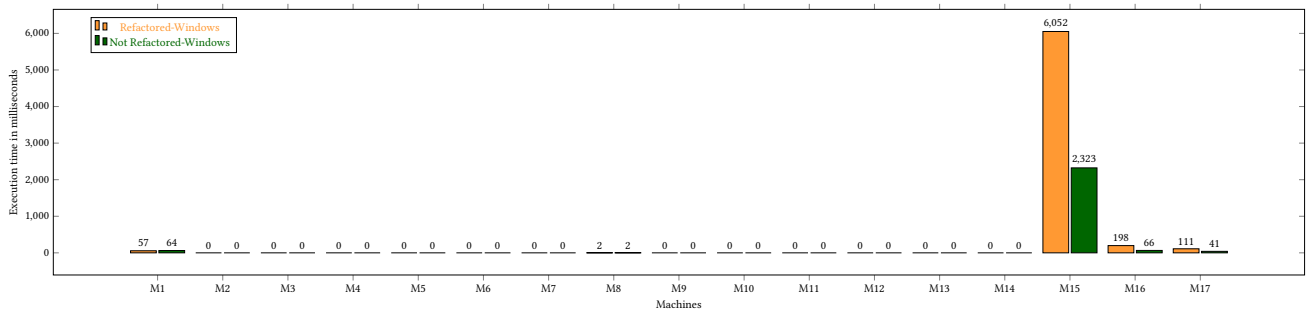
For each machine, the number of test words generated allowing the use of a rule at most once is displayed in Table 2. The median value for the refactored machines is 45 and the median value for the machines that are not refactored is 50, which clearly suggest that it is not reasonable to expect anyone to write a test suite that provides good coverage of the transitions used to reach each state.

The average runtime to test each machine using words generated by using any transition at most once in a computation is displayed in Figure 6. The most salient feature, regardless of whether machines are refactored or not, is that for most machines the running time is under a fifth of a second (i.e., under 200ms), with most machines having an execution time under a millisecond. The sole exception is M15 for which we observe an average running time of about 6 seconds when the machine is refactored and about 2 seconds when the machine is not refactored. This difference in performance is explained by M15's characteristics displayed in Table 1. This machine has 40 edges and no dead states. Therefore, the time penalty observed is attributed to an unsuccessful attempt to remove dead states in a machine with a large transition relation that has no

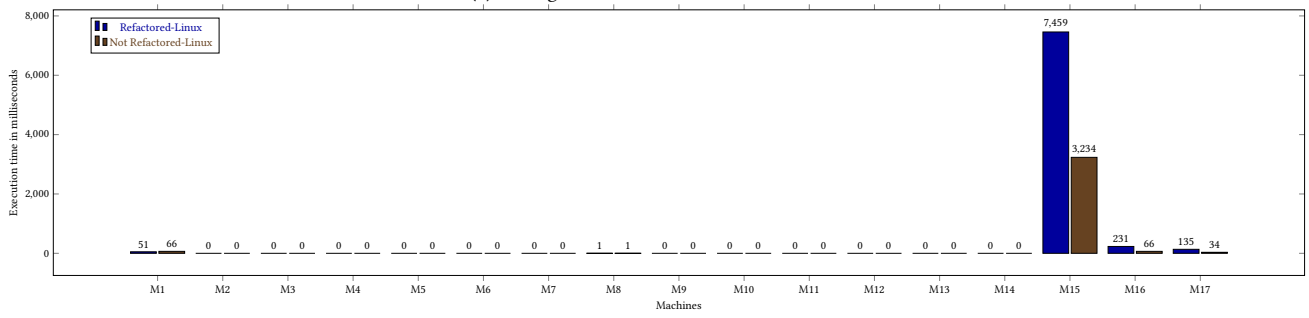
Machine	Refactored	Not refactored	Refactored	Not refactored
Edge use	once	once	twice	twice
M1	325	51024	36834	unfeasible
M2	45	65	1607	3830
M3	15	15	57	57
M4	26	26	66	66
M5	66	66	351	351
M6	10	10	18	18
M7	17	17	43	43
M8	62	1598	363	165131
M9	62	62	363	363
M10	12	37	47	408
M11	11	11	125	125
M12	4	4	6	6
M13	50	50	3159	3159
M14	8	8	12	12
M15	1284774	1284774	unfeasible	unfeasible
M16	51024	520124	unfeasible	unfeasible
M17	27060	27060	unfeasible	unfeasible

Table 2: Number of test words generated for each machine using rules at most once and twice using `Windows`.

Figure 6 Average testing times using words generated from computations that use a transition at most once.



(a) Average execution time under Windows.

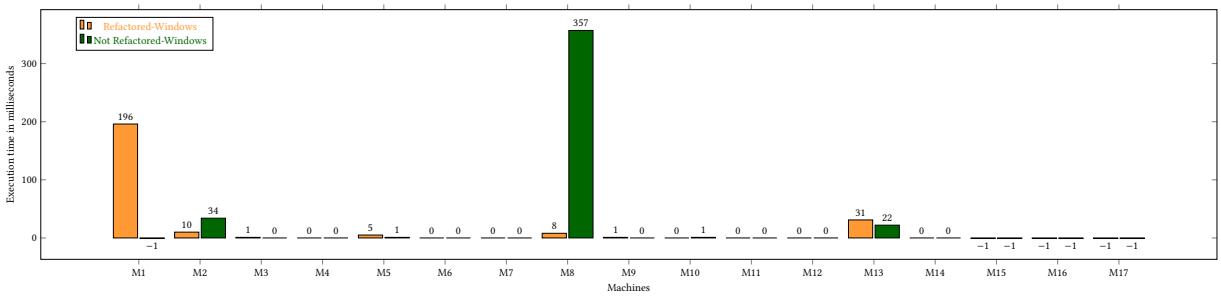


(b) Average execution time under Linux.

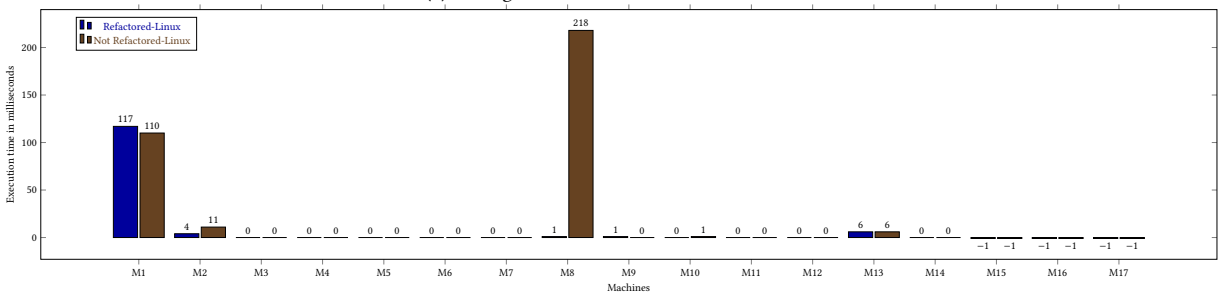
dead states. Despite the performance gap, refactoring the machine still yields an execution time that does not disrupt a smooth user experience. We conclude, therefore, that classroom usage of the tool is feasible as part of a typical undergraduate FLAT course. In addition, we note that the data also suggests that it is feasible for students to use the tool to debug their designs for typical assignments found in FLAT textbooks.

The average runtime to test each machine using words generated by using any transition at most twice in a computation is displayed in Figure 7. This data paints a much more nuanced picture. First, we observe that testing some machine becomes unfeasible

Figure 7 Average testing times using words generated from computations that use a transition at most twice.



(a) Average execution time under Windows.



(b) Average execution time under Linux.

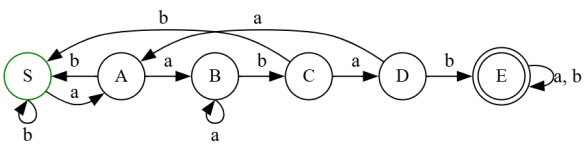
(i.e., `M15`–`M17`), which is indicated by a height of -1 in Figure 7. We note that testing `M1`, without refactoring, becomes unfeasible using `Windows`. Although we cannot be absolutely certain, it is reasonable to hypothesize that the Linux Out-Of-Memory (`OOM`) Killer kernel function [20] prevents the `DrRacket IDE` from crashing by terminating non-essential processes to make enough memory available for testing to terminate normally. Second, we observe that significant gains in performance are achieved for some machines (e.g., `M8`, and `M1` under `Windows`) and that the penalty for refactoring when no gains are observed is extremely small. Thus, we conclude that the machine refactoring step ought to remain an integral part of the testing algorithm.

5 Machine Debugging

Two examples are presented to illustrate how the software to automatically test state invariant predicates is used to debug machines.

5.1 Debugging `CONTAINS-aabab`

The first is a student-designed machine that served as the inspiration for the machine displayed in Figure 4. The student has implemented the following buggy `dfa` in `FSM` to decide if a given word contains `aabab`:



She has also implemented the state predicate invariants displayed in Figure 5 along with a testing suite for them. Her machine and invariant tests all pass. A perceptive instructor is likely to immediately notice that `'(a a b a a b a b)` is rejected. Clearly, that is a bug given that the word ends with `aabab` and ought to be accepted. Naturally, the student is upset because all her tests pass. After some discussion, it becomes clear that simply trying to analyze the transition diagram is leading the student nowhere.

At this point, the student is encouraged to use `FSM`'s tool to automatically test state invariant predicates as follows:

```
(sm-test-invs CONTAINS-aabab
              (list 'S S-INV)
              (list 'A A-INV)
              (list 'B B-INV)
              (list 'C C-INV)
              (list 'D D-INV)
              (list 'E E-INV))
```

The tool yields the following results:

```
'(((a a b a a)       A)
  ((a a a b a a)     A)
  ((b a a b a a)     A)
  ((b a a a b a a)   A)
  ((a a b a a b)     S)
  ((a a a b a a b)   S)
  ((b a a b a a b)   S)
  ((b a a a b a a b) S))
```

The invariant predicates for `S` and `A` do not always hold. The list of failing words provides the student with a new starting point for debugging. She can observe how the machine processes the problematic words by launching the dynamic visualization tool. Figure 8 displays the final state of the dynamic visualization tool when `CONTAINS-aabab` is applied to `'(a a b a a)`. The student

Figure 8 Final dynamic visualization tool state.

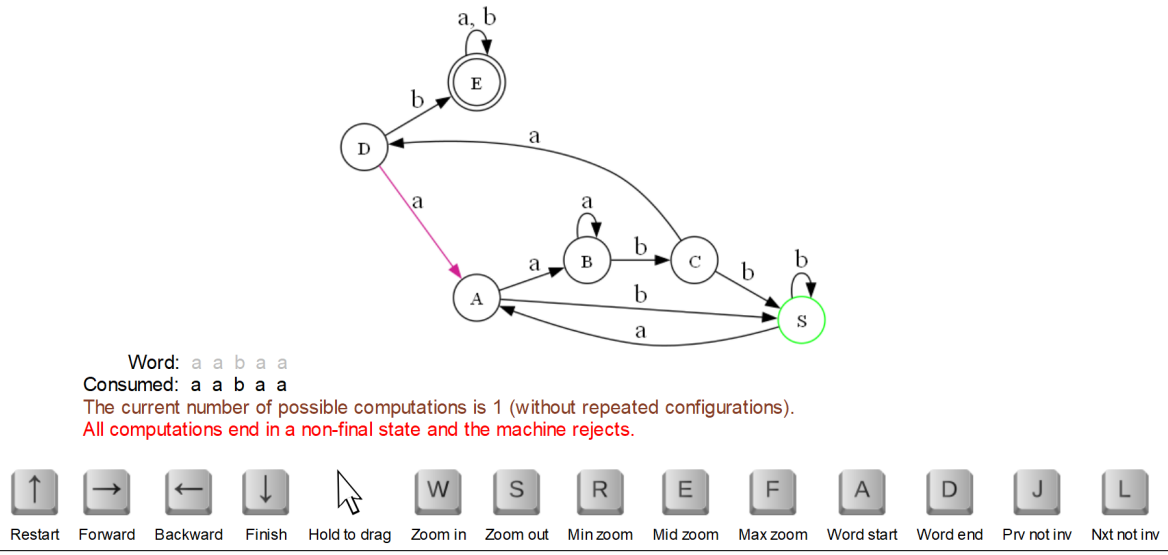
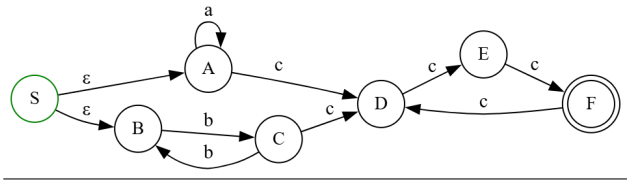


Figure 9 The ndfa for L=(b(bb)* ∪ a*)(ccc)+.



can see why A-INV fails. The machines ends in A (indicated by the violet edge), but should not given that the given word ends with '(a a). This suggests to the student that there is a bug in the transition function and they quickly realize that from D on an a the machine ought to move to B, not A. Upon updating (D a A) to (D a B) all tests pass and sm-test-invs returns the empty list. The student also adds the following test:

```
(check-equal?
  (sm-test-invs CONTAINS-aabab
                (list 'S S-INV)
                (list 'A A-INV)
                (list 'B B-INV)
                (list 'C C-INV)
                (list 'D D-INV)
                (list 'E E-INV))
  '())
```

Observe that this is a succinct and elegant way of testing state invariant predicates.

5.2 Debugging an ndfa

Consider implementing an ndfa to decide:

L = (b(bb)* ∪ a*)(ccc)+

Following the steps of the design recipe, a student has implemented in FSM the ndfa displayed in Figure 9. A cursory look at the transition diagram suggests that the machine is correct. There are two paths from S to D: one consumes a* and the other consumes b(bb)*. This correctly implements the union of these two languages. Both of these paths end by consuming a c. Thus, when the machine transitions into D, ci∈(b(bb)* ∪ a*)c. Observe that the number of cs is multiple of 3 minus 2. When the machine transitions into E and F, respectively, the number of cs is multiple of 3 minus 1 and a multiple of 3.

Based on this the student argues that the machine is correct, but tests for the predicate invariants for D, E, and F are failing. The tests are not shown in the interest of brevity. Given that the transition relation appears correct, the predicate implementation needs to be examined:

```
(define (D-INV ci)
  (let* [(bcs (takef ci (λ (s) (not (eq? s 'c)))))
         (cs (drop ci (length bcs)))]
    (and (or (B-INV bcs) (A-INV bcs))
         (andmap (λ (s) (eq? s 'c)) cs)
         (= (remainder (length cs) 3) 1))))
(define (E-INV ci)
  (let* [(bcs (takef ci (λ (s) (not (eq? s 'c)))))
         (cs (drop ci (length bcs)))]
    (and (or (B-INV bcs) (A-INV bcs))
         (andmap (λ (s) (eq? s 'c)) cs)
         (= (remainder (length cs) 3) 2))))
(define (F-INV ci)
  (let* [(bcs (takef ci (λ (s) (not (eq? s 'c)))))
         (cs (drop ci (length bcs)))]
    (and (or (B-INV bcs) (A-INV bcs))
         (andmap (λ (s) (eq? s 'c)) cs)
         (= (remainder (length cs) 3) 0))))
```

Upon using the described automatic invariant testing tool the following result is obtained:

```
'(((b c) D)
  ((b c c) E)
  ((b c c c) F))
```

The words that are causing the invariant predicates to fail all have the correct properties. That is, an odd number of bs followed by a number of cs modulo 3 corresponding to 1, 2, and 0, respectively, for D, E, and F. Further observe, that all the test words that fail are for computations traversing D. This suggests that there is a bug in at least D-INV. Upon inspecting its implementation, the student realizes that they have used B-INV, instead of C-INV, in the implementation of D-INV. They further acknowledge that the same bug is present in E-INV and F-INV, because they copied, pasted, and edited the second argument to remainder to implement E-INV and F-INV.

6 Related Work

6.1 Validation in FLAT Education

Most FLAT textbooks do not address validation of dfas and ndfas (e.g., [14, 21, 24, 38, 41]). This is expected for two reasons. First, students are not expected to take their ideas beyond the pencil and paper design and, therefore, there is no possibility of running tests. Second, FLAT textbooks rarely, if ever, address the correctness of state machines and focus much more on the correctness of transformation constructors (e.g., the transformation of an ndfa to a dfa). A notable exception is Morazán's textbook which is tightly-coupled with FSM [29]. The work presented in this article builds on his work by automating the testing of state invariant predicates. The result is the use of a single function to validate all state invariant predicates for a machine. Instead of writing individual tests for each predicate, a single expression is used and when invariant predicates fail, the failed words in the test suite are returned to the programmer to aid in debugging.

GUI-based state machine simulators offer machine testing facilities. JFLAP [39, 40] and Automaton Simulator [7], for example, allow users to build their own machines by manually drawing a transition diagram (i.e., placing nodes and edges). Like FSM, such machines may be tested by providing a list of words to determine which are accepted and which are rejected. Most simulators, however, only allow for a single word to be tested at a time (e.g., [1, 3, 12, 13, 18, 43]). Automata Tutor [4, 9] is a popular tool to provide students machine building exercises. Unlike FSM, users only receive feedback based on the exercise they are trying to solve. That is, it is not a platform for building arbitrary machines.

Before FSM, the only software system that addressed the need to validate state invariants is ProofChecker [42]. This system is restricted to dfas, but is notable given that it allows students to specify state invariants and then performs automatic validation of invariants. Unlike FSM, invariant validation is not linked to generating a test suite based on using the given machine's transition relation. Instead, all words of a predetermined length are generated and used to test invariants. As a consequence, thorough test coverage is not guaranteed.

6.2 Test Generation

Arguably, the best-known test generation software is QuickCheck [2, 16]. A programmer defines a predicate for a property that specifies how a piece of code ought to work. QuickCheck generates random type-safe inputs and tests if the predicate holds. If the predicate fails, a shrinking step is performed to reduce the input to a minimal counterexample. To allow programmers to perform better targeted testing, QuickCheck allows programmers to write their own test generators. Like QuickCheck, the work presented in this article automatically generates tests for the programmer. In contrast, the work presented does not rely on random generation of tests. Instead, finite-state machines provide a domain that allows the generation of tests by traversing a transition relation. QuickCheck's shrinking step, to some degree, is built-in to the traversal of the transition relation by not allowing the use of a transition relation within a computation more than once.

Westphal and Voigtländer describe a specification and testing domain-specific language to teach students how to write interactive Haskell programs [44]. This framework provides support for expressing read/write specifications, capturing the trace of an executed program, writing a function to determine if a trace is valid given the specification, and a testing procedure to check programs against the specification. Testing is performed by repeatedly taking a random sample of a generalized trace for a given specification, extracting the inputs from said trace, and determining if the trace of a program being tested when given said inputs is covered by the generalized trace. In the work presented in this article, the notion of a generalized trace is tantamount to a refactored machine after removing states that do not lead to a final state. The extraction of inputs is performed by the traversal of the transition relation. In contrast, the work presented does not specify the notion of a trace. Instead, validation is performed by testing properties specified by the programmer as state invariant predicates.

There is a growing pool of software tools to perform automatic testing. Hypothesis is a property-based testing library for Python that is inspired by QuickCheck, which provides mechanisms for targeted testing [23]. Proptest is a property testing framework for Rust that is inspired by Hypothesis and QuickCheck [6]. Unlike QuickCheck that only allows one generator and shrinker per type, Proptest allows for multiple of these for a type using strategies–each defines how to generate and shrink values. ScalaCheck [37] and RapidCheck [5] are libraries for property-based testing used, respectively, in Scala and C++ that are also inspired by QuickCheck. Like cited work, these libraries substitute test writing with property writing and randomly generate input values. In contrast to all of the cited work, the work presented moves away from random testing given that thorough testing may be achieved by traversing a machine's transition relation. In addition, FSM programmers do not need to write properties–which simplifies testing for FLAT students.

7 Concluding Remarks

This article presents a novel tool to test state invariant properties for deterministic and nondeterministic finite-state machines. The tool is integrated into the FSM and generates a thorough testing suite that covers all possible transitions on accepting computations. The testing suite is generated by traversing the given machine's

transition relation (or, equivalently, the machine's transition diagram) and accumulating, for each state A, all words that transition the machine into A. The process is guaranteed to terminate given that any loop in the transition relation is traversed only once for any computation. However, given that finding all paths in graph is a #P problem, the tool is only proven useful for the typical FLAT classroom. Nonetheless, the result is a testing function that allows all state invariants to be elegantly tested using a single expression. When invariants fail, the testing function returns the states and the words for which invariants failed. This information is used by the programmer to debug their design before attempting to verify their machine.

Future work is three-fold. First, the described software will be extended to serve as the basis for generating test cases for `dfas` and `ndfas` using `QuickCheck` [2, 15]. This will require the generation of regular expressions, not words, for all paths to a state in any computation that may lead to accept. In essence, a regular language will be defined for each state in a given machine. Second, the methodology will be extended to pushdown automata. Third, human-factors studies will be conducted to ascertain student perceptions about the usefulness of the tool to debug machines and to assist in developing correctness arguments.

References

[1] Carl Burch. 2008. Automaton Simulator. https://cburch.com/proj/autosim/download.html. Last Accessed: June 2025.

[2] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. doi:10.1145/351240.351266

[3] Joshua J. Cogliati, Frances W. Goosey, Michael T. Grinder, Bradley A. Pascoe, Rockford J. ROSS, and Cheston J. Williams. 2005. Realizing the Promise of Visualization in the Theory of Computing. *J. Educ. Resour. Comput.* 5, 2 (June 2005), 5–es. doi:10.1145/1141904.1141909

[4] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How Can Automatic Feedback Help Students Construct Automata? *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 9 (mar 2015), 24 pages. doi:10.1145/2723163

[5] RapidCheck Developer. 2021. RapidCheck User Guide. https://github.com/emil-e/rapidcheck/blob/master/doc/user_guide.md. Last Accessed: 08/2025.

[6] Proptest Developers. 2025. Proptest. https://proptest-rs.github.io/proptest/intro.html. Last Accessed: 08.2025.

[7] Kyle Dickerson. 2022. Automaton Simulator. https://automatonsimulator.com/. Last Accessed: May 2025.

[8] Shamil Dzhatdoyev, Josephine A. Des Rosiers, and Marco T. Morazán. 2024. Error Messages for Students Taught Using a Systematic Program Design Curriculum. In *Trends in Functional Programming - 25th International Symposium, TFP 2024, South Orange, NJ, USA, January 10-12, 2024, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 14843)*, Jason Hemann and Stephen Chang (Eds.). Springer, 195–219. doi:10.1007/978-3-031-74558-4_9

[9] Loris D'Antoni, Martin Helfrich, Jan Kretinsky, Emanuel Ramneantu, and Maximilian Weininger. 2020. Automata Tutor v3. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 3–14. doi:10.1007/978-3-030-53291-8_1

[10] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (Second ed.). MIT Press, Cambridge, MA, USA.

[11] Matthew Flatt, Robert Bruce Findler, and PLT. 2024. *The Racket Reference.* PLT. https://docs.racket-lang.org/reference/index.html Last accessed: June 2024.

[12] Michael T. Grinder. 2002. Animating Automata: A Cross-Platform Program for Teaching Finite Automata. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 63–67. doi:10.1145/563340.563364

[13] Michael T. Grinder. 2003. A Preliminary Empirical Evaluation of the Effectiveness of a Finite State Automaton Animator. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Navada, USA) *(SIGCSE '03)*. Association for Computing Machinery, New York, NY, USA, 157–161. doi:10.1145/611892.611958

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[15] John Hughes. 2010. *Software Testing with QuickCheck*. Springer Berlin Heidelberg, Berlin, Heidelberg, 183–223. doi:10.1007/978-3-642-17685-2_6

[16] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *Proceedings of the Conference on Dependable Software Systems and Experience-Based Models*. 141–152. doi:10.1007/978-3-319-30936-1_9

[17] John Hughes. 2017. Don't Write Tests! https://www.youtube.com/watch?v=hXnS_Xjwk2Y. Last Accessed: May 2025.

[18] Romauld Jagielski. 1988. Visual Simulation of Finite State Machines. *SIGCSE Bull.* 20, 4 (Dec. 1988), 38–40. doi:10.1145/54138.54145

[19] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. doi:10.1137/0206024

[20] Joongjin Kook, Sukil Hong, Wooseung Lee, Eunkyeung Jae, and JungYeop Kim. 2011. Optimization of out of Memory Killer for Embedded Linux Environments. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (TaiChung, Taiwan) *(SAC '11)*. Association for Computing Machinery, New York, NY, USA, 633–634. doi:10.1145/1982185.1982324

[21] Harry R. Lewis and Christos H. Papadimitriou. 1997. *Elements of the Theory of Computation* (2nd ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA. doi:10.1145/300307.1040360

[22] Peter Linz and Susan H. Rodger. 2023. *An Introduction to Formal Languages and Automata* (seventh ed.). Jones & Bartlett Learning, Burlington, MA, USA.

[23] David R. MacIver. 2019. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open Source Software* 4, 43 (2019), 1891. doi:10.21105/joss.01891

[24] John C. Martin. 2003. *Introduction to Languages and the Theory of Computation* (3 ed.). McGraw-Hill, Inc., New York, NY, USA.

[25] Tijana Minić, Marco T. Morazán, Andrés M. Garced, and David Anthony K. Fields. 2025. Visualizing Construction Algorithms for Closure Properties of Regular Languages. In *Proceedings of the 2025 on Innovation and Technology in Computer Science Education V. 1* (Nijmegen, The Netherlands) *(ITiCSE 2025)*. Association for Computing Machinery, New York, NY, USA.

[26] Mostafa Kamel Osman Mohammed. 2020. Teaching Formal Languages through Visualizations, Simulators, Auto-graded Exercises, and Programmed Instruction. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE 2020, Portland, OR, USA, March 11-14, 2020*, Jian Zhang, Mark Sherriff, Sarah Heckman, Pamela A. Cutter, and Alvaro E. Monge (Eds.). ACM, 1429. doi:10.1145/3328778.3372711

[27] Marco T. Morazán. 2022. *Animated Problem Solving - An Introduction to Program Design Using Video Game Development.* Springer, Cham, Switzerland. doi:10.1007/978-3-030-85091-3

[28] Marco T. Morazán. 2022. *Animated Program Design - Intermediate Program Design Using Video Game Development.* Springer, Cham, Switzerland. doi:10.1007/978-3-031-04317-8

[29] Marco T. Morazán. 2024. *Programming-Based Formal Languages and Automata Theory - Design, Implement, Validate, and Prove.* Springer, Cham, Switzerland. doi:10.1007/978-3-031-43973-5

[30] Marco T. Morazán and Rosario Antunez. 2014. Functional Automata - Formal Languages for Computer Science Students. In *Proceedings 3rd International Workshop on Trends in Functional Programming in Education, TFPIE 2014, Soesterberg, The Netherlands, 25th May 2014 (EPTCS, Vol. 170)*, James L. Caldwell, Philip K. F. Hölzenspies, and Peter Achten (Eds.). Open Publishing Association, Australia, 19–32. doi:10.4204/EPTCS.170.2

[31] Marco T. Morazán, David Anthony K. Fields, Andrés M. Garced, and Tijana Minić. 2025. Visual Execution and Validation of Finite-State Machines and Pushdown Automata. In *Proceedings Fourteenth International Workshop on Trends in Functional Programming in Education, TFPIE@TFP 2025, University of Oxford, Oxford, United Kingdom, 13th January 2025 (EPTCS)*, Rose Bohrer (Ed.). Open Publishing Association, Australia.

[32] Marco T. Morazán and Oliwia Kempinski. 2024. Using Computation Graphs to Explain Nondeterminism to Students. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E* (Pasadena, CA, USA) *(SPLASH-E '24)*. Association for Computing Machinery, New York, NY, USA, 23–33. doi:10.1145/3689493.3689978

[33] Marco T. Morazán, Oliwia Kempinski, and Andrés M. Garced. 2025. The Implementation of FSM Computation Graphs. In *Proceedings of the 36th Symposium on Implementation and Application of Functional Languages* (Nijmegen, The Netherlands) *(IFL '24)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/3723325.3723337

[34] Marco T. Morazán and Tijana Minić. 2024. Finite-State Automaton To/From Regular Expression Visualization. *Electronic Proceedings in Theoretical Computer Science* 405 (July 2024), 36–55. doi:10.4204/eptcs.405.3

[35] Marco T. Morazán and Tijana Minić. 2024. Nondeterministic to Deterministic Finite-State Machine Visualization: Implementation and Evaluation. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York,

NY, USA, 262–268. doi:10.1145/3649217.3653641

[36] Marco T. Morazán, Joshua M. Schappel, and Sachin Mahashabde. 2020. Visual Designing and Debugging of Deterministic Finite-State Machines in FSM. *Electronic Proceedings in Theoretical Computer Science* 321 (August 2020), 55–77. doi:10.4204/eptcs.321.4

[37] Rickard Nilsson. 2021. ScalaCheck: Property-Based Testing for Scala. https://scalacheck.org/. Last Accessed: 08/2025.

[38] Elaine Rich. 2019. *Automata, Computability and Complexity: Theory and Applications.* Pearson Prentice Hall.

[39] Susan H. Rodger. 2006. *JFLAP: An Interactive Formal Languages and Automata Package.* Jones and Bartlett Publishers, Inc., USA.

[40] Susan H. Rodger, Bart Bressler, Thomas Finley, and Stephen Reading. 2006. Turning automata theory into a hands-on course. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2006, Houston, Texas, USA, March 3-5, 2006*, Doug Baldwin, Paul T. Tymann, Susan M. Haller, and Ingrid Russell (Eds.). ACM, 379–383. doi:10.1145/1121341.1121459

[41] Michael Sipser. 2013. *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.

[42] Matthias F. Stallmann, Suzanne P. Balik, Robert D. Rodman, Sina Bahram, Michael C. Grace, and Susan D. High. 2007. ProofChecker: An Accessible Environment for Automata Theory Correctness Proofs. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Dundee, Scotland) *(ITiCSE '07)*. Association for Computing Machinery, New York, NY, USA, 48–52. doi:10.1145/1268784.1268801

[43] Luiz Filipe M. Vieira, Marcos Augusto M. Vieira, and Newton J. Vieira. 2004. Language Emulator, a Helpful Toolkit in the Learning Process of Computer Theory. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA) *(SIGCSE '04)*. Association for Computing Machinery, New York, NY, USA, 135–139. doi:10.1145/971300.971348

[44] Oliver Westphal and Janis Voigtländer. 2020. Implementing, and Keeping in Check, a DSL Used in E-Learning. In *Functional and Logic Programming: 15th International Symposium, FLOPS 2020, Akita, Japan, September 14–16, 2020, Proceedings* (Akita, Japan). Springer-Verlag, Berlin, Heidelberg, 179–197. doi:10.1007/978-3-030-59025-3_11

[45] Masaki Yamamoto. 2017. Approximately Counting Paths and Cycles in a Graph. *Discrete Applied Mathematics* 217 (2017), 381–387. doi:10.1016/j.dam.2016.09.002

banner above paper title

Energy-aware Data-Parallel Functional Array Processing for Heterogeneous Platforms

Clemens Grelck
Friedrich Schiller University Jena
Jena, Germany
Clemens.Grelck@uni-jena.de

Abstract

Functional array processing is a variant of functional programming where not lists and trees are in the focus of attention but densely stored, immutable arrays. This focus fosters data-parallel operations on functional arrays that by construction are free of side-effects. On larger arrays this offers a wealth of concurrency that can be exploited by compiler and runtime system for parallel execution in a completely transparent way, i.e. without any traces of parallelism and without any directions how to precisely exploit concurrency in the code. Single Assignment C (SAC) is one example of a functional array language that supports fully implicit parallelization for a variety of architectures, almost like a compiler optimization.

The ever changing landscape of hardware platforms in the multicore domain gives rise to new issues that we aim to address in this work. One such issue pertains to the ever growing number of cores that creates an ever growing design space regarding the effective number of cores to use for some data-parallel operation within an application. Here, we extend our previous work on feedback-directed runtime adaptation with a stronger focus on energy savings.

The other issue we address is the impact of heterogeneous platforms on code generation. Functional array processing demands a fully automatic solution to the scheduling and mapping of data-parallel operations. We leverage our existing feedback-directed compilation architecture for this purpose.

Categories and Subject Descriptors Software and its engineering [*Software notations and tools*]: Dynamic compilers

Keywords Functional array programming, Single Assignment C, heterogeneous multicore, loop scheduling, data parallelism

1. Introduction

Single Assignment C (SAC) is a purely functional, data-parallel array language [Grelck and Scholz(2006b), Grelck(2012)] with a C-like syntax (hence the name). SAC features homogeneous, multi-dimensional, immutable arrays and supports both shape- and rank-polymorphic programming: SAC functions may not only abstract from the concrete shapes of argument and result arrays, but even from their ranks (i.e. the number of dimensions). A key feature of SAC is fully compiler-directed parallelisation for a variety of architectures. From the very same source code the SAC compiler generate code for general-purpose multi-processor and multi-core systems [Grelck(2005)], CUDA-enabled GPGPUs [Guo et al.(2011)], heterogeneous combinations thereof [Diogo and Grelck(2013)] as well as clusters of workstations [Macht and Grelck(2019)].

One of the advantages of a fully compiler-directed approach to parallel execution is that compiler and runtime system are technically free to choose any number of cores for execution, and by design the choice cannot interfere with the program logic. This property is usually referred to as *malleability*. Malleability raises one of the central questions of this paper: what would be the best number of threads to choose for the execution of a data-parallel operation? This choice depends on a number of factors, including but not limited to

- the number of array elements to compute,
- the computational complexity per array element and
- the compute platform.

For a large array of computationally challenging values making use of all available cores is a rather trivial choice on almost any machine. However, smaller arrays, less computational complexity or both inevitably lead to the observation illustrated in Fig. 1. While for a small number of threads/cores we often achieve almost linear speedup, the additional benefit of using more cores increasingly diminishes until some (near-)plateau is reached. Beyond this plateau using even more cores often shows a detrimental effect on performance. This is an effect of diminishing performance returns per core in the presence of super-linear cost for synchronisation and the organisation of parallel execution in general [Nieplocha et al.(2007), Saini et al.(2006), Suleman et al.(2008), Pusukuri et al.(2011)].

From a pure performance perspective we would aim at the number of threads that yield the lowest execution time, i.e. the highest speedup in relation to sequential execution. In the example of Fig. 1 that would be 16 threads. However, we typically observe a performance plateau around that optimal number. In the given example we can observe that from 12 to 20 threads the performance obtained only marginally changes.

Now, the landscape of computing has profoundly changed over the last decade, and investing a disproportional amount of resources, and hence energy, for a marginal gain in performance is a bad choice in most, if not almost any, application domain and scenario. The 8 additional cores in the example of Fig. 1 could more productively be used for other tasks or powered down to save energy.

Hence, we are looking at the gradient of the speedup curve. If the additional performance benefit of using one more core drops

IFL 2025 October 1–3, 2025, Montevideo, Uruguay

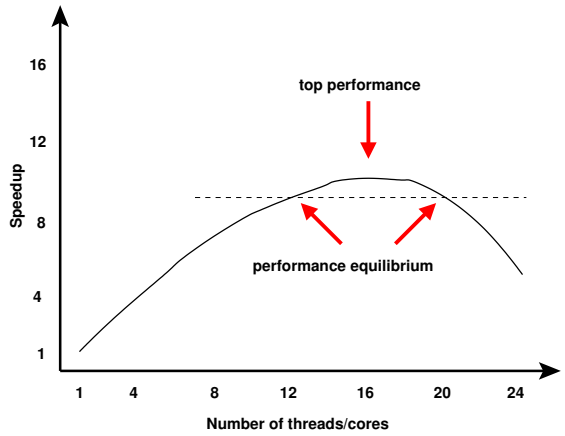


Figure 1. Typical speedup graph observed for multi-core execution, reproduced from [Grelck and Blom(2020)]

below a certain threshold, we constitute that we have reached the optimal number of threads. Where exactly this threshold lies, is highly application-, situation- and platform-dependent.

In classical, high performance oriented parallel computing our issues have hardly been addressed because in this area users have typically strived for solving the largest possible problem size that still fits the constraints of the computing system used. In today's ubiquitous parallel computing world [Catanzaro et al.(2010)], however, the situation has completely changed, and problem sizes are much more often determined by problem characteristics than machine constraints. However, even in high performance computing domain some problem classes inevitably run into the described issues: multi-scale methods. Here, the same function(s) is/are applied to arrays of systematically varied shape and size [Grelck(2002)].

We illustrate multi-scale methods in Fig. 2 based on the example of the NAS benchmark MG (multigrid) [Bailey et al.(1991), Grelck(2002)]. In this so-called *v-cycle* algorithm (A glimpse at Fig. 2 suffices to understand the motivation of the name.) we start the computational process with a 3-dimensional array of large size and then systematically reduce the size by half in each dimension. This process continues until some predefined minimum size is reached, and then the process is sort of inverted and array sizes now double in each dimension until the original size is reached again. The whole process is repeated many times until some form of convergence is reached.

With a given data-parallel operation and for some platform of choice it is the size of the involved arrays that determines the number of cores that can efficiently be used. Hence, the best number of cores to use is different on the various levels of the v-cycle. Regardless of the initial problem size, we always reach problem sizes where we better reduce the number of cores before we tend towards purely sequential execution for the smalles problem sizes.

All the above examples and discussions lead to one insight: in most non-trivial applications we cannot expect to find the one number of cores that is best across all data-parallel operations within an application. But even for a single operation on a given platform the best number of cores to use appears impossible to determine via some form of static analysis.

Therefore, we proposed a (mostly) transparent feedback-directed compilation architecture [Grelck and Blom(2020)]. We follow a two-phase approach that distinguishes between offline training runs and online production runs of an application. In training mode compilation we instrument the generated code to produce an individual performance profile for each data-parallel operation. In production mode compilation we associate each data-parallel operation with an oracle that based on the performance profiles gathered offline chooses the number of threads based on the array sizes encountered at production runtime.

Even for a given platform, the overall design space in terms of array operations, array sizes and potential core counts is huge and, typically, too large to exhaustively explore via profiling. In our previous work [Grelck and Blom(2020)] we used fourth-order polynomial interpolation of the measurement results. This has proven suboptimal, in particular on cc-NUMA architectures, where the performance profile often does not expose the text book characteristics of Fig. 1. In this work we revise the choice for more reliable results.

Orthogonal to rising numbers of cores on a chip, we can observe a trend towards heterogeneous compute platforms. Heterogeneity can have many facets, such as CPU/GPU, tensor cores, crypto cores, etc, but in our current work we focus on binary compatible cores that expose different time/energy characteristics when executing the very same code. We address (variations of) ARM's BIG.little architecture with some number of energy-hungry performance cores and a number of energy-saving less-performant cores. Our general assumption is a small number of classes of cores with a certain number of identical cores per class.

Such architectures offer an even greater design space for scheduling and mapping. In any data-parallel operation, not specific to functional array programming, we typically oberved a degree of concurrency that excedes the number of cores available by various to many orders of magnitude. Therefore, a scheduler is required that decides which core computes which elements of a data-parallel operation. This is generally referred to as *loop scheduling*. While functional array programming in SAC is characterised by multi-dimensional array comprehension instead of nested loops typical for imperative solutions, the resulting scheduling problems are quite similar.

Like for instance OpenMP in the imperative parallel programming world, SAC features both static and dynamic scheduling techniques. With static scheduling the mapping between elements of the multidimensional index space of an array comprehension is fixed a-priori. This choice has the advantage of efficiency as neither synchronisation nor communication between cores is needed during the execution of a data-parallel operation. However, static scheduling silently assumes that the time to compute one element of the index space is about the same as for any other element.

This property indeed holds for many typical data-parallel operations, but not for all. If the time to compute one element of the index space significantly varies from element to element, dynamic scheduling techniques are recommended. Dynamic schedulers incrementally assign work to cores, and whenever a core has completed its assignment, it receives more work until completion of the operation. Starting out from the basic concept of *self-scheduling*, a plethora of dynamic scheduling techniques have been proposed. They guarantees decent load balancing at the expense of higher synchronisation overhead, in particular as core counts are on the rise.

Even more silently static schedulers assume homogeneity of compute units, silently because homogeneity used to be a given until very recently. With multiple classes of cores the underlying assumptions of static schedulers are no longer met. Now, we could switch to dynamic schedulers to solve our problem, but we do not favour the overhead introduced by dynamic scheduling. Moreover, we can observe that the heterogeneity of platforms (as described above) is much more regular than the potential performance heterogeneity of irregular array operations, where computational demands could be randomly distributed over the index space of an array comprehension.

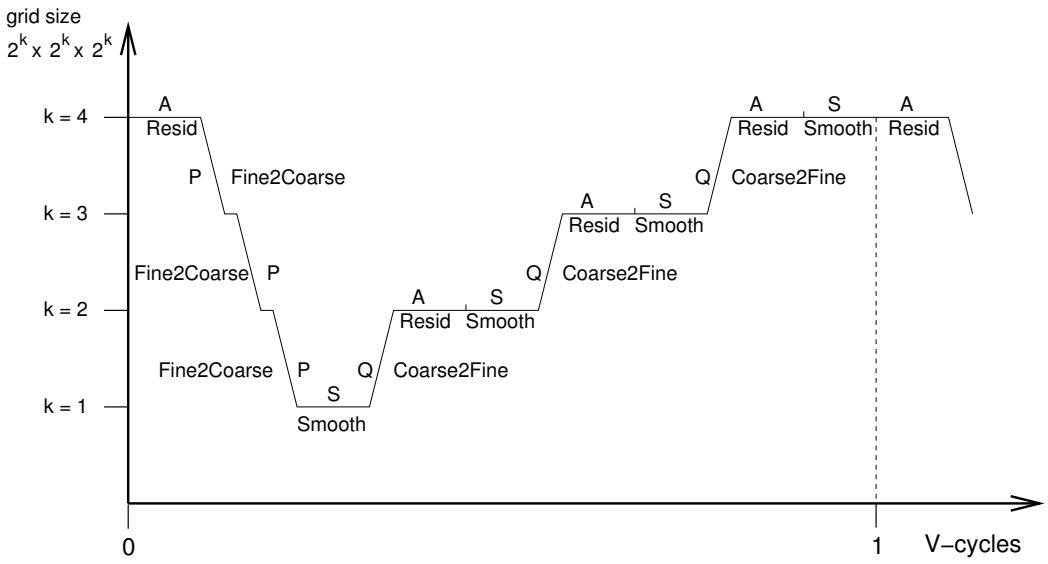


Figure 2. Algorithmic v-cycle structure of NAS benchmark MG as a representative of multi-scale methods, reproduced from [Grelck(2002)]

Therefore, we propose to leverage our feedback-directed compilation archtecture instead and determine the relative performance difference between the different kinds of cores available on the platform. This can be done in a very fine-grained manner, not only specific to a certain application, but effectively specific to an individual array comprehension kernel. In production mode we can, therefore, apply efficient static scheduling and obtain excellent load balancing capabilities on heterogeneous platforms at the same time.

The remainder of the paper is organised as follows. Section 2 provides some background information on SAC, and in Section 3 we sketch out its compilation for (homogeneous) multi-core architectures. In Section 4 we describe our feedback-directed compilation approach. In Section 5 we revise some aspects of feedback-directed compilation based on experience gained. We describe the modifications for heterogeneous system architectures in Section 6. Finally, we draw conclusions in Section 7.

2. Single Assignment C (SAC)

As the name "Single Assignment C" suggests, SAC combines a purely functional semantics based on the context-free substitution of expressions with a C-like syntax and overall look-and-feel. This design is meant to facilitate adoption in compute-intensive application domains, where imperative concepts prevail. We interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail recursion; details can be found in [Grelck and Scholz(2006b)]. Despite the fundamentally different semantics, all syntactic constructs adopted from C show precisely the same operational behaviour as in C proper.

SAC provides genuine support for truly multidimensional and truly stateless/functional arrays using a shape-generic style of programming: any SAC expression evaluates to an array, and arrays are passed to and from functions call-by-value. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]`, and arrays of any rank, e.g. `int[*]`. The latter include scalars, which we consider rank-0 arrays with an empty shape vector.

SAC only features a very small set of built-in array operations, among others to query for rank and shape or to select array elements. Aggregate array operations are specified in SAC itself using WITH-loop array comprehensions:

```
with {
  ( lower_bound <= idxvec < upper_bound) : expr;
    ...
  ( lower_bound <= idxvec < upper_bound) : expr;
}: genarray( shape, default)
```

Here, the keyword `genarray` characterises the WITH-loop as an array comprehension that defines an array of shape *shape*. The default element value is *default*, but we may deviate from this default by defining one or more index partitions between the keywords `with` and `genarray`.

Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to induction variables in FOR-loops. In contrast to FOR-loops, we deliberately do not define any order on these index sets. We call the specification of such an index set a *generator* and associate it with some potentially complex SAC expression that is in the scope of *idxvec* and thus may access the current index location. As an example, consider the WITH-loop

```
A = with {
      ([1,1]<=iv<[4,5]): 10*iv[0]+iv[1];
      ([4,0]<=iv<[5,5]): 42;
    }: genarray( [5,5], 10);
```

that defines the 5×5 matrix

$$\begin{pmatrix} 10 & 10 & 10 & 10 & 10 \\ 10 & 11 & 12 & 13 & 14 \\ 10 & 21 & 22 & 23 & 24 \\ 10 & 31 & 32 & 33 & 34 \\ 42 & 42 & 42 & 42 & 42 \end{pmatrix}$$

WITH-loops are extremely versatile. In addition to the dense rectangular index partitions, as shown above, SAC supports also strided generators. In addition to the `genarray`-variant, SAC features further variants, among others for reduction operations. Furthermore, a single WITH-loop may define multiple arrays

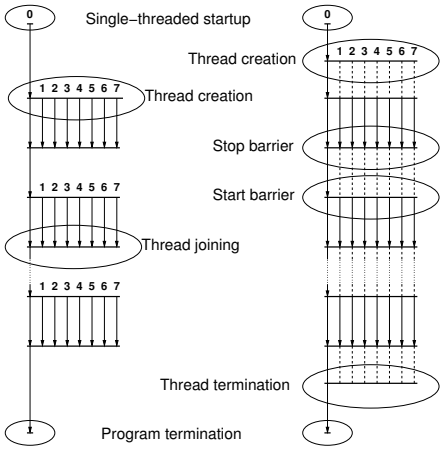


Figure 3. Multithreaded execution models: conceptual fork-join model (left) and start/stop barrier implementation (right)

or combine multiple array comprehensions with further reduction operations, etc. For a complete, tutorial-style introduction to SAC as a programming language we refer the interested reader to [Grelck(2012)].

3. Compiling SAC for multithreaded execution

Compiling SAC programs into efficiently executable code for a variety of parallel architectures is a challenge, where WITH-loops play a vital role. Many of our optimisations are geared towards the composition of multiple WITH-loops into one [Grelck and Scholz(2006a)]. These compiler transformations systematically improve the ratio between productive computing and organisational overhead. Consequently, when it comes to generating multithreaded code for parallel execution on multi-core systems, we focus on individual WITH-loops. WITH-loops are data-parallel by design: any WITH-loop can be executed in parallel. The subject of our current work is: should it?

Originally, the SAC compiler has generated two alternative codes for each WITH-loop: a sequential and a multithreaded implementation. The choice which route to take is made at runtime based on two criteria:

- If the size of an index set is below a configurable threshold, we evaluate the WITH-loop sequentially.
- If program execution is already in parallel mode, we evaluate nested WITH-loops sequentially.

Multithreaded program execution follows an offload (or fork/join) model, as illustrated in Fig. 3. Program execution always starts in single-threaded mode. Only when execution reaches a WITH-loop for which both above criteria for parallel execution are met, worker threads are created. These worker threads join the master thread in the data-parallel execution of the WITH-loop. A WITH-loop-scheduler assigns index space indices to worker threads according to one of several scheduling policies. At last, the master thread resumes single-threaded execution following a barrier synchronisation. We refer the interested reader to [Grelck(2005)] for all details.

The total number of threads, eight in the illustration of Fig. 3, is once determined at program startup and remains the same throughout program execution. This number is typically motivated by the hardware resources of the deployment system. Due to the malleability property of the data-parallel applications concerned, application characteristics are mostly irrelevant. While it would be technically simple to determine the number of available cores at application start, SAC for the time being expects this number to be provided by the user, either through a command line parameter or through an environment variable.

As illustrated on the right hand side of Fig. 3 SAC does not literally implement the fork/join-model. We rather spawn all worker threads right at program start time and, thus, before the first WITH-loop is encountered during program execution. Furthermore, all worker threads are preserved until program termination. The conceptual fork/join model is implemented through two dedicated barriers: the *start barrier* and the *stop barrier*. At the start barrier worker threads wait for activation by the master thread. At the stop barrier the master thread waits for all worker threads to complete the parallel section while the worker threads immediately pass on to the following start barrier.We use highly efficient tailor-made implementations that exploit properties of the cache coherence protocol, but are essentially based on spinning. All details about our barrier implementations in particular and SAC's multicore implementation in general can be found in [Grelck(2003), Grelck(2005)].

4. Feedback-directed compilation

As usual, feedback-directed compilation in SAC consists of two modes: the *training mode* and the *production mode*. When compiling for training mode, the SAC compiler instruments the generated multithreaded code in such a way that

- for each WITH-loop and each problem size found in the code we systematically explore the entire design space regarding the number of threads;
- we repeat each experiment sufficiently many times to ensure a meaningful timing granularity while avoiding excessive training times;
- profiling data is stored in a custom binary database.

Fig. 4 shows pseudo code that illustrates the structure of the generated code. To make the pseudo code as concrete as possible, we pick up the example WITH-loop introduced in Section 2.

The core addition to our standard code generation scheme is a `do-while`-loop plus a timing facility wrapped around the original code generated from our WITH-loop. Let us briefly explain the latter first. The pseudo function `StartThreads` is meant to lift the start barrier for `num_threads-1` worker threads. They subsequently execute the generated function `spmd_fun` that contains most of the code generated from the WITH-loop, among others the resulting nesting of C `for`-loops, the WITH-loop-scheduler and the stop barrier. The record `spmd_frame` serves as a parameter passing mechanism for `spmd_fun`. In our concrete example, it merely contains the memory address of the result array, but in general all values referred to in the body of the WITH-loop are made available to all worker threads via `spmd_frame`. After lifting the start barrier, the master thread temporarily turns itself into a worker thread by calling `spmd_fun` directly via a conventional function call. Note that the first argument given to `spmd_fun` denotes the thread ID. All worker threads require the number of active threads (`num_threads`) as input for the WITH-loop-scheduler.

Coming back to the specific code for training mode, we immediately identify the timing facility, which profiles program execution, but why do we wrap the whole code within another loop? Firstly, the functional semantics of SAC and, thus, the guaranteed absence of side-effects allow us to actually execute the compiled code multiple times without affecting semantics. In a non-functional context this would immediately raise a plethora of concerns whether running some piece of code repeatedly may have an impact on application logic.

```
1    size = 5 * 5;
2
3    A = allocate_memory( size * sizeof(int));
4
5    spmd_frame.A = A;
6    num_threads = 1;
7    repetitions = 1;
8
9    do {
10     start = get_real_time();
11
12     for (int i=0; i<repetitions; i++) {
13       StartThreads( num_threads,
14                     spmd_fun,
15                     spmd_frame);
16       spmd_fun( 0, num_threads, spmd_frame);
17     }
18
19     stop = get_real_time();
20
21     repetitions, num_threads
22       = TrainingOracle (unique_id,
23                         size,
24                         num_threads,
25                         max_threads,
26                         repetitions,
27                         start,
28                         stop);
29   }
30   while (repetitions > 0);
```

Figure 4. Compiled pseudo code of the example WITH-loop from Section 2 in smart decision training mode. The variable `max_threads` denotes a user- or system-controlled upper limit for the number of threads used (reproduced from [Grelck and Blom(2020)]).

However, the reason for actually running a single WITH-loop multiple times is to obtain more reliable timing data. A-priori we have no insight into how long the with-loop is going to run. Shorter runtimes often result in greater relative variety of measurements. To counter such effects, we first run the WITH-loop once to obtain an estimate of its execution time. Following this initial execution a *training oracle* decides about the number of repetitions to follow in order to obtain meaningful timings while keeping overall execution time at acceptable levels.

In addition to controlling the number of repetitions our training oracle systematically varies the effective number of threads employed. More precisely, the training oracle implements a three step process:

Step 1: Dynamically adjust the time spent on a single measurement iteration to match a certain pre-configured time range. During this step the WITH-loop is executed once by a single thread, and the execution time is measured. Based on this time the training oracle determines how often the WITH-loop could be executed without exceeding a configurable time limit, by default 500ms.

Step 2: Measure the execution time of the WITH-loop while systematically varying the number of threads used. This step consists of many cycles, each running the WITH-loop as many times as determined in step 1. After each cycle the execution time of the previous cycle is stored, and the number of threads used during the next cycle is increased by one.

Step 3: Collect measurement data to create a performance profile that is stored on disk. During this step all time measurements collected in step 2 are packaged together with three characteristic numbers of the profile: a unique identifier of the WITH-loop, the size of the index set (problem size) and the number of repetitions in step 1. The packaged data is stored in the application-specific binary smart decision database file on disk.

Let us have a closer look into the third step. In training mode the SAC compiler determines the unique identifier of each WITH-loop by simply counting all WITH-loops in a SAC module. The resulting identifier is compiled into the generated code as one argument of the training oracle. Here, it is important to understand that we do not count the WITH-loops in the original source code written by the user, but those in the intermediate representation after substantial program transformations by the compiler [Grelck and Scholz(2006a)].

The index set size may be known at compile time, as in our simple example, or may only be computed at runtime. In case of a `genarray` or `modarray` WITH-loop the size of the index set coincides with that of the array defined, and is already required for the purpose of memory allocation independent of our current work. However, in the case of a `fold`-WITH-loop we need to generate an expression that symbolically describes the index set size based on the generators' lower and upper bound specifications (and possibly their strides).

Possibly in contrast to readers' expectations we do not systematically vary the problem size, although quite obviously the problem size has a major impact on execution time as well as on the optimal number of threads to be used. Our rationale is twofold: firstly, it is quite possible (and hard to rule out for a compiler) that the problem size does affect the program logic (not so in our simplistic running example, of course). For example, the NAS benchmark MG, that we referred to in Section 1, assumes 3-dimensional argument arrays whose extents along each of the three axes are powers of two. Silently running the code for problem sizes other than the ones prescribed by the application, may lead to unexpected and undesired behaviour, including runtime errors. Secondly, only the user application knows the relevant problem sizes. Unlike the number of threads, whose alternative choices are reasonably restricted by the hardware under test, the number of potential problem sizes is theoretically unbounded and practically too large to systematically explore.

We store all preprocessed and aggregated profiling data obtained in training mode in a custom binary data base file for subsequent use in feedback-directed production mode. All details can be found in [Grelck and Blom(2020)].

When compiling production mode binaries in feedback-directed compilation mode the comoiler locates and reads the corresponding database files created by training mode binaries. This way we can move almost all overhead to production mode compile time while keeping the actual production runtime overhead minimal. In production mode the SAC compiler

1. reads the relevant database files;
2. merges information from several database files;
3. creates a recommendation table for each WITH-loop.

These recommendation tables are compiled into the SAC code. They are used by the compiled code at runtime to determine the number of threads to execute each individual array comprehension kernel.

The combination of *`name`* and *`architecture`* must match with at least one database file, but it is well possible that a specific combination matches with several files, for example if the training is first done with a maximum of two threads and later repeated with a maximum of four threads. In such cases we read all matching

database files for any maximum number of threads and merge them. The merge process is executed for each WITH-loop individually.

As in training mode we identify each WITH-loop by a unique identifier. Since training and production mode compilation does not lead to different intermediate code representations otherwise, we are guaranteed to obtain the same unique identifier for each WITH-loop in either mode. These unique identifiers are matched with the identifiers in the database files to create subselections of database rows.

Average execution times are turned into a performance graph by taking the inverse of each measurement and normalising it to the range zero to one. To diminish the effect of outliers we use fourth-order polynomial interpolation of the measurement results to improve curve fitting. With a fourth-order polynomial there can be more than one point matching the user-defined gradient. We use the point with the least number of threads as recommendation. Then, we determine the gradient between any two adjacent numbers of threads in the performance graph and compare it with a configurable threshold gradient (default: 10 degrees). The recommended number of threads is the highest number of threads for which the gradient towards using one more thread is above the gradient threshold. The gradient threshold is the crucial knob whether to tune for performance alone or for performance/energy trade-offs. At last, the entire recommendation table is compiled into the production SAC code, just in front of the corresponding WITH-loop.

5. Revising feedback-directed compilation

In [Grelck and Blom(2020)] we evaluate our approach using two different machines.The smaller one is equipped with two Intel Xeon quad-core E5620 processors with hyperthreading enabled. These eight hyperthreaded cores run at 2.4 GHz; the entire system has 24GB of memory. Our larger machine features four AMD Opteron 6168 12-core processors running at 1.9 GHz and has 128GB of memory. Both systems are operated in batch mode giving us exclusive access for the duration of our experiments. We refer to these systems as the Intel and as the AMD system, respectively, from here on.

As microbenchmark, to start with, we use the element-wise addition of two matrices. Since we are interested in stressing the platform architecture as well as our multi-core runtime system, we investigate two fairly small problem sizes of 400x400 elements and 50x50 elements, respectively. The rationale here is that it is typically much easier to obtain good performance with parallel execution if the problem sizes are large. We show the results of our experiments in the form of speedup graphs in Fig. 5 for the 16-core Intel system and in Fig. 6 for the 48-core AMD system.

What we can observe is that our original feedback-directed does not necessarily find the sweet spot in parallelisation that we are aiming for. We identify as the main culprit the interpolation of training data by a fourth-order polynomial. This choice was motivated to smooth training data under the assumption that performance curves behave (roughly) as shown in Fig. 1 and we mainly would need to identify the characteristic parameters of said curve for each individual array comprehension, data size and platform.

However, as Fig. 5 and in Fig. 6 reveal, this is not necessarily the case. Furthermore, (relatively) small problem sizes, the ones we are most interested in, are particularly prone to runtime performance behaviour that does not follow the characteristics of Fig. 1.

The 16-core Intel machine effectively only has 8 cores that are hyperthreaded and thus appear to the operating system as 16 cores. As a consequence, we observe in Fig. 5 that speedups grow linearly up to 8 cores, then drop sharply when using 9 cores and, again, rise thereafter. Such behaviour significantly irritates our polynomial interpolation, and the feedback mechanism recommends to use

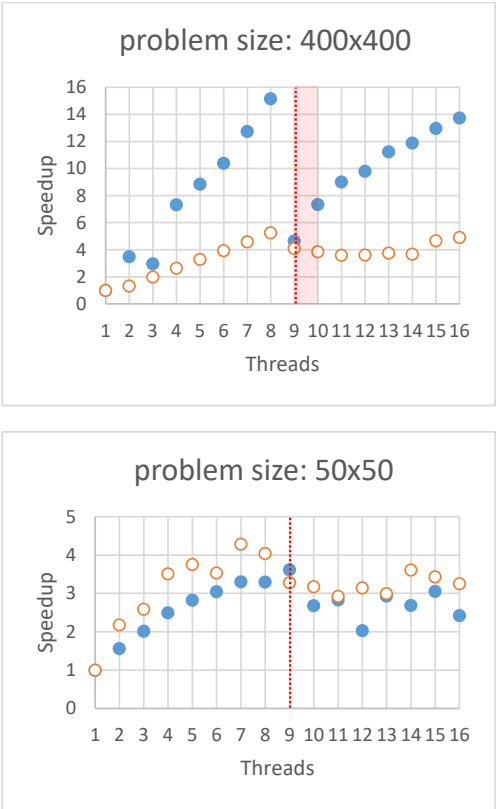


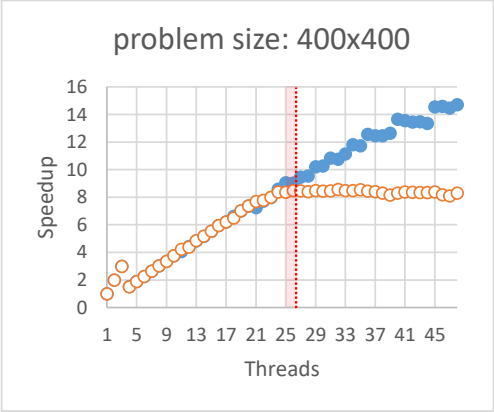
Figure 5. Performance on Intel 8-core hyperthreaded system with (red) and without (solid blue) feedback-directed compilation for two different problem sizes; feedback-directed recommendations: 9 and 9.

9 cores, one off from the actual best-performing configuration of 8 cores.

The 48-core AMD machine has 4 processors with 12 cores each, organised as a cc-NUMA architecture. This results in excellent speedups when using one core per processor, but in a temporary decline in performance when going beyond. This decline is quickly amortised when increasing the number of cores uses, but the decline suffices to, again, irritate our polynomial interpolation. As a result, we observe for the larger problem size that our tooling recommends to only use 26 cores, although the vanilla speedup figures scale all the way up to 48 cores. For the smaller problem size, the recommendation of 24 cores is decent for absolute performance, but suboptimal for energy concerns as 12 cores roughly deliver the same performance.

In consequence of the empirical data we obtained with feedback-directed compiation we give up on fourth-order polynomial interpolation and actually on interpolation at all. Instead, we interpret the raw data and draw conclusions from those. This fairly minor change to our implementation avoids the artifacts of interpolation that we observe in Fig. 5 and in Fig. 6.

So far, we have always trained on all possible core counts. This yields a complete performance overview, but with continuously growing core counts, training time might become infeasible, or at least undesirable. We propose to refine our training programme to start with powers of two numbers of cores and then to run additional experiments based on a feedback-directed oracle that strategically



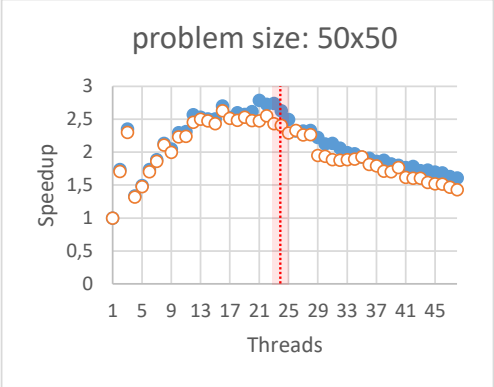


Figure 6. Performance on AMD 48-core system with (red) and without (solid blue) feedback-directed compilation for two different problem sizes; feedback-directed recommendations: 26 and 24.

aims at identifying interesting turning points in performance characteristics.

6. Feedback-directed compilation for heterogeneous platforms

Despite their architectural peculiarities both our experimental systems are in principle uniform in the sense that all cores are of the same kind and would deliver the very same performance when executed in isolation. In particular on mobile platforms and whenever energy consumption is of a particular concern, this is less and less the case. While we see special-purpose compute hardware on the rise that requires custom binary code, ARM has pioneered the big.LITTLE architectural model. Here, binary-compatible cores could exeute the very same code, but with very different performance characteristics in both execution time and energy consumption. Popular examples of these architectures are the Odroid family of boards or NVidia's Jetson product line.

Such platforms create particular challenges for implicit or automatic support for parallelism. Here, compiler and runtime system must not only decide about the best number of cores given the specific user preferences, but thay must decide about the best number of cores in each category of cores, usually high performance cores with high power draw vs low performance cores with low power draw. The optimal choice could be in using only one category of cores or some cores of either category or even all cores available.

Since this problem escapes any static analysis even more than our original problem, we propose to leverage our existing feedback-oriented compilation infrastructure to address this problem as well.

With thread pinning we have complete control over which cores we use during the training phase. Hence, we can first apply our existing solution to each kind of cores individually, and then expand the training to combinations of core types. Previously, we have used a very simple metric for energy consumption: using fewer cores is better than using more cores. With heterogeneous architectures, the problem becomes more tricky. However, we can leverage the performance counters typically available on modern hardware to create a multi-dimensional performance/energy map.

7. Conclusions and future work

Feedback-directed compilation is a one key to high performance in high-level function array prgramming. In particular for automatic parallelisation the design space for code generation significantly grows. With today's dozens or even hundreds of cores, the question where to effectively exploit which level of cincurrency becomes a relevant problem. Feedback-directed compilation in SAC addresses this problem while mostly retaining the promise of functional programming to program what to compute but not how.

Unfortunately, some design aspects of our initial solution, namely interpolation by fourth-order polynomials, do not stand the test of time. In this work we analyse the issues and revise this design choice. Moreover, we propose an extension of our feedback-directed compilation scheme to improve load balancing on heterogeneous platforms without incurring the overhead of dynamic load balancing schemes.

Both the Odroid family of boards or NVidia's Jetson product line also feature an onboard GPU. However, for the time being we ignore the GPU in our mapping decisions. In the future, we plan to leverage our feedback-directed compilation scheme to include GPUs. This would be very interesting beyond platforms such as Odroid and Jetson and likewise apply to homogeneous CPU architectures accompanied by one or multiple GPUs for workload acceleration.

References

[Bailey et al.(1991)] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, T.A. Schreiber, R.S. Simon, V. Venkatakrishnam, and S.K. Weeratunga. 1991. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications* 5, 3 (1991), 63–73.

[Catanzaro et al.(2010)] B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B. Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. 2010. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro* 30, 2 (2010), 41–55.

[Diogo and Grelck(2013)] M. Diogo and C. Grelck. 2013. Towards Heterogeneous Computing without Heterogeneous Programming. In *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK (Lecture Notes in Computer Science, Vol. 7829)*, K. Hammond and H.W. Loidl (Eds.). Springer, 279–294.

[Grelck(2002)] Clemens Grelck. 2002. Implementing the NAS Benchmark MG in SAC. In *16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, USA*, Viktor K. Prasanna and George Westrom (Eds.). IEEE Computer Society Press.

[Grelck(2003)] Clemens Grelck. 2003. A Multithreaded Compiler Backend for High-Level Array Programming. In *2nd International Conference on Parallel and Distributed Computing and Networks (PDCN'03), Innsbruck, Austria*, Mohammed H. Hamza (Ed.). ACTA Press, 478–484.

[Grelck(2005)] Clemens Grelck. 2005. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* 15, 3 (2005), 353–401.

[Grelck(2012)] C. Grelck. 2012. Single Assignment C (SAC): High Productivity meets High Performance. In *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary (Lecture Notes in Computer Science, Vol. 7241)*, V. Zsók, Z. Horváth, and R. Plasmeijer (Eds.). Springer, 207–278.

[Grelck and Blom(2020)] C. Grelck and C. Blom. 2020. Resource-aware Data Parallel Array Processing. *International Journal of Parallel Programming* 48, 4 (2020), 652–674.

[Grelck and Scholz(2006a)] Clemens Grelck and Sven-Bodo Scholz. 2006a. Merging Compositions of Array Skeletons in SAC. *Journal of Parallel Computing* 32, 7+8 (2006), 507–522.

[Grelck and Scholz(2006b)] Clemens Grelck and Sven-Bodo Scholz. 2006b. SAC: A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427.

[Guo et al.(2011)] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*. ACM Press, 15–24.

[Macht and Grelck(2019)] T. Macht and C. Grelck. 2019. SAC Goes Cluster: Fully Implicit Distributed Computing. In *33rd International Parallel and Distributed Processing Symposium (IPDPS'19), Rio de Janeiro, Brazil*. IEEE Computer Society Press.

[Nieplocha et al.(2007)] J. Nieplocha, A. M'arquez, J. Feo, D. Chavarria-Miranda, G. Chin, C. Scherrer, and N. Beagley. 2007. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *4th International Conference on Computing Frontiers (CF'07)*. ACM, 47–58.

[Pusukuri et al.(2011)] K.K. Pusukuri, R. Gupta, and L.N. Bhuyan. 2011. Thread Reinforcer: Dynamically determining number of threads via OS level monitoring. In *IEEE International Symposium on Workload Characterization (IISWC'11), Austin, TX, USA*. IEEE Computer Society, 116–125.

[Saini et al.(2006)] S. Saini et al. 2006. A scalability study of Columbia using the NAS parallel benchmarks. *Journal of Computational Methods in Science and Engineering* SI, 1 (2006), 33–45.

[Suleman et al.(2008)] M.A. Suleman, M.K. Qureshi, and Y.N. Patt. 2008. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII), Seattle, WA, USA*. ACM, 277–286.

A Quantum-Control Lambda-Calculus with Multiple Measurement Bases*

Alejandro Díaz-Caro
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France
Dpto. de Ciencia y Tecnología. Universidad Nacional de Quilmes.
Bernal, Buenos Aires, Argentina.
alejandro.diaz-caro@inria.fr

Nicolas A. Monzon
Universidad de la República, PEDECIBA-Informática
Montevideo, Uruguay
Universidad Argentina de la Empresa
Ciudad Autónoma de Buenos Aires, Argentina
nimonzon@uade.edu.ar

Abstract

We introduce Lambda-SX, a typed quantum lambda-calculus that supports multiple measurement bases. By tracking duplicability relative to arbitrary bases within the type system, Lambda-SX enables more flexible control and compositional reasoning about measurements. We formalise its syntax, typing rules, subtyping, and operational semantics, and establish its key meta-theoretical properties. This proof-of-concept shows that support for multiple bases can be coherently integrated into the type discipline of quantum programming languages.

1 Introduction

Quantum computing is a computing paradigm to model quantum mechanics. We can think quantum states and its evolution analogous to a computation process. We can create a language model then, and the study of it with language theory. In particular, we can add type theory and correctness properties on it development. Eventually, this create connections with the Curry–Howard isomorphism [19]. Recently, we observe a possible connection with linear logic and may it lead a formal study of logic on quantum mechanics.

Quantum algorithms are traditionally described using circuits, but the need for higher-level abstractions led to the notion of classical control, where a classical computer drives quantum execution. This idea, rooted in Knill's qRAM model [15], was formalised by Selinger [17] to enable classical control flow over quantum hardware. This approach led to the development of the Quantum Lambda Calculus [18], where programs are expressed by a tuple of a lambda term together with a quantum memory. This calculus has been the basis of languages like Quipper [13] and QWIRE [16].

An alternative paradigm is *quantum control*, introduced by Altenkirch and Grattage in the language QML [1]. Here, the goal is to avoid relying on a classical machine to drive a quantum computer, and instead allow quantum data to control computation directly. Following this paradigm, a quantum-control extension of the lambda calculus—later called Lambda-S_1—was proposed in 2019 [8], using realizability techniques [14], and given a categorical model in [10].

Lambda-S_1 was the result of a long line of research on quantum control, started by Lineal [2]—the first extension of the lambda calculus to embody quantum control. Lineal is an untyped lambda calculus extended with arbitrary linear superpositions. Its rewrite rules ensure confluence and avoid cloning arbitrary terms—a forbidden operation in quantum computing [20]—and terms normalize to canonical vector forms. To prevent cloning, it uses a *call-by-base* strategy: applying a lambda abstraction $\lambda x.t$ to a superposition $(\alpha.v + \beta.w)$ yields $\alpha.(\lambda x.t)v + \beta.(\lambda x.t)w$. This guarantees that all abstractions are linear and supports expressing matrices, vectors, and hence quantum programs. These include non-unitary maps and unnormalised vectors.

However, call-by-base breaks down in the presence of measurement. For instance, if $\lambda x.\pi^1 x$ denotes a measurement on the computational basis, then applying it to a superposition yields $\alpha.(\lambda x.\pi^1 x)v + \beta.(\lambda x.\pi^1 x)w$, which fails to produce a probabilistic collapse and instead behaves like the identity.

To solve this, Lambda-S [7] introduced a type-guided approach. In Lambda-S, a superposed term of type A is marked with $S(A)$, allowing beta-reduction to be guided by the argument's type. If $\mathbb{B}$ is the type of base qubits $|0\rangle$ and $|1\rangle$, then $S(\mathbb{B})$ is the type of arbitrary qubits. Thus, in $(\lambda x^{\mathbb{B}}.t)(\alpha.|0\rangle+\beta.|1\rangle)$, call-by-base applies, whereas in $(\lambda x^{S(\mathbb{B})}.t)(\alpha.|0\rangle + \beta.|1\rangle)$, a call-by-name strategy is used. The latter requires a linearity check on t: the variable must not be duplicated.

This modal distinction is dual to that of linear logic [12], where types $!A$ are duplicable. In Lambda-S, $S(A)$ marks non-duplicable types—and this duality is made explicit by its categorical models [9, 11].

Among various quantum lambda-calculus extensions, Lambda-S stands out for its ability to distinguish between superposed states and base states with respect to a given measurement basis.

Lambda-S_1 can be seen as a restriction of Lambda-S in which only unitary matrices and normalized vectors are considered. The technique to enforce this restriction was introduced in [8], and a full definition of the restricted language was given in [10], merging Lambda-S with that technique.

These languages favour the use of the computational basis, which is sufficient for quantum computation. Indeed, a measurement in an arbitrary basis can always be simulated by a rotation, followed by a measurement in the computational basis, and then a rotation back. However, restricting to a single basis introduces two important drawbacks.

First, duplicability is not unique to the computational basis: it is allowed in any basis, as long as the basis is known. Therefore, if we can determine that a quantum state is in a given basis, we can treat it as classical information.

*This is an extended abstract of an accepted paper at APLAS 2025 [6].

IFL 2025, Montevideo, Uruguay
Draft submission for talk.

$$
\begin{array}{rr}
\nu := \mathbb{B} \mid \mathbb{X} & \text{Atomic types (\textbf{A})} \\
\Psi := \mathbb{M} \mid S(\Psi) \mid \Psi \times \Psi & \text{Qubit types (\textbf{Q})} \\
\mathbb{M} := \nu \mid \mathbb{M} \times \mathbb{M} & \text{Base types (\textbf{B})} \\
A := \Psi \mid \Psi \Rightarrow A \mid S(A) & \text{Types (\textbf{T})}
\end{array}
$$

Figure 1: Type Grammar

$$
\frac{}{A \preceq A} \quad \frac{A \preceq B \quad B \preceq C}{A \preceq C} \quad \frac{}{A \preceq S(A)}
$$

$$
\frac{}{S(S(A)) \preceq S(A)} \quad \frac{}{\prod_{i=0}^{n} \nu_i \preceq S(\prod_{i=0}^{n} \nu'_i)}
$$

$$
\frac{A \preceq B}{S(A) \preceq S(B)} \quad \frac{A \preceq B \quad \Psi_1 \preceq \Psi_2}{\Psi_2 \Rightarrow A \preceq \Psi_1 \Rightarrow B} \quad \frac{\Psi_1 \preceq \Psi_2 \quad \Psi_3 \preceq \Psi_4}{\Psi_1 \times \Psi_3 \preceq \Psi_2 \times \Psi_4}
$$

Figure 2: Subtyping relation

$$
\begin{array}{rr}
t := x \mid \lambda x^{\Psi}.t \mid tt & \text{(Lambda calculus)} \\
\mid |0\rangle \mid |1\rangle \mid |+\rangle \mid |-\rangle \mid ?t{\cdot}t \mid ?_{\mathbb{X}}t{\cdot}t & \text{(Constants)} \\
\mid \vec{0} \mid t + t \mid \alpha.t \mid \lightning \mid \pi^m t \mid \pi^m_{\mathbb{X}} t & \text{(Linear combinations)} \\
\mid t \otimes t \mid \mathsf{hd}\ t \mid \mathtt{tl}\ t \mid \Uparrow^{\ell} t \mid \Uparrow^{r} t & \text{(Lists)}
\end{array}
$$

Figure 3: Preterms

Second, while Lambda-S and Intuitionistic Linear Logic (ILL) can be seen as categorical duals—via an adjunction between a Cartesian closed category and a monoidal category, where Lambda-S is interpreted in the Cartesian side and superpositions are captured by a monad, while ILL is interpreted in the monoidal side with duplicable data captured by a comonad—this duality is not complete. The asymmetry arises from Lambda-S being defined relative to a fixed basis, while ILL does not favour any particular basis.

In this extended abstract, we take a first step toward addressing this limitation by extending Lambda-S to track duplicability with respect to multiple bases. We present a proof-of-concept system that remains first-order for simplicity; the rationale and consequences of this choice are discussed in Section 2.2. Furthermore, we restrict attention to single-qubit bases, extended pointwise to non-entangled multi-qubit systems. Supporting entangled measurement bases would require additional complexity, and we leave such extensions for future work. These and other simplifications are intentional: our goal is not to provide a fully general system, but to highlight a specific capability that has not been explored in the literature so far—the ability to track duplicability with respect to multiple bases.

For further details and a complete version of this work, see our paper available on arXiv.

2 Lambda-SX

2.1 Types and terms

We consider two measurement bases: the computational basis, denoted by the type $\mathbb{B}$, and the Hadamard basis, denoted by the type $\mathbb{X}$. The set **B** of base types is defined as $\{\mathbb{B}, \mathbb{X}\}$, closed under Cartesian product, as shown in Figure 1.

Qubit types may be base types, their spans (denoted by the modality S), or Cartesian products. The language is first-order: function types are only allowed over qubit types. We work modulo associativity of the product, and parentheses are therefore omitted. We also use the notation $\prod_{i=1}^{n} \Psi_i$ to denote $\Psi_1 \times \ldots \times \Psi_n$.

We define a subtyping relation, shown in Figure 2. The intuition behind subtyping is that it corresponds to set inclusion. For example, $A \preceq S(A)$ holds because any set is included in its span, and $S(S(A)) \preceq S(A)$ reflects the fact that the span operation is idempotent. If $A \preceq B$ and $B \preceq A$, then A and B are considered *equivalent types*, and we write $A \approx B$. If A and B are syntactically identical, we write $A = B$.

The set of *preterms* is denoted by Λ and is defined by the grammar shown in Figure 3. As usual in algebraic calculi [2, 3, 8], the symbol + is treated as associative and commutative, so preterms are considered modulo these equational laws. The grammar includes first-order lambda calculus terms, constants (and their conditionals– we write $t?r{\cdot}s$ as a shorthand for $(?r{\cdot}s)t$), linear combinations (with measurement as a destructor), and tensor product terms, written using list notation since product types are considered associative.

The symbol $\lightning$ denotes an error and is used to handle measurements of the zero vector when normalisation fails. The measurement operations π^m and $\pi^m_{\mathbb{X}}$ are responsible for normalising their input prior to measurement. The casting operations $\Uparrow^{\ell}$ and $\Uparrow^{r}$ allow converting between lists of superpositions and superpositions of lists. Indeed, lists are used to represent tensor products. Consequently, a tensor product of superpositions can be regarded as a superposition of tensor products, which loses information about separability. We may use $\Uparrow$ to denote either $\Uparrow^{\ell}$ or $\Uparrow^{r}$, depending on the context.

Free variables are defined as usual, and the set of free variables of a preterm t is denoted by $\mathsf{FV}(t)$. The sets of base terms ($\mathcal{B}$) and values ($\mathcal{V}$) are defined by:

$$
\begin{array}{rr}
b := |0\rangle \mid |1\rangle \mid |+\rangle \mid |-\rangle \mid b \otimes b & \text{Base terms } (\mathcal{B}) \\
v := x \mid \lambda x^{\Psi}.t \mid b \mid \vec{0} \mid v + v \mid \alpha.v \mid v \otimes v & \text{Values } (\mathcal{V})
\end{array}
$$

The type system is presented in Figure 4. A *term* is a preterm t for which there exists a context Γ and a type A such that $\Gamma \vdash t : A$ is derivable.

2.2 Operational semantics

The operational semantics for terms is defined by the relation $\longrightarrow_p$, The parameter $p \in [0, 1]$ represents a probability and is primarily used in the probabilistic reduction rule associated with measurement. Rule (β_{n}) is the standard call-by-name beta-reduction rule, which applies when the argument is not basis-typed. Rules (β_{b}), ($\mathsf{lin}^{+}_{\mathsf{r}}$), ($\mathsf{lin}^{\alpha}_{\mathsf{r}}$), and ($\mathsf{lin}^{0}_{\mathsf{r}}$) implement the call-by-base strategy [2], distributing the function over the argument when the bound variable is basis-typed. For example $(\lambda x^{\mathbb{B}}.\, x \otimes x)(|0\rangle + |1\rangle)$ reduces first to $(\lambda x^{\mathbb{B}}.\, x \otimes x)\,|0\rangle + (\lambda x^{\mathbb{B}}.\, x \otimes x)\,|1\rangle$ by rule ($\mathsf{lin}^{+}_{\mathsf{r}}$), and then to $|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle$ by rule (β_{b}). We have other rules for conditionals, vector space axioms, casting and measurement.

$$\frac{}{x^{\Psi} \vdash x : \Psi}\ Ax \qquad \frac{\Gamma, x^{\Psi} \vdash t : A}{\Gamma \vdash \lambda x^{\Psi}.t : \Psi \Rightarrow A}\ \Rightarrow_I$$

$$\frac{\Gamma \vdash t : \Psi \Rightarrow A \quad \Delta \vdash r : \Psi}{\Gamma, \Delta \vdash tr : A}\ \Rightarrow_E$$

$$\frac{\Gamma \vdash t : S(\Psi \Rightarrow A) \quad \Delta \vdash r : S(\Psi)}{\Gamma, \Delta \vdash tr : S(A)}\ \Rightarrow_{ES}$$

$$\frac{}{\vdash |0\rangle : \mathbb{B}}\ |0\rangle \qquad \frac{}{\vdash |1\rangle : \mathbb{B}}\ |1\rangle$$

$$\frac{}{\vdash |+\rangle : \mathbb{X}}\ |+\rangle \qquad \frac{}{\vdash |-\rangle : \mathbb{X}}\ |-\rangle$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash r : A}{\Gamma \vdash ?t{\cdot}r : \mathbb{B} \Rightarrow A}\ If \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash r : A}{\Gamma \vdash ?_{\mathbb{X}} t{\cdot}r : \mathbb{X} \Rightarrow A}\ If_{\mathbb{X}}$$

$$\frac{}{\vdash \vec{0} : S(A)}\ \vec{0} \qquad \frac{\Gamma \vdash t : A \quad \Delta \vdash r : A}{\Gamma, \Delta \vdash t + r : S(A)}\ S_I^+ \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \alpha.t : S(A)}\ S_I^{\alpha}$$

$$\frac{}{\Gamma \vdash \lightning : \Psi}\ \mathrm{e}$$

$$\frac{\Gamma \vdash t : S\left(\prod_{i=1}^{n} \nu_i\right) \quad 0 < m \leq n}{\Gamma \vdash \pi^m t : \mathbb{B}^m \times S\left(\prod_{i=m+1}^{n} \nu_i\right)}\ S_E$$

$$\frac{\Gamma \vdash t : S\left(\prod_{i=1}^{n} \nu_i\right) \quad 0 < m \leq n}{\Gamma \vdash \pi_{\mathbb{X}}^m t : \mathbb{X}^m \times S\left(\prod_{i=m+1}^{n} \nu_i\right)}\ S_{E_{\mathbb{X}}}$$

$$\frac{\Gamma \vdash t : \Psi \quad \Delta \vdash r : \Phi}{\Gamma, \Delta \vdash t \otimes r : \Psi \times \Phi}\ \times_I$$

$$\frac{\Gamma \vdash t : \nu \times \mathbb{M}}{\Gamma \vdash \mathtt{hd}\ t : \nu}\ \times_{Er} \qquad \frac{\Gamma \vdash t : \nu \times \mathbb{M}}{\Gamma \vdash \mathtt{tl}\ t : \mathbb{M}}\ \times_{El}$$

$$\frac{\Gamma \vdash t : S(\Psi \times S(\Phi)) \quad \Psi \neq S(\Psi')}{\Gamma \vdash \Uparrow^{\ell} t : S(\Psi \times \Phi)}\ \Uparrow^{\ell}$$

$$\frac{\Gamma \vdash t : S(S(\Phi) \times \Psi) \quad \Psi \neq S(\Psi')}{\Gamma \vdash \Uparrow^{r} t : S(\Phi \times \Psi)}\ \Uparrow^{r}$$

$$\frac{\Gamma \vdash t : \mathbb{X}}{\Gamma \vdash \Uparrow t : S(\mathbb{B})}\ \Uparrow_{\mathbb{X}} \qquad \frac{\Gamma \vdash t : \mathbb{B}}{\Gamma \vdash \Uparrow t : \mathbb{B}}\ \Uparrow_{\mathbb{B}}$$

$$\frac{\Gamma \vdash t : A \quad A \leq B}{\Gamma \vdash t : B}\ \leq \qquad \frac{\Gamma \vdash t : A}{\Gamma, x^{\mathbb{M}} \vdash t : A}\ W$$

$$\frac{\Gamma, x^{\mathbb{M}}, y^{\mathbb{M}} \vdash t : A}{\Gamma, x^{\mathbb{M}} \vdash t[y/x] : A}\ C$$

Figure 4: Type system

We restrict the calculus to first-order terms for simplicity. In a higher-order setting, one could consider the term $\lambda x^{S(\mathbb{B})}.\lambda y^{\mathbb{B}}.x$, which embed an unknown qubit within a perfectly duplicable lambda abstraction. Several solutions are possible: restricting weakening to non-arrow types, restricting the language to first-order, or introducing annotations that prevent duplication of such terms. In this work, we adopt the second option, as our goal is to provide a proof-of-concept system for handling multiple measurement bases.

Figure 5 presents the reduction rules *schemas* for measurement: each instantiation depends on the specific shape of the term being measured.

The operation π^m applies to a term of type $S(\prod_{i=1}^{n} \nu_i)$. Before measurement, the term is implicitly converted to the computational basis, yielding a sum of distinct basis vectors: $\sum_{a=1}^{f} \beta_a |c_{a1}\rangle \otimes \cdots \otimes |c_{an}\rangle$, with $c_{aj} \in \{0, 1\}$. Measurement is performed on the first m qubits, producing a collapse to $|k\rangle \otimes |\phi_k\rangle$ with probability $p_k = \frac{1}{Z} \sum_{a \in I_k} |\beta_a|^2$, where I_k is the set of indices a such that the prefix $|c_{a1} \cdots c_{am}\rangle$ equals $|k\rangle$, and Z is the squared norm of the original input.

$$\pi^m \left(\sum_{i=1}^{e} [\alpha_i.] \bigotimes_{h=1}^{n} |b_{hi}\rangle \right) \longrightarrow_{p_k} |k\rangle \otimes |\phi_k\rangle \qquad (\mathrm{proy})$$

$$\pi^m \vec{0} \longrightarrow_1 \lightning \qquad (\mathrm{proy}^0)$$

$$\pi_{\mathbb{X}}^m \left(\sum_{i=1}^{e} [\alpha_i.] \bigotimes_{h=1}^{n} |b_{hi}\rangle \right) \longrightarrow_{p_k} |k\rangle \otimes |\phi_k\rangle \qquad (\mathrm{proy}_{\mathbb{X}})$$

$$\pi_{\mathbb{X}}^m \vec{0} \longrightarrow_1 \lightning \qquad (\mathrm{proy}_{\mathbb{X}}^0)$$

Figure 5: Measurement rules for the computational and Hadamard bases

The state $|\phi_k\rangle$ is defined by normalising the suffixes of the terms in I_k:

$$|\phi_k\rangle = \sum_{a \in I_k} \frac{\beta_a}{\sqrt{\ell}} |c_{a,m+1}\rangle \otimes \cdots \otimes |c_{an}\rangle, \qquad \text{where } \ell = \sum_{a \in I_k} |\beta_a|^2.$$

The rule for measurement in the Hadamard basis, $\pi_{\mathbb{X}}^m$, behaves analogously, with the input expressed in the Hadamard basis and $|k\rangle$ ranging over $\{+, -\}^m$.

If the input to π^m or $\pi_{\mathbb{X}}^m$ is or reduces to $\vec{0}$, the result is $\lightning$.

Example 2.1 (Measurement). To simplify notation, we write $|abcd\rangle$ instead of $|a\rangle \otimes |b\rangle \otimes |c\rangle \otimes |d\rangle$. Consider the following two semantically equivalent terms: $\pi^2(\alpha |0 + 10\rangle + \beta |10 - 0\rangle)$ and $\pi^2(\frac{\alpha}{\sqrt{2}} |0010\rangle + \frac{\alpha}{\sqrt{2}} |0110\rangle + \frac{\beta}{\sqrt{2}} |1000\rangle - \frac{\beta}{\sqrt{2}} |1010\rangle)$. Both reduce, for instance, to $|10\rangle \otimes (\frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |01\rangle)$ with probability $\frac{|\alpha|^2}{\sqrt{|\alpha|^2 + |\beta|^2}}$.

2.3 Examples

This section illustrates how the type system supports quantum states, operations, and measurements in a functional style, highlighting its flexibility across multiple measurement bases.

Example 2.2 (Hadamard gate). The Hadamard gate can be implemented in multiple ways, depending on the desired type. For example:

$$\vdash \lambda x^{\mathbb{B}}.x?|-\rangle{\cdot}|+\rangle : \mathbb{B} \Rightarrow \mathbb{X}$$

$$\vdash \lambda x^{\mathbb{X}}.x?|0\rangle{\cdot}|1\rangle : \mathbb{X} \Rightarrow \mathbb{B}$$

$$\vdash \lambda x^{\mathbb{B}}.x?(\Uparrow |-\rangle){\cdot}(\Uparrow |+\rangle) : \mathbb{B} \Rightarrow S(\mathbb{B})$$

All these implementations yield equivalent results on arbitrary inputs (in either basis), but the first preserves duplicability on inputs $|0\rangle$ or $|1\rangle$, and the second does so on $|+\rangle$ or $|-\rangle$.

As an instance, we could write $\lambda x^{\mathbb{B}}.(\lambda y^{\mathbb{X}}.y \otimes y)Hx$ as soon as H is the first implementation of the Hadamard gate. Notice that we are cloning a qubit; however, since the basis is tracked since the beginning, this is perfectly valid.

Example 2.3 (CNOT gate). In the same way, the CNOT gate can be implemented in multiple ways, for example

$$\vdash \lambda x^{\mathbb{B}\times\mathbb{B}}.(\mathtt{hd}\ x) \otimes (\mathtt{hd}\ x?\mathbf{NOT}(\mathtt{tl}\ x){\cdot}\mathtt{tl}\ x) : \mathbb{B} \times \mathbb{B} \Rightarrow \mathbb{B} \times \mathbb{B}$$

$$\vdash \lambda x^{\mathbb{X}\times\mathbb{B}}.|0\rangle \otimes (\mathtt{tl}\ x) + |1\rangle \otimes ((\mathtt{hd}\ x?_{\mathbb{X}}\mathbf{NOT}{\cdot}(-1).\mathbf{NOT})(\mathtt{tl}\ x)) : \mathbb{X} \times \mathbb{B} \Rightarrow S(\mathbb{B} \times \mathbb{B})$$

where **NOT** is the NOT gate given by $\vdash \lambda x^{\mathbb{B}}.x?|0\rangle{\cdot}|1\rangle : \mathbb{B} \Rightarrow \mathbb{B}$.

Example 2.4 (Bell states). The entangled Bell states can be produced by the following term, applied to a pair of computational basis qubits:

$$\vdash \lambda x^{\mathbb{B}\times\mathbb{B}}.\mathbf{CNOT}(\mathbf{H}(\mathsf{hd}\ x)\otimes \mathtt{tl}\ x) : \mathbb{B}\times\mathbb{B} \Rightarrow S(\mathbb{B}\times\mathbb{B})$$

However, there are more interesting implementations. For example, the following term maps $|+\rangle$ to the Bell state β_{00} and $|-\rangle$ to β_{10}:

$$\vdash \lambda x^{\mathbb{X}}.(\lambda y^{\mathbb{B}}.y\otimes y)(\Uparrow x) : \mathbb{X} \Rightarrow S(\mathbb{B}\times\mathbb{B})$$

Example 2.5 (Applying gates to multi-qubit states). In general, we can apply a gate to a qubit in a multi-qubit state, even if entangled, with the same technique as used to apply Hadamard to the first qubit in the Bell state. For example, applying CNOT to the first of three qubits can be done as follows:

$$\mathbf{CNOT}^3_{1,2} = \lambda x^{\mathbb{B}\times\mathbb{B}\times\mathbb{B}}.\mathbf{CNOT}((\mathsf{hd}\ x)\otimes(\mathtt{hd}\ \mathtt{tl}\ x))\otimes(\mathtt{tl}\ \mathtt{tl}\ x)$$

Example 2.6 (Teleportation). We can use the Bell state to implement teleportation, which allows the transmission of an arbitrary qubit state from Alice to Bob using an entangled pair and classical communication. The term implementing it would be

$$\vdash \lambda x^{S(\mathbb{B})}.\pi^2\ \Uparrow^\ell\ \mathbf{Bob}(\Uparrow^\ell\ (\mathbf{Alice}(x\otimes\mathbf{Bell}(|0\rangle\otimes|0\rangle)))) : S(\mathbb{B}) \Rightarrow (\mathbb{B}\times\mathbb{B}\times S(\mathbb{B}))$$

where: **Bell** produces the Bell state, as defined earlier; **Alice** implements Alice's part of the protocol, defined as: $\lambda x^{S(\mathbb{B})\times S(\mathbb{B}\times\mathbb{B})}.\pi^2(\Uparrow^r \mathbf{H}^3_1(\mathbf{CNOT}^3_{1,2}(\Uparrow^\ell\ (\Uparrow^r\ x)))))$, where $\mathbf{H}^3_1$ is the Hadamard operator on the first qubit: $\lambda x^{\mathbb{B}\times\mathbb{B}\times\mathbb{B}}.\mathbf{H}(\mathsf{hd}\ x)\otimes\mathtt{tl}\ x$. **Bob** implements Bob's part: $\lambda x^{\mathbb{B}\times\mathbb{B}\times\mathbb{B}}.(\mathsf{hd}\ x)\otimes(\mathtt{hd}\ \mathtt{tl}\ x)\otimes(\mathbf{C\text{-}Z}(\mathsf{hd}\ x)\otimes(\mathbf{CNOT}(\mathtt{hd}\ \mathtt{tl}\ x)\otimes(\mathtt{tl}\ \mathtt{tl}\ x)))$, where **C-Z** is: $\lambda x^{\mathbb{B}\times\mathbb{B}}.\mathsf{hd}\ x?\mathbf{Z}(\mathtt{tl}\ x)\cdot\mathtt{tl}\ x$.

3 Correctness

In this section we establish the main meta-theoretical properties of our calculus. These results show that the type system is well-behaved with respect to the operational semantics, and they guarantee consistency.

We begin in Section 3.1 with the proof of *subject reduction* (Theorem 3.4), showing that the type of a term is preserved under reduction. We then prove *progress* (Theorem 3.5), ensuring that well-typed terms are either values or can take a reduction step. The *linear casting* property follows (Theorem 3.6), showing that terms of type $S(\mathbb{B}^n)$ can be rewritten—via explicit casting reductions—as linear combinations of terms of type $\mathbb{B}^n$. This result provides a semantic justification for viewing casting as a projection onto a measurement basis.

Finally, we show that all well-typed terms are *strongly normalising* (Section 3.2); that is, all well-typed terms always terminate.

3.1 Type soundness

3.1.1 Subject reduction. The typing rules are not syntax-directed due to application (which has two typing rules), subtyping, weakening, and contraction. Therefore, a generation lemma—stating the conditions under which a typing judgment $\Gamma \vdash t : A$ can be derived—is needed.

The substitution lemma, which plays a central role in the proof of subject reduction, is stated as follows.

Lemma 3.1 (Substitution). *If* $\Gamma, x^A \vdash t : C$ *and* $\Delta \vdash r : A$, *then* $\Gamma,\Delta \vdash t[r/x] : C$. □

The type preservation property is strongly related to subtyping. In its proof, several auxiliary properties of the subtyping relation are repeatedly used. The following lemma states these properties.

Lemma 3.2 (Properties of the subtyping relation). *The subtyping relation* $\preceq$ *satisfies the following properties:*

1. *If* $A \Rightarrow B \preceq C \Rightarrow D$, *then* $C \preceq A$ *and* $B \preceq D$.
2. *If* $A \Rightarrow B \preceq S(C \Rightarrow D)$, *then* $C \preceq A$ *and* $B \preceq D$.
3. *If* $S(A) \preceq B$, *then* $B = S(C)$ *and* $A \preceq C$, *for some* C.
4. *If* $\Psi_1 \times \Psi_2 \preceq A$, *then* $A \approx S(\Psi_3 \times \Psi_4)$ *or* $A \approx \Psi_3 \times \Psi_4$, *for some* Ψ_3, Ψ_4.
5. *If* $A \preceq B$, *then* A *and* B *contain the same number of product constructors.*
6. *If* $A \preceq \Psi_1 \Rightarrow C$ *then* $A \approx \Psi_2 \Rightarrow D$ *for some* Ψ_2 *and* D.
7. *If* $S(A) \preceq S(\Psi_1 \Rightarrow C)$ *then* $S(A) \approx S(\Psi_2 \Rightarrow D)$, *for some* Ψ_2 *and* D.
8. *If* $S(\Psi_1 \Rightarrow A) \preceq S(\Psi_2 \Rightarrow B)$, *then* $\Psi_2 \preceq \Psi_1$ *and* $B \preceq A$.

Proof. The proof relies on several technical properties. Please, refer to the paper arXiv version for the full proof. □

The following properties are crucial for analysing cast elimination, as they constrain the shape of product-type subtypes and relate them to simpler forms.

Lemma 3.3 (Properties of the subtyping relation on products). *The subtyping relation* $\preceq$ *satisfies the following properties on product types:*

1. *If* $\varphi \times \mathbb{M} \preceq S(\Psi_1 \times S(\Psi_2))$, *then* $\varphi \times \mathbb{M} \preceq S(\Psi_1 \times \Psi_2)$.
2. *If* $\mathbb{M} \times \varphi \preceq S(S(\Psi_1) \times \Psi_2)$, *then* $\mathbb{M} \times \varphi \preceq S(\Psi_1 \times \Psi_2)$.

Proof. The proof relies on several technical lemmas. Please, refer to the paper arXiv version for the full proof. □

We are now ready to state the main result of this section: the subject reduction property, which ensures that types are preserved under reduction.

Theorem 3.4 (Subject reduction). $\Gamma \vdash t : A$ *and* $t \longrightarrow_p r$ *imply* $\Gamma \vdash r : A$.

Proof. By induction on the reduction $t \longrightarrow_p r$. As an illustrative case, consider rule $(\mathsf{lin}^+_\mathsf{l})$, where $t = (t_1 + t_2)t_3$ and $r = t_1t_3 + t_2t_3$, with $\Gamma \vdash t : A$. By the generation lemma, we have $\Gamma = \Gamma_1, \Gamma_2, \Xi$, $\mathsf{T}(\Xi) \subseteq \mathbf{B}$, $\Gamma_1, \Xi \vdash t_1 + t_2 : S(\Psi_1 \Rightarrow C)$, $\Gamma_2, \Xi \vdash t_3 : S(\Psi_1)$, and $S(C) \preceq A$. By the generation lemma again, $\Gamma_1, \Xi = \Gamma'_1, \Gamma'_2, \Xi'$ with $\Gamma'_1, \Xi' \vdash t_1 : D$, $\Gamma'_2, \Xi' \vdash t_2 : D$, and $S(D) \preceq S(\Psi_1 \Rightarrow C)$. Then Lemma 3.2.7 gives $S(D) \approx S(\Psi_2 \Rightarrow E)$ with $S(\Psi_2 \Rightarrow E) \preceq S(\Psi_1 \Rightarrow C)$, and Lemma 3.2.8 yields $\Psi_1 \preceq \Psi_2$ and $E \preceq C$. Hence, from $\Gamma'_1, \Xi' \vdash t_1 : S(\Psi_2 \Rightarrow E)$ and $\Gamma_2, \Xi \vdash t_3 : S(\Psi_2)$, we derive $\Gamma \vdash t_1t_3, t_2t_3 : S(E)$, and thus $\Gamma \vdash t_1t_3 + t_2t_3 : S(S(E))$. Since $S(S(E)) \preceq S(E) \preceq S(C) \preceq A$, by transitivity we conclude $\Gamma \vdash r : A$. □

3.1.2 Progress. The next result is the *progress* theorem, stating that well-typed terms in normal form must be either values or the error term.

Theorem 3.5 (Progress). *If* $\vdash t : A$ *then* t *reduces, is a value, or is* $\lightning$.

3.1.3 Linear casting. Our system also exhibits a property we call *linear casting*, which expresses the idea that terms of type $S(\mathbb{B}^n)$ can be written using other terms of type $\mathbb{B}^n$, thanks to our linear casting reduction rules. This is not trivial, as by subtyping we have $S(\mathbb{B}^n) = S(\prod_{i=1}^n \nu_i)$.

Theorem 3.6 (Linear casting theorem). $\vdash \Uparrow t : S(\mathbb{B}^n)$ *implies* $\Uparrow t \longrightarrow_1^* \sum [\alpha_i.]\, b_i$ *with* $\vdash b_i : \mathbb{B}^n$.

Proof. By Theorems 3.5 and 3.4, we know that $\Uparrow t \longrightarrow_1^* v$ with $\vdash v : S(\mathbb{B}^n)$. By the generation lemma, we distinguish the following cases:

- If $v = |0\rangle$ or $v = |1\rangle$, then $\vdash v : \mathbb{B}$.
- Note that $v \neq |+\rangle$, since $\Uparrow t \not\longrightarrow |+\rangle$: casting can only be eliminated through rules $(\text{cast}_{|+\rangle})$, $(\text{cast}_{|-\rangle})$, $(\text{cast}_{|0\rangle})$, or $(\text{cast}_{|1\rangle})$. Hence, this case cannot occur. Similarly, $v \neq |-\rangle$ for the same reason.
- If v has the form $b_1 \otimes b_2 \otimes \cdots \otimes b_n$ with $n \geq 1$, and each b_i a value, then each b_i must be either $|0\rangle$ or $|1\rangle$ by the same argument.
- If v has the form $\sum [\alpha_i.]\, b_i$, with the b_i having distinct types, then each b_i has the form $b_1 \otimes b_2 \otimes \cdots \otimes b_n$, where each $b_i \in \{|0\rangle, |1\rangle\}$.

Therefore, $v = \sum [\alpha_i.]\, b_i$, and $\Uparrow t \longrightarrow_1^* \sum [\alpha_i.]\, b_i$ with $\vdash b_i : \mathbb{B}^n$. □

3.2 Strong normalization

We conclude the correctness properties by showing that terms are *strongly normalising*, meaning that every reduction sequence eventually terminates.

Let SN denote the set of strongly normalising terms, and $|t|$ the number of reduction steps in a reduction sequence starting from t. We also write $\mathsf{Red}(t)$ for the set of one-step reducts of t, i.e. $\mathsf{Red}(t) = \{r \mid t \longrightarrow_p r\}$.

We start our proof by observing that, excluding rules (β_n) and (β_b), all other rules strictly decrease a well-defined measure. This measure is invariant under commutativity and associativity of addition, meaning that terms like $t+r$ and $r+t$, or $(t+r)+s$ and $t+(r+s)$, receive the same value. We define this measure in Definition 3.7 and prove these properties in Theorem 3.8.

Definition 3.7 (Measure). We define the following measure on terms:

$$\begin{aligned}
\|x\| &= 0 & \|\lambda x^{\Psi}.t\| &= \|t\| \\
\|\vec{0}\| &= 0 & \|tr\| &= (3\|t\| + 2)(3\|r\| + 2) \\
\|\lightning\| &= 0 & \|t \otimes r\| &= \|t\| + \|r\| + 1 \\
\|\,|0\rangle\,\| &= 0 & \|\Uparrow t\| &= \|t\| + 5 \\
\|\,|1\rangle\,\| &= 0 & \|\alpha.\Uparrow t\| &= \|\Uparrow t\| \\
\|\,|+\rangle\,\| &= 0 & \|\alpha.t\| &= 2\|t\| + 1 \\
\|\,|-\rangle\,\| &= 0 & \|\Uparrow t + \Uparrow r\| &= \max\{\|t\|, \|r\|\}
\end{aligned}$$

$$\begin{aligned}
\|\mathtt{hd}\ t\| &= \|t\| + 1 & \|t + r\| &= \|t\| + \|r\| + 2 \quad \text{(if not both are casts)} \\
\|\mathtt{tl}\ t\| &= \|t\| + 1 & \|?t{\cdot}r\| &= \|t\| + \|r\| \\
\|\pi^m t\| &= \|t\| + m & \|?_{\mathbb{X}} t{\cdot}r\| &= \|t\| + \|r\| \\
\|\pi^m_{\mathbb{X}} t\| &= \|t\| + m
\end{aligned}$$

Theorem 3.8 (Measure decrease and invariance). *The measure defined in Definition 3.7 satisfies the following properties:*

- *If* $t = r$ *by the commutativity or associativity properties of* $+$, *then* $\|t\| = \|r\|$.
- *If* $t \longrightarrow_p r$ *using a rule other than* (β_n) *and* (β_b), *then* $\|t\| > \|r\|$.

Proof. We only provide an example here.

$$\begin{aligned}
\|\alpha.(t + r)\| &= 1 + 2\|t + r\| = 5 + 2\|t\| + 2\|r\| = 3 + \|\alpha.t\| + \|\alpha.r\| \\
&= 1 + \|\alpha.t + \alpha.r\| > \|\alpha.t + \alpha.r\|
\end{aligned}$$

□

The previous result shows that all reduction sequences that do not involve β-redexes are strongly normalising. We now combine this with a standard reducibility argument to obtain the general strong normalisation theorem. As a first step, we show that linear combinations of strongly normalising terms are themselves strongly normalising.

Lemma 3.9 (Strong normalisation of linear combinations). *If* $r_i \in \mathsf{SN}$ *for all* $1 \leq i \leq n$, *then* $\sum_{i=1}^n [\alpha_i.]\, r_i \in \mathsf{SN}$.

Proof. By induction on the lexicographic order of $\left(\sum_{i=1}^n |r_i|,\ \|\sum_{i=1}^n [\alpha_i.]\, r_i\|\right)$. □

We now define the *interpretation of types* used in the strong normalisation argument. From this point onwards, we write $t : A$ to mean that the term t has type A in *any context*.

Definition 3.10 (Type interpretation). Given a type A, its interpretation $[\![A]\!]$ is defined inductively as follows:

$$\begin{aligned}
[\![\nu]\!] &= \{t : S(\nu) \mid t \in \mathsf{SN}\} \\
[\![\Psi_1 \times \Psi_2]\!] &= \{t : S(\Psi_1 \times \Psi_2) \mid t \in \mathsf{SN}\} \\
[\![\Psi \Rightarrow A]\!] &= \{t : S(\Psi \Rightarrow A) \mid \text{for all } r \in [\![\Psi]\!],\ tr \in [\![A]\!]\} \\
[\![S(A)]\!] &= \{t : S(A) \mid t \in \mathsf{SN},\ \exists p \text{ s.t. } t \\
&\qquad \longrightarrow_p^* \textstyle\sum_i [\alpha_i.] r_i,\ r_i \in [\![A]\!] \text{ on all paths}\}
\end{aligned}$$

with the convention that $\sum_{i=1}^0 [\alpha_i.] r_i = \vec{0}$.

Since our language is first-order, this interpretation is sufficient. Traditionally, type interpretations are defined either by introduction or elimination. In our case, however, this distinction is not necessary for product types. This is because we define their interpretation using a superposition type rather than a direct product. This design choice is motivated by the interpretation of function types, which requires certain linearity properties on the argument type—properties that hold only when the type is a superposition.

A term $t \in [\![A]\!]$ is said to be *reducible*, and an application tr is said to be *neutral*. We write $\mathcal{N}$ for the set of neutral terms. In particular, expressions of the form $t?r{\cdot}s$ are in $\mathcal{N}$, since this is shorthand for $(?r{\cdot}s)t$.

We now establish the main properties of reducibility. What we refer to as **LIN1** and **LIN2** are in fact specific instances of the more general *adequacy* property (Theorem 3.13).

Lemma 3.11 (Reducibility properties). *For every type A, the following hold:*

- **(CR1)** *If $t \in [\![A]\!]$, then $t \in \mathsf{SN}$.*
- **(CR2)** *If $t \in [\![A]\!]$, then $\mathsf{Red}(t) \subseteq [\![A]\!]$.*
- **(CR3)** *If $t : S(A)$, $t \in \mathcal{N}$, and $\mathsf{Red}(t) \subseteq [\![A]\!]$, then $t \in [\![A]\!]$.*
- **(LIN1)** *If $t \in [\![A]\!]$ and $r \in [\![A]\!]$, then $t + r \in [\![A]\!]$.*
- **(LIN2)** *If $t \in [\![A]\!]$, then $\alpha.t \in [\![A]\!]$.*
- **(HAB)** *$\vec{0} \in [\![A]\!]$, $\lightning \in [\![A]\!]$, and for every variable $x : A$, we have $x \in [\![A]\!]$.*

Proof. All of these properties are proved simultaneously by induction on the structure of the type. This unified presentation is essential, as several cases rely on the inductive hypotheses applied to their subcomponents. □

Lemma 3.12 (Compatibility with subtyping). *If $A \preceq B$, then $[\![A]\!] \subseteq [\![B]\!]$.* □

Theorem 3.13 (Adequacy). *If $\Gamma \vdash t : A$ and $\theta \vDash \Gamma$, then $\theta(t) \in [\![A]\!]$.*

Proof. By induction on the derivation of $\Gamma \vdash t : A$. □

Corollary 3.14 (Strong normalisation). *If $\Gamma \vdash t : A$, then $t \in \mathsf{SN}$.*

Proof. By Theorem 3.13, if $\theta \vDash \Gamma$, then $\theta(t) \in [\![A]\!]$. By Lemma 3.11.CR1, we have $[\![A]\!] \subseteq \mathsf{SN}$. Moreover, by Lemma 3.11.HAB, we know that $\mathtt{Id} \vDash \Gamma$. Therefore, $\mathtt{Id}(t) = t \in \mathsf{SN}$. □

4 An Arbitrary Number of Bases

4.1 Extending Lambda-SX with Multiple Distinct Bases

Up to this point, we have introduced Lambda-SX with two measurement bases: the computational basis $\mathbb{B}$ and an alternative basis $\mathbb{X}$. However, the design of the system do not need to be restricted to this choice. In fact, the calculus naturally generalises to an arbitrary collection of orthonormal bases.

Let $\mathbb{B}_i$ for $i = 1, \dots, n$ denote a set of alternative bases, where each $\mathbb{B}_i = \{|\uparrow_i\rangle, |\downarrow_i\rangle\}$ is defined by a change of basis from the computational basis: $|\uparrow_i\rangle = \alpha_{i1}\,|0\rangle + \beta_{i1}\,|1\rangle$ and $|\downarrow_i\rangle = \alpha_{i2}\,|0\rangle + \beta_{i2}\,|1\rangle$ for some $\alpha_{i1}, \alpha_{i2}, \beta_{i1}, \beta_{i2} \in \mathbb{C}$. We assume that all $\mathbb{B}_i$ are distinct from the computational basis $\mathbb{B}$ (which retains its special role), and that one of them may coincide with $\mathbb{X}$.

We extend the grammar of atomic types as follows:

$$\nu := \mathbb{B} \mid \mathbb{B}_i$$

and the grammar of terms with corresponding constants and operations:

$$t ::= \cdots \mid |\uparrow_i\rangle \mid |\downarrow_i\rangle \mid ?_{\mathbb{B}_i} t \cdot t \mid \pi^m_{\mathbb{B}_i} t$$

Typing rules are extended in the natural way:

$$\frac{}{\vdash |\uparrow_i\rangle : \mathbb{B}_i}\ |\uparrow_i\rangle \qquad \frac{}{\vdash |\downarrow_i\rangle : \mathbb{B}_i}\ |\downarrow_i\rangle$$

$$\frac{\Gamma \vdash t : S\left(\prod_{i=1}^n \nu_i\right) \quad 0 < m \leq n}{\Gamma \vdash \pi^m_{\mathbb{B}_i} t : \mathbb{B}_i^m \times S\left(\prod_{i=m+1}^n \nu_i\right)}\ S_{E_{\mathbb{B}_i}}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash r : A}{\Gamma \vdash ?_{\mathbb{B}_i} t \cdot r : \mathbb{B}_i \Rightarrow A}\ If_{\mathbb{B}_i} \qquad \frac{\Gamma \vdash t : \mathbb{B}_i}{\Gamma \vdash \Uparrow t : S(\mathbb{B})}\ \Uparrow_{\mathbb{B}_i}$$

The operational semantics is similarly extended with the following rules:

$$|\uparrow_i\rangle ?_{\mathbb{B}_i} t \cdot r \longrightarrow_1 t \qquad (\mathrm{if}_\uparrow)$$

$$|\downarrow_i\rangle ?_{\mathbb{B}_i} t \cdot r \longrightarrow_1 r \qquad (\mathrm{if}_\downarrow)$$

$$\Uparrow |\uparrow_i\rangle \longrightarrow_1 \alpha_{i1} \cdot |0\rangle + \beta_{i1} \cdot |1\rangle \qquad (\mathrm{cast}_{|\uparrow\rangle})$$

$$\Uparrow |\downarrow_i\rangle \longrightarrow_1 \alpha_{i2} \cdot |0\rangle + \beta_{i2} \cdot |1\rangle \qquad (\mathrm{cast}_{|\downarrow\rangle})$$

$$\pi^m_{\mathbb{B}_i}\left(\sum_{j=1}^e \alpha_j \cdot \bigotimes_{h=1}^n |b_{hj}\rangle\right) \longrightarrow_{p_k} |k\rangle \otimes |\phi_k\rangle \qquad (\mathrm{proy}_{\mathbb{B}_i})$$

$$\pi^m_{\mathbb{B}_i} \lightning \longrightarrow_1 \lightning \qquad (\lightning_{\pi_{\mathbb{B}_i}})$$

The definition of $|k\rangle$ and $|\phi_k\rangle$ is analogous to the π and $\pi_{\mathbb{X}}$ cases.

Such flexibility is crucial for modelling quantum procedures that involve intermediate measurements in different bases—for example, variants of phase estimation or error-correction protocols.

A subtle issue arises when a single quantum state belongs to more than one basis. For instance, consider a basis $\mathbb{B}_1 = \{|0\rangle, -|1\rangle\}$. In this case, the vector $|0\rangle$ may be typed both as $\mathbb{B}$ and $\mathbb{B}_1$, but $\mathbb{B}$ and $\mathbb{B}_1$ are not subtypes of each other. Therefore, it is not safe to allow a term like $|0\rangle$ to be freely assigned both types: one must commit to a single basis when typing such terms.

To avoid ambiguity, in this first extension we assume that no basis $\mathbb{B}_i$ shares any element (or scalar multiple of an element) with the computational basis $\mathbb{B}$ or with any other $\mathbb{B}_j$ for $j \neq i$. This restriction simplifies reasoning and ensures disjointness at the type level.

However, in the next section, we revisit this restriction and propose an alternative typing discipline that allows overlapping bases, by making explicit the relationship between different types for the same state.

All the correctness properties from Section 3 generalise straightforwardly to this extended system, and thus, we omit their proofs.

4.2 Extending Subtyping Across Overlapping Bases

Let us return to our earlier example, where $\mathbb{B}_1 = \{|0\rangle, -|1\rangle\}$. In this case, we would like $|0\rangle$ to be typable both as $\mathbb{B}$ and as $\mathbb{B}_1$. To make this possible, we introduce a new type $\mathbb{Q}_{|0\rangle}$, representing the one-dimensional subspace spanned by the vector $|0\rangle$, and require that $\mathbb{Q}_{|0\rangle} \preceq \mathbb{B}$ and $\mathbb{Q}_{|0\rangle} \preceq \mathbb{B}_1$. Such types $\mathbb{Q}_{|\psi\rangle}$ will not correspond to measurement bases, but rather to generators of linear subspaces.

Recall that the type $S(A)$ is interpreted as the linear span of the set denoted by A. Consequently, $S(\mathbb{Q}_{|0\rangle})$ is a strict subspace of $S(\mathbb{B})$, meaning in particular that $S(\mathbb{B}) \not\preceq S(\mathbb{Q}_{|0\rangle})$. This distinction is crucial: throughout the work we rely on the fact that $S(\mathbb{B}) \approx S(\mathbb{X})$, a property that does not hold in general when introducing $\mathbb{Q}_{|\psi\rangle}$ types. To maintain soundness and consistency, we must refine the subtyping relation accordingly.

Let $\mathbf{L}$ denote the class of linear space generators, indexed up to m. Since there are n alternative bases besides the computational basis $\mathbb{B}$, and each base consists of two orthogonal qubit states, we allow

m to range from 3 to $2n+1$ to accommodate all possible subspace generators:

$$\begin{aligned}
\mathbb{Q} &:= \mathbb{Q}_0 \mid \ldots \mid \mathbb{Q}_m && \text{Generators of linear spaces (L)}\\
\nu &:= \mathbb{B} \mid \mathbb{B}_1 \mid \ldots \mid \mathbb{B}_n && \text{Atomic types (A)}\\
\mathbb{M} &:= \nu \mid \mathbb{M} \times \mathbb{M} && \text{Measurement bases (B)}\\
\mathbb{G} &:= \mathbb{Q} \mid \nu && \text{Single-qubit types (G)}\\
\Psi &:= \mathbb{G} \mid S(\Psi) \mid \Psi \times \Psi && \text{Qubit types (Q)}
\end{aligned}$$

As explained, the subtyping relation must be refined to reflect that $S(\mathbb{Q}_{|\psi\rangle})$ may be a strict subspace of $S(\nu)$. For arbitrary atomic bases ν_1 and ν_2, we still have $S(\nu_1) \approx S(\nu_2)$; however, this is not the case for spaces generated by $\mathbb{Q}_{|\psi\rangle}$ and $\mathbb{Q}_{|\phi\rangle}$ in general. To accommodate these distinctions, we introduce the following subtyping rules:

$$\frac{}{\prod_{i=0}^{a}\left(\left(\prod_{j=0}^{b_i} \nu_{ij}\right)\times\left(\prod_{k=0}^{c_i}\mathbb{Q}_{ik}\right)\right) \preceq S\left(\prod_{i=0}^{a}\left(\left(\prod_{j=0}^{b_i} \nu'_{ij}\right)\times\left(\prod_{k=0}^{c_i}\mathbb{Q}_{ik}\right)\right)\right)}$$

$$\frac{}{\mathbb{Q}_{|0\rangle} \preceq \mathbb{B}} \qquad \frac{}{\mathbb{Q}_{|1\rangle} \preceq \mathbb{B}} \qquad \frac{|\psi\rangle \in \mathbb{B}_i}{\mathbb{Q}_{|\psi\rangle} \preceq \mathbb{B}_i}$$

These rules allow each $\mathbb{Q}_{|\psi\rangle}$ to be used in any basis containing $|\psi\rangle$, preserving semantic distinctions across bases. This enables precise typing of states that appear in multiple contexts and supports richer quantum programs involving basis reuse. This extended system preserves the core properties proved for Lambda-SX in Section 3. The adaptation is straightforward.

Example 4.1 (Basis-sensitive choice). The type $\mathbb{Q}_{|0\rangle}$, introduced in Section 4.2, allows a single quantum state to be treated as belonging to multiple bases. For instance, $|0\rangle$ belongs to both the computational basis $\mathbb{B} = \{|0\rangle, |1\rangle\}$ and the alternative basis $\mathbb{Z} = \{|0\rangle, -|1\rangle\}$, and we have $\mathbb{Q}_{|0\rangle} \preceq \mathbb{B}$ and $\mathbb{Q}_{|0\rangle} \preceq \mathbb{Z}$.

This enables the same state to be interpreted differently in different contexts. Suppose we have: $\mathbf{f} : \mathbb{B} \Rightarrow S(\mathbb{B})$ and $\mathbf{g} : \mathbb{Z} \Rightarrow S(\mathbb{B})$. Then both $\mathbf{choice}_{\mathbb{B}} = \lambda x^{\mathbb{Q}_{|0\rangle}}.\mathbf{f}(x)$ and $\mathbf{choice}_{\mathbb{Z}} = \lambda x^{\mathbb{Q}_{|0\rangle}}.\mathbf{g}(x)$ are well-typed. We can also define a basis-sensitive choice controlled by a qubit: $\mathbf{choice} = \lambda y^{\mathbb{B}}.\lambda x^{\mathbb{Q}_{|0\rangle}}.y?\mathbf{f}(x)\cdot\mathbf{g}(x)$. This term, of type $\mathbb{B} \Rightarrow \mathbb{Q}_{|0\rangle} \Rightarrow S(\mathbb{B})$, illustrates how $\mathbb{Q}_{|\psi\rangle}$ types enable basis-dependent behaviour without unsafe coercions or ad hoc annotations. While the example is simple, it demonstrates how the extended subtyping system supports flexible quantum program structuring, paving the way for optimisations based on contextual basis interpretation.

5 Conclusion and Future Work

We have introduced Lambda-SX, a quantum lambda-calculus that supports control over multiple measurement bases and explicit typing for quantum states shared across them. Through a range of small illustrative examples we have shown that tracking duplicability relative to distinct bases enables concise and compositional encodings of quantum procedures. This expressiveness enables facilitates modular descriptions of basis-sensitive constructs, such as conditional control and Hadamard-based branching, while preserving key meta-theoretical properties like strong normalisation.

While Lambda-SX is presented as a proof-of-concept, it opens avenues for exploring richer type disciplines that more closely mirror quantum semantics. Compared to previous approaches that rely on a single fixed basis, our calculus enables direct reasoning about transformations and measurements involving incompatible bases, without resorting to meta-level annotations. The fine-grained typing system not only enforces safety properties like strong normalisation but also provides a framework for understanding and structuring quantum algorithms in a modular, basis-sensitive way.

As future work, we aim to provide a categorical model for Lambda-SX. The foundational results for Lambda-S and Lambda-S_1 [9–11] already establish a connection between quantum control and adjunctions between Cartesian and monoidal categories. Lambda-SX enriches this picture by integrating multiple measurement bases as first-class citizens in the type system. A natural direction is to explore categorical semantics where each measurement basis corresponds to a distinct comonadic modality, and cast operations are interpreted as morphisms connecting these modalities or embedding subspaces into larger measurement spaces, as formalised by the extended subtyping relation.

This fits into a broader research programme toward a computational quantum logic, as outlined in [4], which aims to provide a Curry-Howard-Lambek-style correspondence for quantum computation. In this programme, Lambda-S represents the computational side of this correspondence, while a linear proof language—such as $\mathcal{L}^{\mathcal{S}}$ [5]—captures the logical side. Preliminary results suggest that quantum computation could be understood as a structural dual to intuitionistic linear logic, with semantic models built on adjunctions between symmetric monoidal categories and their Cartesian or additive counterparts. Extending these models to accommodate the richer structure of Lambda-SX, and even exploring connections to graphical calculi such as ZX, may yield new insights into both the foundations and practical implementation of quantum programming languages.

Lambda-SX opens up a new perspective on how the structure of quantum computation can be captured within a typed lambda-calculus, supporting fine control over duplication, measurement, and basis transition—all of which are essential ingredients in the development of expressive and robust quantum programming formalisms.

Acknowledgments

This work is supported by the European Union through the MSCA SE project QCOMICAL (Grant Agreement ID: 101182520), by the Plan France 2030 through the PEPR integrated project EPiQ (ANR-22-PETQ-0007), and by the Uruguayan CSIC grant 22520220100073UD.

References

[1] Thorsten Altenkirch and Jonathan Grattage. 2005. A Functional Quantum Programming Language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*. 249–258.
[2] Pablo Arrighi and Gilles Dowek. 2017. Lineal: A linear-algebraic Lambda-calculus. *Logical Methods in Computer Science* 13, 1:8 (2017).
[3] Ali Assaf, Alejandro Díaz-Caro, Simon Perdrix, Christine Tasson, and Benoît Valiron. 2014. Call-by-value, call-by-name and the vectorial behaviour of the algebraic lambda-calculus. *Logical Methods in Computer Science* 10, 4:8 (2014).
[4] Alejandro Díaz-Caro. 2025. Towards a Computational Quantum Logic: An Overview of an Ongoing Research Program. Invited talk at CiE 2025: Computability in Europe. To appear at LNCS.
[5] Alejandro Díaz-Caro and Gilles Dowek. 2024. A linear linear lambda-calculus. *Mathematical Structures in Computer Science* 34, 10 (2024), 1103–1137.
[6] Alejandro Díaz-Caro and Nicolas A. Monzon. 2025. A Quantum-Control Lambda-Calculus with Multiple Measurement Bases. To appear at LNCS. Full version at arXiv:2506.16244. In *APLAS 2025: 23rd Asian Symposium on Programming Languages and Systems*.

[7] Alejandro Díaz-Caro, Gilles Dowek, and Juan Pablo Rinaldi. 2019. Two linearities for quantum computing in the lambda calculus. *BioSystems* 186 (2019), 104012.
[8] Alejandro Díaz-Caro, Mauricio Guillermo, Alexandre Miquel, and Benoît Valiron. 2019. Realizability in the Unitary Sphere. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019)*. 1–13.
[9] Alejandro Díaz-Caro and Octavio Malherbe. 2020. A Categorical Construction for the Computational Definition of Vector Spaces. *Applied Categorical Structures* 28, 5 (2020), 807–844.
[10] Alejandro Díaz-Caro and Octavio Malherbe. 2022. Quantum Control in the Unitary Sphere: Lambda-S_1 and its Categorical Model. *Logical Methods in Computer Science* 18, 3:32 (2022).
[11] Alejandro Díaz-Caro and Octavio Malherbe. 2023. A concrete model for a typed linear algebraic lambda calculus. *Mathematical Structures in Computer Science* 34, 1 (2023), 1–44.
[12] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101.
[13] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. *ACM SIGPLAN Notices* 48, 6 (2013), 333–342.
[14] S. C. Kleene. 1945. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic* 10, 4 (1945), 109–124.
[15] Emmanuel Knill. 1996. *Conventions for Quantum Pseudocode.* Technical Report LAUR-96-2724. Los Alamos National Laboratory.
[16] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. *ACM SIGPLAN Notices* 52, 1 (2017), 846–858.
[17] Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (2004), 527–586.
[18] Peter Selinger and Benoît Valiron. 2006. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science* 16, 3 (2006), 527–552.
[19] Morten Heine B. Sørensen and Pawel Urzyczyn. 2006. *Lectures on the Curry-Howard Isomorphism.* Elsevier, Amsterdam; Oxford.
[20] William K. Wootters and Wojciech H. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299 (1982), 802–803.

List Fold Operators in Dafny

Juan Michelini Nora Szasz Álvaro Tasistro

Universidad ORT Uruguay
Montevideo, Uruguay

ABSTRACT

We provide explanations based on simple programming techniques of formulations of the operators `foldr` and `foldl` for finite lists in terms of each other that appear rather convolved at first sight, and use the verification aware programming language Dafny to formally prove their correctness.

ACM Reference Format:
Juan Michelini Nora Szasz Álvaro Tasistro, . 2025. List Fold Operators in Dafny. In *Proceedings of (IFL 2025 - Draft papers).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The functions `foldr` and `foldl` for finite lists are defined as follows (in Haskell [1] notation):

```
foldr:: (a -> b -> b) -> b -> [a] -> b
foldr h b []     = b
foldr h b (x:xs) = h x (foldr h b xs)

foldl:: (b -> a -> b) -> b -> [a] -> b
foldl h b []     = b
foldl h b (x:xs) = foldl h (h b x) xs
```

To begin with, we consider the discussion in the Haskell Wiki concerning mutual definitions of these functions, which appear a bit convolved and unmotivated, at least at first sight[1]:

```
foldl f a bs = foldr (\b g x -> g (f x b)) id bs a

foldr f a bs = foldl (\g b x -> g (f b x)) id bs a
```

As initial goal, we try to identify interesting simple techniques of programming allowing for smooth derivations of these definitions. The techniques involve the use of higher order functions for introducing varying parameters in primitive recursive definitions, of so-called accumulating functions, and of `map` over lists of functions.

[1] See (`https://wiki.haskell.org/Foldl_as_foldr`)

IFL 2025 - Draft papers, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

On the other hand, we use the verification-aware programming language Dafny [2] for checking the correctness of the above mentioned derivations. Dafny combines imperative and functional features together with specifications, annotations and features for writing lemmas that are verified or directly proven in automatic manner. As far as we know, the literature on the use of higher-order functions in this context is rather scarce, and so we expect the present work to constitute a useful experience.

The structure of this paper is as follows: In the next section we explain how to derive the equation for expressing `foldl` in terms of `foldr` by accomodating the use of varying parameters in the latter. In section 3 we derive `foldr` in terms of `foldl` by generalizing the use of accumulating functions. In section 4 we employ lists of functions and fusion to mutually derive the operators in question, and in Section 5 we show and discuss the development in Dafny. Section 6 mentions some conclusions and further work.

2 VARYING PARAMETERS IN FOLD RECURSION

The function `foldr` is an abstraction of the following recursion schema:

```
f:: [a] -> b
f []     = b
f (x:xs) = h x (f xs)
```

This is easy to adapt to the case in which additional parameters are to be considered, unless the parameters may vary in the recursive call, as in the following (1):

```
f:: p -> [a] -> b
f p []     = b p
f p (x:xs) = h p x (f (g x p) xs)
```

where we added an extra parameter `p` to `b` and `h`, and `g` is a function that modifies the parameter in the recursive call:

```
b:: p -> b
h:: p -> a -> b -> b
g:: a -> p -> p
```

However, this schema with parameters can also be expressed as a `foldr`, just by defining a function of type `p -> b` by simple recursion. Indeed, consider

```
f':: [a] -> (p -> b)
f' []     = b
f' (x:xs) = \p-> h p x (f' xs (g x p))
```

Note that the arguments of `f'` have been swapped, i.e. we get that, for all `l` and for all `p`:

```
f p l = f' l p
```

Now, we directly express f' as a foldr:

```
f' = foldr (\x r-> \p-> h p x (r (g x p))) b
```

and so, we get (2):

```
f p l = (foldr (\x r-> \p-> h p x (r (g x p))) b) l p
```

Let us observe foldl:

```
foldl h a []     = a
foldl h a (x:xs) = foldl h (h a x) xs
```

Note that foldl h is a case of recursion with varying parameters. For realizing this, it is useful to rewrite the right-hand sides in order to make it easier to match the schema in (1):

```
foldl h a []     = id a
foldl h a (x:xs) = (\p x r-> r) a x (foldl h (h a x) xs)
```

where id :: a -> a is the identity function defined as \x-> x.

Now, we can write foldl as a foldr by adapting (2) to the present case, and after performing some beta reductions, we obtain:

```
foldl h a l = foldr (\x r-> \p-> (r (h p x))) id l a
```

which is the same as the equation in the Haskell Wiki, up to alpha conversions.

3 ACCUMULATING FUNCTIONS

Now we want to express foldr in terms of foldl. Recall that, informally:

```
foldr h b [x1,x2,..., xn] = h x1 (h x2 (... h xn b)...))
```

We can rewrite this using the composition operator
(.) :: (b -> c) -> (a -> b) -> a -> c, defined as
(f .ġ) x = f (g x),
which is associative and has id as identity value:

```
foldr h b [x1,x2,...,xn] = ((h x1) . (h x2) . ... (h xn)) b
                         = (...((h x1) . (h x2)) ... . (h xn)) b
                = (...((id . (h x1)) . (h x2)) ... . (h xn)) b
```

On the other hand, we can express the result of foldl g d on a list [x1,x2,...,xn] writing the function g as an infix operator ‘g‘ as:

```
foldl g d [x1,x2,...,xn] =
                  ((((d `g` x1) `g` x2) `g` ...  `g` xn)
```

It is now direct to match the two last equations, by using id as initial value and \g x-> g .(h x) as the operator. We get:

```
foldr h b l = foldl (\g x-> g .(h x)) id l b
```

We can unfold the definition of composition by adding an explicit parameter in order to get the Haskell Wiki version, up to alpha conversion:

```
foldr h b l = foldl (\g x z-> g (h x z)) id l b
```

4 USING LISTS OF FUNCTIONS AND FUSION

In this section we make use of the function
map :: (a -> b) -> [a] -> [b]
to achieve yet other derivations.

4.1 foldl in terms of foldr

Recall the informal equation

```
foldl h d [x1,x2,...,xk] = (((d `h` x1) `h` x2)...`h` xk)
```

Using (;) :: (a -> b) -> (b -> c) -> a -> c defined as
(f ; g) x = g (f x), we can rewrite this as:

```
foldl h d [x1,x2,...,xk] =
                  ((`h` x1);(`h` x2);...;(`h` xk)) d
```

where (‘h‘ x) is the partial aplication of h to its second argument, i.e., \z -> h z x or flip h x.

Since id is the identity of (;), we can unfold the (;) introducing foldr:

```
foldl h d [x1,x2,...,xk] =
     foldr (;) id [(`h` x1), (`h` x2), ..., (`h` xk)] d
```

Now, we can write the function list in terms of map, which gives:

```
foldl h d xs = foldr (;) id (map (flip h) xs) d
```

And this can be fused into a single pass:

```
foldl h d xs = foldr (\x r -> (;)  ((flip h) x) r) id xs d
```

After some beta-reductions we get:

```
foldl h d xs = foldr (\x r u -> r (h u x)) id xs d
```

which after renaming gives us the same as the Haskell Wiki.

4.2 foldr in terms of foldl

We start with foldr and expand it:

```
foldr h d [x1,x2, ...,xk] = (x1 `h` (x2 `h`...(xk `h` d)))
```

Then we do the same as above with (.) and we can rewrite it as:

```
foldr h d [x1,x2, ...,xk] =
                  ((h x1) . (h x2) .  ... . (h xk)) d
```

Using that id is the identity (.), we can unfold this to

```
foldr h d [x1,x2, ...,xk] =
        foldl (.) id [(h x1),(h x2),  ... ,(h xk)] d
```

Now we can express the function list as a map:

```
foldr h d xs  = foldl (.) id (map h xs) d
```

If we want to write map in terms of foldl, a simple attempt actually gives us mapRev which is equivalent to mapping and reversing the list:

```
mapRev:: (a -> b) -> [a] -> [b]
mapRev f xs = foldl (\ r x -> (f x):r) [] xs
```

Informally,

```
mapRev f [x1,x2, ...,xk] = [f xk, ... ,f x2, f x1]
```

We can use mapRev by flipping (.) (which is (;)) instead:

```
foldr h d xs  = foldl (;) id (mapRev h xs) d
```

which expanded gives us:

```
foldr h d xs  = foldl (;) id (foldl (\r x -> (h x):r) [] xs) d
```

This can now be fused into:

```
foldr h d xs  = foldl (\ r x -> (;) (h x) r) id xs d
```

And after some reductions we get:

```
foldr h d xs  = foldl (\ r x u -> r ((h x) u)) id xs d
```

which up to alpha renamings gives us the Haskell Wiki equation.

5 DAFNY

In this section we show the main points of our development in Dafny [2]. Dafny is a verification-aware programming language that has native support for specifications and is equipped with a static program verifier. It combines imperative and functional features. We will concentrate on the functional part, where we can define inductive types and perform proofs by induction. Its syntax is very similar to other functional programming languages, so we expect that the reader can grasp the code.

We start by defining lists, and the functions foldr and foldl.

```
datatype List<A> = Nil
                 | Cons (A, List<A>)

function foldr<A, B> (h: (A, B) -> B, b: B, l: List<A>): B
{match l
  case Nil => b
  case Cons (x, xs) => h (x, foldr (h, b, xs))
}
function foldl<A,B> (h: (B, A) -> B, b: B, l: List<A>): B
{match l
  case Nil => b
  case Cons (x, xs) => foldl(h, h(b, x), xs)
}
```

First we give proofs that the Haskell Wiki definitions are correct. A lemma in Dafny is a method to deliver evidence of the proposition in the ensures clause. Hypotheses are declared using requires.

Lemma foldl_as_foldr is proven by induction on the list l using the match construct, which is the same as the one for defining functions by pattern matching. We exhibit the code as necessary for Dafny to compile, which implies the correctness of the lemma. For instance, the Nil case in the proof below is empty because Dafny is able to prove it by itself. In the Cons case, one can appreciate the calculational format of the proofs. In general equalities are annotated with hints to help their verification. The relevant case in this case is the use of the recursive call (induction hypothesis). Notice the notation x => e for lambda abstractions.

```
lemma foldl_as_foldr<A,B> (h:(B, A)->B, b: B, l: List<A>)
ensures foldl (h, b, l) == foldr<A, (B)-> B>
                    ((x, r:(B)-> B) => a => r(h(a, x)),
                                (x=>x),
                                l) (b)
{match l
  case Nil =>
  case Cons (x, xs) =>
       calc{
        foldl (h, b, l) ;
        ==          // foldl def.
        foldl(h, h(b, x), xs) ;
        == {foldl_as_foldr (h, h(b, x), xs);}
        foldr ((x, r:(B) ->B) => a => r(h(a, x)),
               (x=>x),
               xs) (h(b, x)) ;
       }
}
```

The lemma below proves a generalization of the formula in Haskell Wiki, i.e., the latter is obtained by instantiating the parameter g with the identity function. It is proven directly by Dafny's verifier.

```
lemma foldr_as_foldl<A,B> (h: (A, B) -> B, b: B, l: List<A>,
                           g: (B) -> B)
ensures g (foldr(h,b,l)) == foldl<A,(B)->B> ((gg:(B)->B,x)
                            => z =>  gg(h(x,z)),g,l)(b)
{}
```

The next lemma proves the correctness of the formulation of the schema for recursion with varying parameters in terms of foldr presented in Section 2.

```
lemma param_foldr<P, A, B> (f: (P) -> (List<A>) -> B,
                            h: (P) -> (A) -> (B) -> B,
                            b: (P) -> B,
                            g: (A, P) -> P,
                            l: List<A>,
                            p: P)
 requires forall l: List<A>, p: P:: f (p) (l)
                                    == match l
                                      case Nil => b(p)
                                   case Cons (x, xs) =>
                                h (p) (x) (f (g(x, p)) (xs))
 ensures f (p) (l)
      == foldr<A, (P) -> B> ((x: A, r: (P) -> B)
                  => p=> h (p) (x) (r(g(x, p))), b, l) (p)
{match l
  case Nil => {calc{
                f (p) (l);
               ==
                b(p);
              }}
  case Cons (x, xs) =>
              {calc{
                foldr<A,(P)->B> ((x:A,r:(P)->B)
                     =>p=>h(p)(x)(r(g(x, p))),b,l)(p);
```

```
            ==
               h(p)(x)(foldr<A,(P)->B>((x:A,r:(P)->B)
                =>p=> h(p)(x)(r(g(x,p))),b,xs)(g(x,p)));
            == {param_foldr(f, h, b, g, xs, g(x, p));}
                 h (p) (x) (f (g(x, p)) (xs));
              }}
}
```

Next we show the main results of section 4.1, firstly the proof that `foldl` can be expressed as a `foldr` over a list of functions:

```
function l_map<A,B> (f: (A) -> B, l: List<A>): List<B>
{foldr ((x, r)=> Cons(f(x), r), Nil, l)}

function f_seq<A, B, C> (f: (A) -> B, g: (B) -> C): (A) -> C
{x=> g(f(x))}

function flip<A, B, C> (f: (A, B) -> C): (B, A) -> C
{(y, x)=> f(x, y)}

lemma foldl_as_foldr_fun_list<A, B> (h: (B, A) -> B,
                                     b: B, l: List<A>)
ensures foldl(h, b, l) ==
            foldr(f_seq, x=> x,
                  l_map (x=> y=> flip(h)(x, y),l))(b)
{match l
  case Nil =>
  case Cons (x, xs) =>
   {calc{
   foldr(f_seq, x=> x, l_map (x=> y=> flip(h)(x, y), l))(b);
     ==
    foldr(f_seq,x=> x,
     Cons(y=>flip(h)(x,y),l_map(x=>y=>flip(h)(x,y),xs)))
       (b);
     ==
    f_seq(y=>flip(h)(x,y),
         foldr(f_seq,x=> x,l_map(x=>y=>flip(h)(x,y),xs)))
        (b);
     ==
   foldr(f_seq, x=> x, l_map (x=> y=> flip(h)(x, y), xs))
        (flip(h)(x,b));
    == {foldl_as_foldr_fun_list (h, flip(h)(x,b), xs);}
    foldl(h, flip(h)(x,b), xs);
        }}
}
```

Then we prove the fusion of `foldr` and `map`:

```
lemma foldr_map_fusion<A, B, C>
           (g: (A,B) -> B,
            b: B,
            f: (C) -> A,
            l: List<C>)
ensures foldr (g, b, l_map (f, l))
        ==
        foldr ((x, r) => g (f(x), r), b, l)
{}
```

6 CONCLUSIONS AND FURTHER WORK

We have employed multiple simple programming techniques to derive the mutual definitions of `foldr` and `foldl` for finite lists. We believe these explanations serve to at least motivate the definitions in question. We also have employed Dafny for verifying the essential parts of the derivations, which we expect to constitute a contribution to the literature in the application of this language and its system, particularly because of the extensive treatment of higher-order functions. The language and system have proven to be extremely handy for the task at hand, allowing to get concise proofs in short time. We consider this highly promising and are currently trying the tool in applications concerning theory and infrastructure of programming languages, among others.

REFERENCES

[1] G. Hutton (2007): *Programming in Haskell.* Cambridge University Press.
[2] K.R.M. Leino & K. Leino (2023): *Program Proofs.* MIT Press.

Formalizating System I with type Top in Agda

Agustín Séttimo
Dpto. de Ciencias de la Computación.
Universidad Nacional de Rosario.
Rosario, Argentina.
agustinsettimo.ips@gmail.com

Cristian Sottile
Dpto. de Ciencia y Tecnología.
Universidad Nacional de Quilmes.
Bernal, Buenos Aires, Argentina.
Instituto de Investigación en Ciencias de la Computación (ICC). CONICET / Universidad de Buenos Aires.
Buenos Aires, Argentina.
csottile@dc.uba.ar

Cecilia Manzino
Dpto. de Ciencias de la Computación.
Universidad Nacional de Rosario.
Rosario, Argentina.
ceciliam@fceia.unr.edu.ar

Abstract

System I is a recently introduced simply-typed lambda calculus with pairs where isomorphic types are considered equal. In this work we propose an extension of System I with the type Top, and present a complete formalization of this calculus in Agda, which includes the proofs of progress and strong normalization.

Keywords: Lambda calculus, Type isomorphisms, Agda, Strong normalization

ACM Reference Format:
Agustín Séttimo, Cristian Sottile, and Cecilia Manzino. 2025. Formalizating System I with type Top in Agda. In *IFL 2025: Proceedings of the 37th Symposium on Implementation and Application of Functional Languages (IFL '25), October 1–3, 2025, Montevideo, Uruguay.* ACM, New York, NY, USA, 10 pages.

1 Introduction

The study of type isomorphisms is a recent field of research with multiple applications. From the perspective of programming languages, if we identify isomorphic types, we can identify programs of the same type with different syntax but semantically equivalent. This allows a novel form of programming, for example, a function could take its arguments in any order, since the types $A \to B \to C$ and $B \to A \to C$ are isomorphic.

On the other hand, from the perspective of proof systems, since types can be thought of as propositions and programs correspond to proofs, considering different proofs of isomorphic propositions as the same proof means that we have a form of proof-irrelevance.

IFL '25, Montevideo, Uruguay

Systems that consider type isomorphisms as equalities are called systems modulo isomorphism. Most of these systems use simply typed lambda calculus as a framework, and extend it so that isomorphic types are considered identical. The first of these systems, which is also the most relevant for this work, was System I [5]: a simply typed lambda calculus with pairs modulo isomorphisms. An interpreter of a preliminary version of this calculus has been implemented in Haskell [7], and variants with η-expansion rules [6], fewer isomorphisms with no type system [2], and polymorphism [12] have been introduced.

The main contributions of this work are:

- The extension of System I with the type Top. Adding new type constructors to a calculus modulo isomorphisms requires extending the equivalence relation on types, considering the isomorphisms involving the new constructor and also the equivalences between terms of isomorphic types. The choice of these term isomorphisms is also motivated by the goal of having a normalizing calculus.
- The formalization in Agda of this calculus, defining its syntax, semantics, and typing rules. The approach we follow in the representation of typed terms and the notation is similar to that used in [14].
- The proofs in Agda of progress and strong normalization. In our setting, strong normalization means that every reduction sequence starting from a typed term is finite.

The complete Agda code is available in the GitHub repository https://github.com/AgusSett/thesis.

The paper is organized as follows. Section 2 presents an overview of some calculi modulo isomorphisms. Section 3 extends System I with the type Top. Section 4 presents the formalization in Agda of the extended calculus and proves Progress. Section 5 proves the Strong Normalization property. Section 6 concludes the paper.

2 STLC modulo isomorphisms

In this section we introduce System I [5], which is the base system that we later extend and formalize. Two types A and

B are isomorphic (denoted by $\equiv$) iff there exist two functions $f : A \to B$ and $g : B \to A$ such that $f \circ g = id_B$ and $g \circ f = id_A$.

All the type isomorphisms that occur in these systems were characterized and grouped into axiomatic sets by Di Cosmo [4]. For example, the set that corresponds to simply typed lambda calculus is formed by the isomorphism called swap ($A \to (B \to C) \equiv B \to (A \to C)$). For the simply typed lambda calculus extended with cartesian product (denoted as $\times$), all the isomorphisms are consequences of the following four:

$$A \times B \equiv B \wedge A \tag{1}$$
$$A \times (B \times C) \equiv (A \times B) \wedge C \tag{2}$$
$$A \to (B \times C) \equiv (A \to B) \times (A \to C) \tag{3}$$
$$(A \times B) \to C \equiv A \to B \to C \tag{4}$$

We note that the swap isomorphism is not included since it can be constructed from (1) and (3).

The grammar of System I for types is the same as in traditional lambda calculus, while its grammar for terms differs only in projection, which is parametrized by a type rather than a position:

$$\begin{array}{rcl} A & := & \tau \mid A \to A \mid A \times A \\ r & := & x \mid \lambda x^A.r \mid rr \mid \langle r, r\rangle \mid \pi_A(r) \end{array}$$

The type system of this calculus is extended, with respect to typed lambda calculus, with a typing rule that asserts that if a term t has type A and $A \equiv B$, then t also has type B:

$$\frac{\Gamma \vdash r : A \qquad A \equiv B}{\Gamma \vdash r : B}\,(\equiv)$$

This rule with the four isomorphisms that characterize the calculus induces some equivalences between terms. For example, since the types $A \times B$ and $B \times A$ are equivalent, the terms $\langle r, s\rangle$ and $\langle s, r\rangle$ where $r : A$ and $s : B$, must represent the same term. Therefore, it is necessary to extend the reduction relation with isomorphisms at the term level. This relation, denoted as $\rightleftarrows$, is presented in Table 1. The relation $\rightleftarrows^*$ is its reflexive and transitive closure that forms an equivalence relation.

We note that these rules were chosen with the aim of obtaining a strongly normalizing and consistent calculus.

The operational semantics of System I was defined by two relations: the equivalence relation between terms and a reduction relation which is similar to the classical β-reduction. We will show here why β-reduction cannot be used directly with the rules of Table 1.

The usual projection rules access a pair through the position of the elements, but the (comm) isomorphism allows the order of a pair to be changed, thus allowing either of the two elements to be projected:

$$\pi_1\langle r, s\rangle \hookrightarrow r$$
$$\pi_1\langle r, s\rangle \rightleftarrows \pi_1\langle s, r\rangle \hookrightarrow s$$

This poses a problem for type preservation and also introduces non-determinism. The solution is to access the element of a pair through its type, so a new rule was defined: if $\Gamma \vdash r : A$, then $\pi^A\langle r, s\rangle \hookrightarrow r$. This rule resolves the problem of type preservation but maintains the non-determinism in the calculus. However, it is possible to encode a deterministic projection even when both terms are of the same type. Then, non-determinism of System I is considered a feature and not a problem.

Another conflict with classical β-reduction and the equality rules of terms occurs as a consequence of the isomorphism: $A \to B \to C \equiv B \to A \to C$, which can be proved with (1) and (4). This means that we can pass any argument to an abstraction first.

To solve the problem of type preservation in this case, β-reduction is modified and a condition is added that requires the term being applied to have the same type as the argument of the abstraction: if $\Gamma \vdash s : A$, then $(\lambda x^A.r)s \hookrightarrow r[s/x]$.

Finally, the reduction relation $\rightsquigarrow$ is defined as the relation $\hookrightarrow$ modulo $\rightleftarrows$ (i.e. $\rightsquigarrow := \rightleftarrows^* \circ \hookrightarrow \circ \rightleftarrows^*$), and $\rightsquigarrow^*$ its reflexive and transitive closure.

$$\langle r, s\rangle \rightleftarrows \langle s, r\rangle \quad (\text{comm})$$
$$\langle r, \langle s, t\rangle\rangle \rightleftarrows \langle\langle r, s\rangle, t\rangle \quad (\text{asso})$$
$$\lambda x^A.\langle r, s\rangle \rightleftarrows \langle \lambda x^A.r, \lambda x^A.s\rangle \quad (\text{dist}_\lambda)$$
$$\langle r, s\rangle t \rightleftarrows \langle rt, st\rangle \quad (\text{dist}_{\text{app}})$$
$$r\langle s, t\rangle \rightleftarrows rst \quad (\text{curry})$$

$$\frac{r \rightleftarrows s}{\lambda x^A.r \rightleftarrows \lambda x^A.s} \qquad \frac{r \rightleftarrows s}{rt \rightleftarrows st} \qquad \frac{r \rightleftarrows s}{tr \rightleftarrows ts}$$
$$\frac{r \rightleftarrows s}{\langle t, r\rangle \rightleftarrows \langle t, s\rangle} \qquad \frac{r \rightleftarrows s}{\langle r, t\rangle \rightleftarrows \langle s, t\rangle} \qquad \frac{r \rightleftarrows s}{\pi_A(r) \rightleftarrows \pi_A(s)}$$

Table 1. Rules of equivalence between terms in System I

3 Adding the type Top

In this section we extend System I with the type Top. The syntax of types and terms is as follows:

$$\begin{array}{rcl} A & := & \mathsf{T} \mid A \to A \mid A \times A \\ r & := & \star \mid x \mid \lambda x^A.r \mid rr \mid \langle r, r\rangle \mid \pi_A(r) \end{array}$$

The type system has one additional rule with respect to System I that types the term $\star$ with T. The typing rules of this calculus are given in Figure 1.

The next step is to define the equivalence on types induced by type isomorphisms. To do so, we take (1), (2), (3), (4) and add the following three, which are those related to T:

$$A \times \mathsf{T} \equiv A \tag{5}$$
$$A \to \mathsf{T} \equiv \mathsf{T} \tag{6}$$
$$\mathsf{T} \to A \equiv A \tag{7}$$

In logic, the type T corresponds to the proposition True. Isomorphism (5) corresponds to the neutrality of True w.r.t. conjunction, while (6) and (7) correspond to the neutral and absorbing nature of True w.r.t. implication.

$$\frac{}{\Gamma, x : A \vdash x : A}\ (ax) \qquad \frac{A \equiv B \qquad \Gamma \vdash r : A}{\Gamma \vdash r : B}\ (\equiv)$$

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x.r : A \to B}\ (\to_i) \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash rs : B}\ (\to_e)$$

$$\frac{\Gamma \vdash r : A \qquad \Gamma \vdash s : B}{\Gamma \vdash \langle r, s\rangle : A \times B}\ (\times_i) \qquad \frac{\Gamma \vdash r : A \times B}{\Gamma \vdash \pi_A(r) : A}\ (\times_e)$$

$$\frac{}{\Gamma \vdash \star : \mathsf{T}}\ (\mathsf{T}_i)$$

Figure 1. Typing rules

The next step is to define the equivalences on terms that each of the new isomorphisms induces.

The isomorphism (5) induces the term equality: $\langle r, \star\rangle \rightleftarrows r$. The isomorphism (6) induces two term equivalences that correspond to the introduction and elimination of $\to$: if t has type T, then $t \rightleftarrows \lambda x^A.t$ and if r has type $A \to \mathsf{T}$, then $r \rightleftarrows \star$. Finally, the isomorphism (7) induces two equivalences: if r has type $\mathsf{T} \to A$, then $r\star \rightleftarrows r$ and $\lambda x^{\mathsf{T}}.r \rightleftarrows r$.

In Table 2 we present the term isomorphisms that we consider for the formalization of this calculus, where the congruence rules are omitted since they are the same as those in Table 1.

The selection of isomorphisms between terms presented here is the result of several design decisions based mainly on two objectives. First, all necessary equivalences were added so that no term can get stuck, and there are no eliminations in normal forms. As a consequence, every closed term always reduces to a value. This point will become clear when we consider the progress proof below. The second objective is to preserve the normalization property in this calculus and make the proof of this property less complex.

For example, η-expansion and the split rule ($r \rightleftarrows \langle \pi_A r, \pi_B r\rangle$ where $r : A \times B$) are both necessary to avoid some terms getting stuck. But if these rules are included directly, the normalization property is lost since, for example, η-expansion can be applied an infinite number of times. The solution is to define new constructors that embed these rules, such as ASSO-SPLIT.

The operational semantics of this calculus is defined similarly to System I, but with the new rules for the equivalence between terms: the reduction relation $\rightsquigarrow$ is defined as the relation $\hookrightarrow$ modulo $\rightleftarrows$ (i.e. $\rightsquigarrow := \rightleftarrows^* \circ \hookrightarrow \circ \rightleftarrows^*$), and $\rightsquigarrow^*$ its reflexive and transitive closure.

$$\langle r, s\rangle \rightleftarrows \langle s, r\rangle \qquad (\text{COMM})$$
$$\langle r, \langle s, t\rangle\rangle \rightleftarrows \langle\langle r, s\rangle, t\rangle \qquad (\text{ASSO})$$
$$\langle r, s\rangle \rightleftarrows \langle\langle r, \pi_B(s)\rangle, \pi_C(s)\rangle \qquad (\text{ASSO-SPLIT})$$
$$\lambda x^A.\langle r, s\rangle \rightleftarrows \langle \lambda x^A.r, \lambda x^A.s\rangle \qquad (\text{DIST}_\lambda)$$
$$\text{If } \Gamma \vdash r : B \times C,\ \lambda x^A.r \rightleftarrows \langle \lambda x^A.\pi_B(r), \lambda x^A.\pi_C(r)\rangle \qquad (\text{DIST}_\lambda\text{-SPLIT})$$
$$\text{If } \Gamma \vdash r : A \to B, \Gamma \vdash s : A \to C,$$
$$\langle r, s\rangle \rightleftarrows \lambda x^A.\langle r\, x, s\, x\rangle \qquad (\text{DIST}_{\lambda\eta})$$
$$\lambda x^A.\lambda y^B.t \rightleftarrows \lambda z^{A\times B}.t[\pi_A(z)/x, \pi_B(z)/y] \qquad (\text{CURRY})$$
$$\text{If } \Gamma \vdash t : B \to C,\ \lambda x^A.t \rightleftarrows \lambda z^{A\times B}.t[\pi_A(z)/x]\pi_B(z) \qquad (\text{CURRY}_\eta)$$
$$\lambda x^{A\times B}.t \rightleftarrows \lambda y^A.\lambda z^B.t[\langle y, z\rangle/x] \qquad (\text{UNCURRY})$$
$$\text{If } \Gamma \vdash t : \mathsf{T},\ t \rightleftarrows \lambda x^A.t \qquad (\text{ABS}_i)$$
$$\text{If } \Gamma \vdash r : A \to \mathsf{T},\ r \rightleftarrows \star \qquad (\text{ABS}_e)$$
$$\text{If } \Gamma \vdash r : \mathsf{T} \to A,\ r\star \rightleftarrows r \qquad (\text{ID}_{\to i})$$
$$\lambda x^{\mathsf{T}}.r \rightleftarrows r \qquad (\text{ID}_{\to e})$$

Table 2. Rules of equivalence between terms in System I with Top

4 Formalization

In this section we present a formalization in Agda of a simply typed lambda calculus with pairs and Top extended with type isomorphisms.

Much of the code presented in this work is based on [14]. The adaptation consists mainly of the addition of type isomorphisms and the semantics of the calculus, which are now defined by β-reduction modulo $\rightleftarrows$.

There are two approaches to formalizing a typed λ-calculus: using extrinsically typed terms, where terms and types are defined independently (so a term can be typed or not); or using intrinsically typed terms, where types are defined first and terms are formed with a given type. There are also two ways of representing variable names: using named variables, which are easier to read, or de Bruijn indices, which make the formalization more compact. In this work we decided to use an intrinsic formulation and de Bruijn indices.

4.1 Types, contexts and variables

The types consist of the type Top, function types, and pairs:

```
data Type : Set where
  ⊤    : Type
```

```
_⇒_ : Type → Type → Type
_×_ : Type → Type → Type
```

Since we use natural numbers to represent variables, type contexts are formalized as lists of types. Unlike classical lists, contexts are read from right to left.

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

Intrinsically typed variables are represented by de Bruijn indices. Each variable is indexed by its type and a context in which the variable is typed.

```
data _∋_ : Context → Type → Set where
  Z : ∀ {Γ A}→ Γ , A ∋ A

  S_ : ∀ {Γ A B} → Γ ∋ B → Γ , A ∋ B
```

Then, $\Gamma \ni A$ is the type of variables that have type A in the context Γ.

For example, the following variables with types ⊤ and ⊤ ⇒ ⊤ are typable in the context ∅, ⊤⇒ ⊤, ⊤and also represent proofs of this.

```
_ : ∅ , ⊤ ⇒ ⊤ , ⊤ ∋ ⊤
_ = Z

_ : ∅ , ⊤ ⇒ ⊤ , ⊤ ∋ ⊤ ⇒ ⊤
_ = S Z
```

As we can see in the examples, the proof that a variable has a type in a context is a de Bruijn index.

4.2 Terms

Here we present the typing rules of the calculus. Each constructor of this data type, which encodes a typing rule since we use an intrinsically typed representation of terms, represents a term of the calculus, with the exception of the constructor [_]≡_.

Taking into account this exception, $\Gamma \vdash A$ is the type of terms that have type A in the context Γ, and each term of this type is a proof of that, so terms are actually typing derivations. The constructor [_]≡_ that does not represent a term is used in the typing derivation of some terms.

```
data _⊢_ : Context → Type → Set where

  ‘_ : ∀ {Γ A}       - (ax)
    → Γ ∋ A
    → Γ ⊢ A

  ★ : ∀ {Γ} → Γ ⊢ ⊤ - (⊤ᵢ)

  [_]≡_ : ∀ {Γ A B} - (≡)
    → A ≡ B
    → Γ ⊢ A
    → Γ ⊢ B

  ƛ_ : ∀ {Γ A B}     - (⇒ᵢ)
    → Γ , A ⊢ B
    → Γ ⊢ A ⇒ B

  _·_ : ∀ {Γ A B}    - (⇒ₑ)
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
    → Γ ⊢ B

  ⟨_,_⟩ : ∀ {Γ A B} - (×ᵢ)
    → Γ ⊢ A
    → Γ ⊢ B
    → Γ ⊢ A × B

  π : ∀ {Γ A B}      - (×ₑ)
    → (C : Type)
    → {proof : (C ≅ A) ⊎ (C ≅ B)}
    → Γ ⊢ A × B
    → Γ ⊢ C
```

We note that the constructor π takes as an argument the type C, which is the type that this function uses to carry out the projection, and an implicit argument that serves as proof that C is either equal to type A or type B. In the type of this argument we use propositional equality denoted as ≅ in this work.

The following are some examples of terms. Each of them is a proof that it types in the given context.

```
T₁ : ∅ , ⊤ ⊢ ⊤ ⇒ ⊤
T₁ = ƛ ‘ Z

T₂ : ∅ , ⊤ ⊢ (⊤ ⇒ ⊤) × ⊤
T₂ = ⟨ T₁ , ★ ⟩

T₃ : ∅ , ⊤ ⊢ ⊤
T₃ = (π fun {inj₁ refl} T₂) · ‘ Z
```

4.3 Substitutions and reduction

Using de Bruijn representation, substitutions are simply mappings of natural numbers to terms, so they can be interpreted as infinite sequences of terms. These sequences can be constructed using some operators [1]:

- id: the identity substitution: $\{i \mapsto i\}$
- $\uparrow$: the shift operator: $\{i \mapsto i+1\}$
- $a \bullet s$: the concatenation of the term a with the substitution s: $\{0 \mapsto a, i+1 \mapsto s(i)\}$
- $\circ$: the composition of substitutions

With these operations we can give an inductive definition of the application of a substitution s on a term t, denoted as $\langle\langle s \rangle\rangle t$. Then, β-reduction can be defined as follows:

$$(\lambda t) r \hookrightarrow_\beta \langle\langle r \bullet id \rangle\rangle t$$

To implement substitutions, we use an approach given by Altenkirch and Reus [3] that was formalized by McBride

in [10]. Substitutions are implemented using renamings, which are functions from variables in one context to variables in another that preserve typing. The function that applies a renaming of variables in a term is named `rename` in this implementation:

```
rename : ∀ {Γ Δ} → (∀{ A} → Γ ∋ A →Δ ∋ A) →
(∀ {A} → Γ ⊢ A → Δ ⊢ A)
```

While substitutions and simple substitutions are implemented with the following functions:

```
⟪_⟫ : ∀ {Γ Δ} → (∀{ A} → Γ ∋ A →Δ ⊢ A) →
(∀ {C} → Γ ⊢ C → Δ ⊢ C)
_[_] : ∀ {Γ A B} → Γ , B ⊢ A → Γ ⊢ B → Γ ⊢ A.
```

Since the implementation of these functions is fairly standard, we omit their definitions here.

Then, we present the reduction relation which is defined as the following data type:

```
data _↪_ : (Γ ⊢ A) → (Γ ⊢ A) → Set where
  β-ƛ : ∀ {t : Γ , A ⊢ B} {s : Γ ⊢ A}
  → (ƛ t) · s ↪ t [ s ]

  β-π₁ : ∀ {r : Γ ⊢ A} {s : Γ ⊢ B}
  → π A {inj₁ refl} ⟨ r , s ⟩ ↪ r

  β-π₂ : ∀ {r : Γ ⊢ A} {s : Γ ⊢ B}
  → π B {inj₂ refl} ⟨ r , s ⟩ ↪ s
```

In this definition we omit the constructors $\xi\text{-}\cdot_1$, $\xi\text{-}\cdot_2$, $\xi\text{-}\langle,\rangle_1$, $\xi\text{-}\langle,\rangle_2$, $\xi\text{-}\pi$, $\xi\text{-}\equiv$ and ζ that represent the congruence rules. The constructor $\beta\text{-}\lambda$ corresponds to beta reduction, and the constructors $\beta\text{-}\pi 1$ and $\beta\text{-}\pi 2$ correspond to the applications of the projections.

Since the relation `_↪_` is indexed by two terms of the same type, it is not necessary to prove that it preserves types. This is one of the advantages of using intrinsically typed terms.

4.4 Type isomorphisms

The type isomorphisms included in this formalization correspond to the axiomatic set given in Section 2, the isomorphism that represents the symmetry of ≡, and some isomorphisms that represent congruence rules.

```
data _≡_ : Type → Type → Set where
  comm  : ∀ {A B} → A × B ≡ B × A
  asso  : ∀ {A B C} → A × (B × C) ≡ (A × B) × C
  dist  : ∀ {A B C} → (A ⇒ B) × (A ⇒ C) ≡ A ⇒ B × C
  curry : ∀ {A B C} → A ⇒ B ⇒ C ≡ (A × B) ⇒ C
  id-×  : ∀ {A} → A × ⊤ ≡ A
  id-⇒  : ∀ {A} → ⊤ ⇒ A ≡ A
  abs   : ∀ {A} → A ⇒ ⊤ ≡ ⊤

  sym      : ∀ {A B} → A ≡ B → B ≡ A
  cong⇒₁ : ∀ {A B C} → A ≡ B → A ⇒ C ≡ B ⇒ C
  cong⇒₂ : ∀ {A B C} → A ≡ B → C ⇒ A ≡ C ⇒ B
  cong×₁  : ∀ {A B C} → A ≡ B → A × C ≡ B × C
  cong×₂  : ∀ {A B C} → A ≡ B → C × A ≡ C × B
```

We do not include in the formalization the isomorphism corresponding to the transitivity of ≡, since it can be obtained using the constructor `[_]≡_` that corresponds to the typing rule (≡), and the reflexivity of ≡, since it does not add expressiveness to the formalization.

4.5 Equivalence of terms

The formalization of the equivalence relation between terms, corresponding to isomorphic types (⇄) is presented below:

```
data _⇄_ : (Γ ⊢ A) → (Γ ⊢ A) → Set where
  comm : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ B}
    → [ comm ]≡ ⟨ r , s ⟩ ⇄ ⟨ s , r ⟩

  asso : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ B} → {t : Γ ⊢ C}
    → [ asso ]≡ ⟨ r , ⟨ s , t ⟩ ⟩ ⇄ ⟨ ⟨ r , s ⟩ , t ⟩

  asso-split : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ B × C}
    → [ asso ]≡ ⟨ r , s ⟩ ⇄
    ⟨ ⟨ r , π B {inj₁ refl} s ⟩ , π C {inj₂ refl} s ⟩

  dist-ƛ : ∀ {r : Γ , C ⊢ A} → {s : Γ , C ⊢ B}
    → [ dist ]≡ ⟨ ƛ r , ƛ s ⟩ ⇄ ƛ ⟨ r , s ⟩

  dist-ƛη_l : ∀ {r : Γ ⊢ C ⇒ A} → {s : Γ ⊢ C ⇒ B}
    → [ dist ]≡ ⟨ r , s ⟩ ⇄
  ƛ ⟨ rename S_ r · ‘ Z , rename S_ s · ‘ Z ⟩

  dist-ƛη_l : ∀ {r : Γ ⊢ C ⇒ A} → {s : Γ , C ⊢ B}
    → [ dist ]≡ ⟨ r , ƛ s ⟩ ⇄ ƛ ⟨ rename S_ r · ‘ Z , s ⟩

  dist-ƛη_r : ∀ {r : Γ , C ⊢ A} → {s : Γ ⊢ C ⇒ B}
    → [ dist ]≡ ⟨ ƛ r , s ⟩ ⇄ ƛ ⟨ r , rename S_ s · ‘ Z ⟩

  curry : ∀ {r : Γ , A , B ⊢ C}
    → [ curry ]≡ (ƛ ƛ r) ⇄ ƛ subst σ-curry r
  curry-η : ∀ {r : Γ , A ⊢ B ⇒ C}
    → [ curry ]≡ (ƛ r) ⇄
    ƛ subst σ-curry (rename S_ r · ‘ Z)
  uncurry : ∀ {r : Γ , A × B ⊢ C}
    → [ sym curry ]≡ (ƛ r) ⇄ ƛ ƛ subst σ-uncurry r

  id-× : ∀ {r : Γ ⊢ A} → {t : Γ ⊢ ⊤}
    → [ id-× ]≡ ⟨ r , t ⟩ ⇄ r

  id-⇒ : ∀ {r : Γ ⊢ ⊤ ⇒ A} → [ id-⇒ ]≡ r ⇄ r · ★

  abs : ∀ {r : Γ ⊢ A ⇒ ⊤} → [ abs ]≡ r ⇄ ★
```

where the functions σ-curry and σ-uncurry are substitutions of types Subst (Γ , A , B) (Γ , A × B) and Subst (Γ , A × B) (Γ , A , B) respectively.

In this definition we omit for simplicity the constructors that corresponds to congruence rules, given in Table 2. We also omit some constructors with the prefix sym, all of which could be obtained from the base equivalences combined with the sym constructor. However, they were included in the formalization in order to demonstrate strong normalization.

We note that the equivalence relation between terms ⇄ and the reduction relation ↪ can only relate terms of the same type. The type preservation of these relations is a consequence of the intrinsic representation of types.

Then, we can define the reduction relation ⇝ in Agda, that is the relation ↪ modulo ⇄ , as follows:

```
_⇝_ : ∀ {Γ A} → (t t' : Γ ⊢ A) → Set
t ⇝ t' = t ↪ t' ⊎ t ⇄ t'
```

Furthermore, based on the above, we can assert that this relationship satisfies type preservation.

4.6 Progress

In this section we prove progress for the calculus, which establishes that any typed and closed term is a value or reduces to some other term.

The first step is to define the notion of value, which in this calculus are: the term ⊤, abstractions, and pairs of values.

```
data Value : ∀ {Γ A} → Γ ⊢ A → Set where
  V-ƛ : ∀ {Γ A B} {t : Γ , A ⊢ B} → Value (ƛ t)
  V-★ : ∀ {Γ} → Value (★ {Γ})

  V-⟨_,_⟩ : ∀ {Γ A B} {r : Γ ⊢ A} {s : Γ ⊢ B}
    → Value r  → Value s  → Value ⟨ r , s ⟩
```

In the definition of progress we present below, we use the call-by-value reduction order as a reduction strategy. When reducing under abstractions, it is possible to find free variables. So the final states of reduction to be considered are normal forms instead of values. Then we prove a version of progress for open terms. To complete the proof of progress for closed terms, we prove that every closed term in normal form is a value.

We start by defining the syntax that characterizes terms in normal form in this language:

$$\textbf{norm} := \langle \textbf{norm}, \textbf{norm} \rangle \mid \lambda x.\textbf{norm} \mid \star \mid \textbf{neu}$$
$$\textbf{neu} := \mathit{var} \mid \textbf{neu} \cdot \textbf{norm} \mid \pi\ \textbf{neu} \mid [\ \mathit{iso}\ ]{\equiv}\ \textbf{neu}$$

where *var* are variables and the syntactic category **neu** characterizes neutral forms, which are terms that cannot be reduced and are not values. In the formalization, the symbols ⇓ are used for neutral forms and ⇑ for normal forms.

```
data ⇓ : ∀ {Γ A} → Γ ⊢ A → Set
data ⇑ : ∀ {Γ A} → Γ ⊢ A → Set
```

```
data ⇓ where
  `_ : ∀ {Γ A} (x : Γ ∋ A)→ ⇓ (` x)

  _·_ : ∀ {Γ A B} {r : Γ ⊢ A ⇒ B} {s : Γ ⊢ A}
    → ⇓ r → ⇑ s → ⇓ (r · s)

  π : ∀ {Γ A B C p} {t : Γ ⊢ A × B}
    → ⇓ t → ⇓ (π C {p} t)

  [_]≡_ : ∀ {Γ A B} {t : Γ ⊢ A}
    → (iso : A ≡ B) → ⇓ t → ⇓ ([ iso ]≡ t)

data ⇑ where
  ^_ : ∀ {Γ A} {t : Γ ⊢ A} → ⇓ t → ⇑ t

  N-ƛ : ∀ {Γ A B} {t : Γ , A ⊢ B} → ⇑ (ƛ t)

  N-⟨_,_⟩ : ∀ {Γ A B} {r : Γ ⊢ A} {s : Γ ⊢ B}
    → ⇑ r → ⇑ s → ⇑ ⟨ r , s ⟩

  N-★ : ∀ {Γ} → ⇑ (★ {Γ})
```

Now we define a relation that captures the cases in which a term t satisfies progress, namely: it can be transformed into an equivalent one, it can take a reduction step, or it is in normal form:

```
data Progress {Γ A} (t : Γ ⊢ A) : Set where
  step⇄ : ∀ {t' : Γ ⊢ A} → t ⇄ t' → Progress t

  step↪ : ∀ {t' : Γ ⊢ A} → t ↪ t' → Progress t

  done : ⇑ t → Progress t
```

We can now prove progress. The variable and unit cases are very simple since both terms are in normal form:

```
progress : ∀ {Γ A} → (t : Γ ⊢ A) → Progress t
progress (` x) = done (^ (` x))
progress ★     = done N-★
```

If the term is a lambda abstraction, then we call the progress function under the binder; if this returns a reduction step, the step is extended using ζ (the congruence rule for abstractions). Otherwise, the abstraction is in normal form:

```
progress (ƛ t) with progress t
... | step⇄ t⇄t'  = step⇄ (ζ t⇄t')
... | step↪ t↪t' = step↪ (ζ t↪t')
... | done ⇑t        = done N-ƛ
```

For pairs the proof is similar. If the term is a projection or an application, we recursively apply progress to each subterm; if the result is a reduction or an equivalence case, we use the congruence rule ζ, and if it is the done case, then we have a normal form or we can apply β-reduction.

The most interesting case for this work is when the term is [iso]≡ t. In this case we recursively apply progress to the subterm t; if the result is a reduction or an equivalence case, we use the congruence rule ξ-≡, and if it is the done case, we continue the proof by case analysis on each normal form and all applicable isomorphisms that can be applied in each case. We show here just a few cases; the remaining cases are similar.

```
progress ([ iso ]≡ t) with progress t
... | step⇄ t⇄t' = step⇄ (ξ-≡ t⇄t')
... | step↪ t↪t' = step↪ (ξ-≡ t↪t')

progress ([ comm ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩ =
step⇄ comm

progress ([ asso ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩ =
step⇄ (asso-split)

progress ([ dist ]≡ _) | done N-⟨ N-ƛ , N-ƛ ⟩ =
step⇄ dist-ƛ
progress ([ dist ]≡ _) | done N-⟨ N-ƛ , ⇑s ⟩  =
step⇄ (dist-ƛη_r)
progress ([ dist ]≡ _) | done N-⟨ ⇑r , N-ƛ ⟩  =
step⇄ (dist-ƛη_l)
progress ([ dist ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩   =
step⇄ (dist-ƛη_lr)

progress ([ curry ]≡ (ƛ ƛ _)) | done N-ƛ =
step⇄ curry
progress ([ curry ]≡ _)       | done N-ƛ =
step⇄ (curry-η)
progress ([ id-× ]≡ _)  | done N-⟨ ⇑r , ⇑s ⟩ =
step⇄ id-×
progress ([ id-⇒ ]≡ _) | done ⇑t           =
step⇄ id-⇒
progress ([ abs ]≡ _) | done ⇑t = step⇄ abs
progress ([ iso ]≡ _) | done (^ ⇓t) = done (^ [ iso ]≡ ⇓t)
```

The definition of progress shows how each term isomorphism corresponds to a case where it is necessary to eliminate a rule (≡) in order to continue with the reduction.

Finally, we complete the proof of progress by proving that any closed term in normal form is a value; we show just the type of this function here:

```
closed⇑→Value : ∀ {A}{t : ∅ ⊢ A} → ⇑ t → Value t
```

The cases where the term is the top value, an abstraction, or a pair, the definition of this function is very simple; when the term is a neutral form, we use the function ⊥-elim with a proof that a closed term cannot be neutral. This last proof can be constructed since a neutral term must contain a free variable, because variables are the only constructor that is not recursive in its definition.

5 Strong Normalization

In this section we prove the strong normalization of the relation ⇝, which states that every typed term cannot reduce indefinitely, meaning that every reduction sequence ends in a value.

The most commonly used technique for proving strong normalization is that of reducibility, introduced by Tait [13] and Girard [8, 9]. In this work we follow Schäfer's approach [11], who introduces several changes to reducibility, achieving a technique that is more suitable for formalization, which he uses to prove strong normalization of System F in Coq. Our formalization is based on the formalization given by András Kovács[1] in Agda for simply typed lambda calculus.

We begin by giving a constructive definition of strongly normalizing terms, originally presented by Altenkirch.

The set of strongly normalizing terms with respect to a reduction ⇝ is inductively defined by the following rule:

$$\frac{\forall t.s \rightsquigarrow t \implies t \in SN}{s \in SN}$$

This means that if for each term t such that $s \rightsquigarrow t$, it holds that t is strongly normalizing, then s is also strongly normalizing.

In Agda, we can define the set SN using the following inductive data type:

```
data SN {Γ A} (t : Γ ⊢ A) : Set where
  sn : (∀ {t'} → t ⇝ t' → SN t') → SN t
```

Our goal is to define a function with this type:

```
strong-norm : ∀ {Γ A} (t : Γ ⊢ A) → SN t
```

We give a definition of this function that represents the proof of normalization by generalizing the type SN with the addition of a predicate about the term. In the proof, we use this predicate to add information about the term when constructing introductions, and then we use that information in the cases of eliminations.

```
data SN* {Γ A} (P : Γ ⊢ A → Set) (t : Γ ⊢ A) : Set where
  sn* : P t → (∀ {t'} → t ⇝ t' → SN* P t') → SN* P t
```

It is easy to see that if a term satisfies SN*, then it also satisfies SN.

To begin the proof, we need to provide some definitions. First, we define the "interpretation of the term", which will be used for adding some extra hypotheses that allow us to unblock the proof in the cases of eliminations. A pair is interpreted as the product of the normalization of each element of the pair, an abstraction (λ t) is interpreted as the normalization of the term ρ(t)[u], where u is any normalizing term and ρ is any rename function, and the other terms are interpreted as the unit type of the module Data.Unit.

[1] https://github.com/AndrasKovacs/misc-stuff/blob/master/agda/STLCStrongNorm/StrongNo

```
[[_]] : ∀ {Γ A} → Γ ⊢ A → Set
[[ (ƛ t) ]]     = ∀ {Δ}{ρ : Rename _ Δ}{u} →
SN* [[_]] u → SN* [[_]] (⟪ u • (ids ∘ ρ) ⟫ t)
[[ ⟨ a , b ⟩ ]] = SN* [[_]] a ⊗ SN* [[_]] b
[[ t ]]         = Top
```

where ⊗ is the renaming of the operator × of the module Data.Product, and Top the renaming of ⊤of Data.Unit. The definition of interpretation of terms can be extended to substitutions by defining a predicate that states that a substitution σ is adequate in a context Γ, written as $\Gamma \vDash \sigma$, when all terms that result from the application of σ to any variable are strongly normalizing.

```
_⊨_ : ∀{Δ} → (Γ : Context) → (σ : Subst Γ Δ) → Set
Γ ⊨ σ = ∀{A} (v : Γ ∋ A) → SN* [[_]] (σ {A} v)
```

In particular, the identity substitution ids is an adequate substitution:

```
⊨ids : ∀{Γ} → Γ ⊨ ids
⊨ids _ = SN*-rename S_ (σ v)
```

The complexity of the normalization proof lies in the proof of a fundamental theorem named adequacy, which states that for any term t and adequate substitution σ, SN* [[_]] (⟪ σ ⟫ t) holds. The proof of this theorem is extensive and requires proving some extra lemmas. In the next subsection we present some of the most relevant cases of the proof; here we just give the type of the theorem:

```
adequacy : ∀ {Γ Δ A} {σ : Subst Γ Δ} → (t : Γ ⊢ A) →
Γ ⊨ σ → SN* [[_]] (⟪ σ ⟫ t)
```

Finally, the strong normalization property is proved by instantiating adequacy with the identity substitution:

```
strong-norm : ∀ {Γ A} (t : Γ ⊢ A) → SN t
strong-norm t = transport SN sub-id (SN*-SN
(adequacy t ⊨ids))
```

where the lemma sub-id : ∀{Γ A} {t : Γ ⊢ A} → [ids := t]() ≡ t eliminates the application of the identity substitution to the term, and the function transport is a renaming of subst, defined in the module Relation.Binary.PropositionalEquality.

5.1 The adequacy function

We prove the adequacy theorem by induction on the term t. In each case of the proof we use auxiliary lemmas. To simplify the explanation, we present the proof by cases.

Case Variable

When the term t is a variable, we have to prove that the substitution σ applied to the variable satisfies SN* [[_]] . Since we have as a hypothesis that σ is an adequate substitution, it suffices to apply σ to the variable:

```
adequacy (` v) ⊨σ = ⊨σ v
```

Case Top

This is the simplest case, since there is no possible reduction step from ★:

```
adequacy ★ _ = sn* tt (λ ())
```

Case Pair

When the term t is a pair, the following lemma is defined to allow us to conclude SN* [[_]] t, if SN* [[_]] holds for the subterms of the pair t.

```
lemma-⟨,⟩ : ∀ {Γ A B} → {a : Γ ⊢ A} {b : Γ ⊢ B} →
  SN* [[_]] a → SN* [[_]] b → SN* [[_]] (⟨ a , b ⟩)
```

The proof of this lemma is solved by case analysis on the reduction step taken by SN* [[_]] t, which can be a left or right congruence of the relations ↪ and ⇄ ; in each case we use the inductive hypotheses on the subterms.
Then, we complete the proof of adecuacy for this case as follows:

```
adequacy ⟨ a , b ⟩ ⊨σ =
  lemma-⟨,⟩ (adequacy a ⊨σ) (adequacy b ⊨σ)
```

Case Application

A similar lemma is needed for the case in which t is an application:

```
lemma-· : ∀ {Γ A B} → {a : Γ ⊢ A ⇒ B} {b : Γ ⊢ A} →
  SN* [[_]] a → SN* [[_]] b → SN* [[_]] (a · b)
```

In this lemma we proceed similarly as in the previous lemma, with the exception that now the step can also be a β-reduction. This case is solved by applying the interpretation of the left term of the application (which must be an abstraction) to the inductive hypothesis of the right term.
The adequacy lemma for this case is solved as:

```
adequacy (a · b) ⊨σ =
  lemma-· (adequacy a ⊨σ) (adequacy b ⊨σ)
```

Case Projection

The case where t is a projection requires the following lemma, which asserts that SN* [[_]] t holds if SN* [[_]] holds for the term being projected:

```
lemma-π : ∀ {Γ A B C p} → {a : Γ ⊢ A × B} →
  SN* [[_]] a → SN* [[_]] (π C {p} a)
```

The proof of this lemma is similar to the proof of lemma-· in the sense that when the step is a reduction (β-π_1 or β-π_2), the lemma is solved using the interpretation of the subterm, which in this case is the interpretation of a pair ⟨ a, b ⟩ that has the form SN* [[_]] a ⊗ SN* [[_]] b.
The adequacy lemma for this case is solved as:

```
adequacy (π _ x) ⊨σ = lemma-π (adequacy x ⊨σ)
```

Case Abstraction

The case where t is an abstraction is more difficult than the previous ones. As in these cases, we need to prove SN* [[_]] (ƛ t) from SN* [[_]] t, given the interpretation of (ƛ t):

```
lemma-ƛ : ∀ {Γ A B} → {t : Γ , B ⊢ A} →
  [[ ƛ t ]] → SN* [[_]] t → SN* [[_]] (ƛ t)
```

This proof required several lemmas, for example it was necessary to prove that for any terms u and v, and substitution σ:

1. if u ↪ v then, ⟪ σ ⟫ u ↪ ⟪ σ ⟫ v
2. if u ⇄ v then, ⟪ σ ⟫ u ⇄ ⟪ σ ⟫ v
3. if u is normalizing and σ is adequate, then u • σ is adequate.
4. if u ↪ v then SN* [[_]] u ↪ SN* [[_]] v
5. if u ⇄ v then SN* [[_]] u ⇄ SN* [[_]] v

In addition, we need to prove that an adequate substitution composed with a rename is also an adequate substitution:

```
⊨rename : ∀{Γ Δ Δ₁} {σ : Subst Γ Δ} →
  Γ ⊨ σ → (ρ : Rename Δ Δ₁) → Γ ⊨ (⟪ ids ∘ ρ ⟫ ∘ σ)
```

and that an extension of an adequate substitution is also adequate:

```
⊨exts : ∀{Γ Δ A} {σ : Subst Γ Δ} →
  Γ ⊨ σ → (Γ , A) ⊨ (exts σ)
```

Now, it is possible to define the case of adequacy for abstraction as follows:

```
adequacy {σ = σ} (ƛ t) ⊨σ =
  lemma-ƛ
    (λ { {ρ = ρ}{u = u} SNu →
      transport (SN* [[_]])
        (subst-split {t = t})
        (adequacy t (⊨ SNu • (⊨rename ⊨σ ρ)))})
    (adequacy t (⊨exts ⊨σ))
```

The function subst-split is used to combine the substitutions σ and u • (ids ∘ ρ) into a single substitution.

Case Iso

Finally, the case where t is the constructor of isomorphism is the most difficult case, because we must resolve the cases for each equivalence of terms. As in the other cases, we have a principal lemma:

```
lemma-≡ : ∀ {Γ A B iso} → {t : Γ ⊢ A} →
  SN* [[_]] t → SN* {A = B} [[_]] ([ iso ]≡ t)
```

This function is defined in terms of this auxiliary lemma:

```
aux : ∀ {Γ A iso t'} → {t : Γ ⊢ A} →
  SN* [[_]] t → ([ iso ]≡ t) ⇝ t' → SN* [[_]] t'
```

which is defined by case analysis on the second argument (the reduction step). The goal of this lemma is to derive SN* [[_]] t' from SN* [[_]] t, where t and t' are the terms related by the corresponding isomorphism. In this proof we use the lemmas defined above for the different term constructors.

We will not present the proof of this theorem here due to its length and complexity, we just mention some of the critical points of the proof and the techniques used:

- The case in which the isomorphism is $\langle \lambda x^A.r, \lambda x^A.s \rangle \rightleftarrows \lambda x^A.\langle r, s \rangle$, presents the difficulty of instantiating the interpretations of the left abstractions for constructing the interpretation of the right abstraction. To resolve this, we use a lemma that concludes SN* [[_]] t from SN* [[_]] (ƛ t). The idea behind the lemma is to replace the first variable of the term with index zero, thus obtaining exactly the same term.
- In cases involving the curry isomorphism, multiple substitutions must be combined into a single one that can be obtained through the interpretation of the abstraction.

As a conclusion, we note that the interpretations of the terms allow us to resolve the cases of elimination (projection and application), but, on the other hand, the cases of introduction (product and abstraction) become more complex, since it is in these cases that such interpretations are constructed.

6 Evaluation

Once the strong normalization property has been proven, the evaluation function can be defined in Agda since we have proof that it terminates.

First, we define the relation _⇝*_ as the transitive and reflexive closure of _⇝_:

```
data _⇝*_ {Γ A} : (Γ ⊢ A) → (Γ ⊢ A) → Set where
  _∎ : (t : Γ ⊢ A) → t ⇝* t
  _⇄⟨_⟩_ : (t : Γ ⊢ A) {r t' : Γ ⊢ A} →
    t ⇄ r → r ⇝* t' → t ⇝* t'
  _↪⟨_⟩_ : (t : Γ ⊢ A) {r t' : Γ ⊢ A} →
    t ↪ r → r ⇝* t' → t ⇝* t'
```

Given a closed typed term t, the return type of the evaluation function will be the reduction sequence from t to a value t', defined as the following data type:

```
data Steps {A} : ∅ ⊢ A → Set where
  steps : {t t' : ∅ ⊢ A} →
    t ⇝* t' → Value t' → Steps t
```

Then, the evaluation function is defined in terms of this function:

```
eval´ : ∀ {A} → (t : ∅ ⊢ A) → SN t → Steps t
eval´ t _ with progress t
eval´ t _ | done ⇑t = steps (t ∎) (closed⇑→Value ⇑t)

eval´ t (sn f)
  | step⇄ {r} t⇄r with eval´ r (f (inj₂ t⇄r))
```

```
...| steps r⇝t' fin=steps (t ⇄⟨ t⇄r ⟩ r⇝t') fin
eval´ t (sn f)
   | step↪ {r} t↪r with eval´ r (f (inj₁ t↪r))

...| steps r⇝t' fin=steps (t ↪⟨ t↪r ⟩ r⇝t') fin
```

This function apply progress to the typed term. There are three possibilities:

- If the term `t` is in normal form, the reduction sequence is trivial: `t ⇝* t` and `t` is also a value.
- If `t ⇄ r`, we recursively call `eval` with the proof that `r` is strong normalizing. The result is the reduction sequence obtained by adding the step `t ⇄ r` to the result of the recursion.
- If `t ↪ r`, we proceed similarly.

We note that the argument that provide the evidence that `t` is strong normalizing is necessary to pass Agda's termination checker. In each recursive step, a constructor `sn` is removed from this argument, making it structurally smaller.

Finally, we define `eval` using the strong normalization theorem:

```
eval : ∀ {A} → (t : ∅ ⊢ A) → Steps t
eval t = eval´ t (strong-norm t)
```

7 Conclusions

In this work, we formalized System I with the addition of the type Top in Agda, proving the strong normalization and progress theorems.

The strong normalization theorem is important for two main reasons. From a formalization perspective, the constructive proof allows us to define a total evaluation function for the calculus, by induction on the strong normalization evidence. From the Curry–Howard perspective, considering the language as a logic, it ensures its consistency.

The formalization given in this work has some interesting features. One of them is the representation of intrinsically typed terms, where each term of the calculus is a direct implementation of a typing rule. As a result of this encoding, the implementation of functions over these typed terms preserves types, and the application of term isomorphisms is syntax directed. As a consequence, the relation $\rightleftarrows$ eliminates the type isomorphisms of a term, in the same way that the relation $\hookrightarrow$ eliminates applications and projections. This point is essential for making reduction sequences finite, since it prevents some term isomorphisms, such as $\langle r, s\rangle \rightleftarrows \langle s, r\rangle$, from being applied an indefinite number of times.

Another interesting feature is that the proof given for strong normalization follows Schäfer's technique, which, while framed within Tait and Girard's reducibility, presents several changes aimed at a smoother formalization process. Within that approach, we chose Kovács' variant, more appropriate for the constructive setting of Agda. It is important to note that this is the first version we know of for a formalization of the normalization proof of a lambda calculus modulo isomorphisms.

In this kind of calculus, the number of reduction cases that must be considered in the normalization proof is very large due to the equivalence relation on terms, so working in a rigorous setting such as that provided by Agda was very helpful.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. 1991. Explicit substitutions. *Journal of Functional Programming* 1, 4 (1991), 375–416. https://doi.org/10.1017/S0956796800000186

[2] Beniamino Accattoli and Alejandro Díaz-Caro. 2020. Functional Pearl: The Distributive λ-Calculus. In *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12073)*, Keisuke Nakano and Konstantinos Sagonas (Eds.). Springer, 33–49. https://doi.org/10.1007/978-3-030-59025-3_3

[3] T. Altenkirch and B. Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic (CSL '99)*. Springer-Verlag, Berlin, Heidelberg, 453–468.

[4] Roberto Di Cosmo. 1995. *Isomorphisms of types: from λ-calculus to information retrieval and language design.* Birkhauser, Switzerland.

[5] Alejandro Díaz-Caro and Gilles Dowek. 2019. Proof Normalization in a Logic Identifying Isomorphic Propositions. *LIPIcs (FSCD'19)* 131 (2019), 14:1–14:23.

[6] Alejandro Díaz-Caro and Gilles Dowek. 2023. Extensional proofs in a propositional logic modulo isomorphisms. *Theor. Comput. Sci.* 977 (2023), 114172. https://doi.org/10.1016/J.TCS.2023.114172

[7] Alejandro Díaz-Caro and Pablo E. Martínez López. 2015. Isomorphisms considered as equalities: Projecting functions and enhancing partial application through an implementation of λ^+. *ACM IFL* 2015, 9 (2015), 1–11.

[8] Jean-Yves Girard. 1971. Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types. In *Proceedings of the Second Scandinavian Logic Symposium*, J.E. Fenstad (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63–92. https://doi.org/10.1016/S0049-237X(08)70843-7

[9] Jean-Yves Girard. 1972. Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur. https://api.semanticscholar.org/CorpusID:117631778

[10] Conor McBride. 2005. Type-Preserving Renaming and Substitution. (2005).

[11] Steven Schäfer. 2019. *Engineering formal systems in constructive type theory.* Ph. D. Dissertation. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28298

[12] Cristian Sottile, Alejandro Díaz-Caro, and Pablo E. Martínez López. 2020. Polymorphic System I. In *IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020*, Olaf Chitil (Ed.). ACM, 127–137. https://doi.org/10.1145/3462172.3462198

[13] William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

[14] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda.* https://plfa.inf.ed.ac.uk/22.08/

Reducibility candidates modulo isomorphisms

Cristian Sottile*
Dpto. de Ciencia y Tecnología.
Universidad Nacional de Quilmes.
Bernal, Buenos Aires, Argentina.

Instituto de Investigación en Ciencias de la Computación (ICC). CONICET / Universidad de Buenos Aires.
Buenos Aires, Argentina.
csottile@dc.uba.ar

Alejandro Díaz-Caro*
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France.

Dpto. de Ciencia y Tecnología.
Universidad Nacional de Quilmes.
Bernal, Buenos Aires, Argentina.
alejandro.diaz-caro@inria.fr

Abstract

In this work in progress, we adapt the technique of reducibility candidates to a polymorphic system where types and terms are equated by isomorphisms, *i.e.*, if a term has type A and A is isomorphic to B, then the term also has type B. We follow Parigot's reducibility style, since neutrality of terms is not preserved by isomorphism. We define a way to relate sets in the family of candidates, following the type equivalence, so that a term not only satisfies the conditions syntactically posed by its type, but also those that the isomorphic types impose. We characterize the equivalence classes of all terms and analyze the new set of possible one–step reducts.

CCS Concepts: • **Theory of computation → Lambda calculus**; **Type theory**; **Proof theory**.

Keywords: Lambda calculus, Polymorphic type system, Type isomorphisms

ACM Reference Format:
Cristian Sottile and Alejandro Díaz-Caro. 2025. Reducibility candidates modulo isomorphisms. In *IFL 2025: Proceedings of the 37th Symposium on Implementation and Application of Functional Languages (IFL '25), October 1–3, 2025, Montevideo, Uruguay*. ACM, New York, NY, USA, 9 pages.

1 Introduction

Two types A and B are considered isomorphic ($\equiv$) if there exist functions f and g of types $A \Rightarrow B$ and $B \Rightarrow A$, such that composing g with f and f with g is equivalent (from a meta-theoretical perspective, *i.e.* not necessarily f and g must be writable in the calculus) to the identity in A and B. Di Cosmo et al. [2] characterized isomorphic types in different systems, finding axiomatic sets of isomorphisms that serve as the building blocks for all the existing ones. Typed λ–calculi modulo isomorphisms are built upon these characterizations, which covered several calculi among which we can find simply typed λ–calculus with pairs and System F. Díaz–Caro and Dowek first defined System I [3], which accounts for the simply typed λ–calculus (with pairs) modulo isomorphisms. Follow–up work implemented it in Haskell [5], adding numbers and recursion; introduced expansion rules [4], such as η, obtaining new properties; restricted isomorphisms to only distributivity in both typed and untyped λ–calculus [1], proving progress for the latter; and added polymorphism, including the usual constructions plus the isomorphisms related to the $\forall$ type constructor.

In the setting of typed λ–calculi modulo isomorphisms, where equivalence relations at the level of both types and terms are introduced, the strong normalization property carries several challenges. Mainly, these are: extend the definition of reducibility/interpretation to reflect the newly allowed term combinations, replicate the equivalence relation on types in the candidates, and properly characterize the possible one–step reducts of all terms. An adaptation of Tait [11] and Girard's [6, 7] reducibility technique, following Parigot's presentation [9], was introduced by Díaz–Caro and Dowek [3] to prove it for the simply typed version. Just like the original technique, the adaptation must undergo many changes when shifting to System F. In this work we propose a technique that deals with polymorphism following the same direction, while relying more heavily on Parigot's inductive definition of candidates, which is the place where we replicate the isomorphisms equivalence.

2 Classic reducibility

Reducibility sets were introduced to determine which restrictions each set of types should satisfy in order to comply with termination. To put it shortly, an induction on terms is not enough to prove $\mathcal{SN}$: let ts with t and s both $\mathcal{SN}$; since ts might itself form a redex, we need more information to conclude $\mathcal{SN}$ for it. Reducibility sets are the result of looking for something that could provide us with that information. Concretely, it proposes asking not only that terms are $\mathcal{SN}$,

*Funded by the European Union through the MSCA SE project QCOMICAL (Grant Agreement ID: 101182520).

IFL '25, Montevideo, Uruguay

but also to *follow* their type constructors asking to remain $\mathcal{SN}$ while these are eliminated. This property must hold until reaching atomic types, so we can define the interpretation of a type as simply $\mathcal{SN}$ for atomic types, and as remaining reducible when applied with reducible terms.

$$\begin{array}{rcl} [\![\tau]\!] & = & \mathcal{SN}_\tau \\ [\![A \to B]\!] & = & \{t : A \to B \mid \forall s \in [\![A]\!].ts \in [\![B]\!]\} \end{array}$$

Resuming the ts example, inductive hypothesis now tells us that t and s are both reducible, which means we can apply them and stay reducible, so the redex formed in ts is not a problem anymore. Essentially, asking for terms of types to be reducible is enough to guarantee termination.

Previous explanation works for simply typed λ–calculus, since atomic types cannot be eliminated in any way, so they can be safely interpreted as $\mathcal{SN}$. In the case of System F, atomic types are type variables; these cannot be eliminated either, but they can be replaced with other types. We could then define reducibility for variables simply by replacing the variable with its type argument, but the inductive definition would fail if we substitute the variable with a larger term than the one we started with. Let us show the naive definition and then an example.

$$\begin{array}{rcl} [\![X]\!] & = & \mathcal{SN}_X \\ [\![A \to B]\!] & = & \{t : A \mid \forall s \in [\![A]\!].ts \in [\![B]\!]\} \\ [\![\forall X.A]\!] & = & \{t : \forall X.A \mid \forall B \in K.tB \in [\![A[B/X]]\!]\} \end{array}$$

Then consider $[\![\forall X.X \to X]\!] = \{t : \forall X.X \to X \mid \forall B \in K.tB \in [\![B \to B]\!]\}$. If we take $B = \forall X.X \to X$, we get $[\![(\forall X.X \to X) \to (\forall X.X \to X)]\!] = \{t : (\forall X.X \to X) \to (\forall X.X \to X) \mid \forall s \in [\![\forall X.X \to X]\!].ts \in [\![\forall X.X \to X]\!]\}$. The dynamism of types prevents us from simultaneously following types syntax and term's type application, since the latter changes the former. Girard then introduced two changes to the definition of reducibility. Although they are meant to be one change, since they do not work separately, we will introduce them separately for clarity.

1. *Candidates*: Reducibility sets are clearly defined for simply typed λ–calculus: we just follow type structure and collect restrictions. In the case of System F, since we cannot define the notion of reducibility together with that of interpretation, we must take a distinct approach. Recall that, intuitively, reducibility sets are built by the restrictions that the terms of a type must satisfy to ensure termination. The reducibility set of a given type cannot be provided until knowing the reducibility sets corresponding to its free variables. Instead of expecting reducibility sets to match free variable restrictions, we will accept *reducibility candidates*, which is any set satisfying some specific conditions, and we will *build* the exact reducibility set throughout the Adequacy proof. One of the necessary restrictions of any reducibility set is to be a subset of $\mathcal{SN}$, so that will be the starting candidate. At any given point of the proof, the restrictions posed by the "growing" candidate will be enough to proceed.
2. *Parametricity*: To keep types structure from growing along with the definition of reducibility, we add another parameter ρ consisting of a map from type variables to reducibility candidates. Then, instead of asking for tB to be in $[\![A[B/X]]\!]$, we ask for it to be in $[\![A]\!]_{\{X \mapsto F\}}$, where F stands for any valid candidate of B.

$$\begin{array}{rcl} [\![X]\!]_\rho & = & \rho(X) \\ [\![A \to B]\!]_\rho & = & \{t : A \mid \forall s \in [\![A]\!]_\rho.ts \in [\![B]\!]_\rho\} \\ [\![\forall X.A]\!]_\rho & = & \{t : \forall X.A \mid \forall B \in K.tB \in [\![A]\!]_{\rho \cdot \{X \mapsto F\}}\} \end{array}$$

3 Reducibility candidates à la Parigot

Parigot introduced in 1997 [9] an alternative reducibility candidates technique, in order to prove $\mathcal{SN}$ in the $\lambda\mu$-calculus, which is a calculus for classical logic. The main difference with Girard's is that reducibility candidates are presented as an inductive membership relation, instead of being any set satisfying some properties. This allows us to reason inductively over candidates, since we have a derivation tree as a proof for all candidates that belong to the family, so we have structure to work with when dealing with one. Given a candidate, we can follow its structure to determine which are the possible ways to *apply* its terms, obtaining the set of *eliminators* that a term must withstand (*i.e.* be $\mathcal{SN}$[1] when applied with) to be a part of the candidate.

In this section we provide a presentation of the pure technique, *i.e.* restricting it to the framework of System F by ignoring the classical logic constructions of $\lambda\mu$.

Let $\mathcal{T}_A$ be the set of typed terms of type A, $\mathcal{T}_*$ the set of all typed terms, K the set of all types, $\mathcal{SN}_A$ the set of strongly normalisable typed terms of type A, and $\mathcal{SN}_A$ the set of all strongly normalisable typed terms.

We define the sets that will consist of the *building blocks* of reducibility candidates. The candidate constructors are defined according to the ways the system allows eliminations for each type. In System F, this means that, for instance, the constructor for the arrow type can be eliminated via application over an argument.

Definition 3.1 (Constructors of potential candidates)**.**

$$\begin{array}{rcl} F \tilde{\to} G & = & \{t \in \mathcal{T}_{A \to B} \mid \forall u \in F.\ ts \in G\} \\ & & \text{with} \quad F \subseteq \mathcal{T}_A \text{ and } G \subseteq \mathcal{T}_B \\ \tilde{\forall} B.U_B & = & \{t \in \mathcal{T}_{\forall X.A} \mid \forall B \in K.\ tB \in U_B\} \\ & & \text{with} \quad (U_B)_{B \in K} \text{ family s.t.} \\ & & \text{and} \quad U_B \subseteq \mathcal{T}_{A[B/X]} \text{ for all } B \in K \\ S \tilde{\Rightarrow}_A G & = & \{t \in \mathcal{T}_A \mid \forall \vec{u} \in S.\ t\vec{u} \in G\} \\ & & \text{with} \quad S \subseteq (\mathcal{T}_* \cup K)^{<\omega} \\ & & \text{and} \quad G \subseteq \mathcal{T} \end{array}$$

[1] Actually, terms must remain *reducible* through elimination, not only $\mathcal{SN}$. This is explained later.

Definition 3.2 (Family $(\mathcal{R}_A)_{A\in K}$ of sets of *reducibility candidates* of type A)**.**

$$\frac{}{\mathcal{SN}_A \in \mathcal{R}_A} \qquad \frac{U \in \mathcal{R}_A \quad V \in \mathcal{R}_B}{U\tilde{\to}V \in \mathcal{R}_{A\to B}}$$

$$\frac{X \subseteq \mathcal{R}_A}{\bigcap X \in \mathcal{R}_A} \qquad \frac{U_B \in \mathcal{R}_{A[B/X]} \text{ (for all } B \in K)}{\tilde{\forall}B.U_B \in \mathcal{R}_{\forall X.A}}$$

Lemma 3.3 (Candidates termination and inhabitation)**.** *If $F \in \mathcal{R}_A$, then $F \subseteq \mathcal{SN}_A$ and $x^A \in F$.*

Proof. We prove a slightly more general lemma: if $F \in \mathcal{R}_A$, then $F \subseteq \mathcal{SN}_A$ and $x^C\vec{u} \in \mathcal{SN}_A$ implies $x^C\vec{u} \in F$. By taking $C = A$ and $\vec{u} = \varepsilon$ we get $x^A \in F$. We proceed by induction on $F \in \mathcal{R}_A$.

1. $\mathcal{SN}_A$: trivially $\mathcal{SN}_A \subseteq \mathcal{SN}_A$ and $x^C\vec{u} \in \mathcal{SN}_A$ implies $x^C\vec{u} \in \mathcal{SN}_A$.
2. $\dfrac{U \in \mathcal{R}_A \quad V \in \mathcal{R}_B}{U\tilde{\to}V \in \mathcal{R}_{A\to B}}$: we recall that $U\tilde{\to}V$ is the set of terms t that, when applied to a U, belong to V. By IH, $V \subseteq \mathcal{SN}_B$ and $x^A \in U$. Since $tx^A \in V$ for any $t \in U\tilde{\to}V$, we have $tx^A \in \mathcal{SN}_B$. A *loop* in t would also produce a loop in tx^A, therefore $t \in \mathcal{SN}_{A\to B}$. Let $x^C\vec{u} \in \mathcal{SN}_{A\to B}$, then $x^C\vec{u}s$ has type B for all $s \in U$, since, by IH, $U \subseteq \mathcal{SN}_A$. Furthermore, since redexes in $x^C\vec{u}$ and s cannot interact and both terms are $\mathcal{SN}_*$, we can conclude $x^C\vec{u}s \in \mathcal{SN}_B$. We have then, by IH, that $x^C\vec{u}s \in V$; this satisfies the definition of $\tilde{\to}$, so we conclude $x^C\vec{u} \in U\tilde{\to}V$.
3. $\dfrac{\{F_i \mid i \in I\} \subseteq \mathcal{R}_A}{\bigcap\{F_i \mid i \in I\} \in \mathcal{R}_A}$: we must check that given any subset of candidates in $\mathcal{R}_A$, their intersection is also $\in \mathcal{SN}_A$; this follows from IH $F_i \in \mathcal{SN}_A$. Same goes for $x^C\vec{u} \in \mathcal{SN}_A$, since by IH $x^C\vec{u} \in \mathcal{SN}_A$ implies $x^C\vec{u} \in F_i$ for all $i \in I$.
4. $\dfrac{U_B \in \mathcal{R}_{A[B/X]} \text{ (for all } B \in K)}{\tilde{\forall}B.U_B \in \mathcal{R}_{\forall X.A}}$: we recall that $\tilde{\forall}B.U_B$ is the set of terms t of type $\forall X.A$ such that for all B, $tB \in U_B$; *i.e.* type application puts them in the associated member of the family U. To prove that any t there is $\mathcal{SN}_{\forall X.A}$, we observe that $tB \in \mathcal{SN}_{A[B/X]}$ by IH, and conclude since redexes are the same in both terms. The variable result is even more direct than that of $\to$. Let $x^C\vec{u} \in \forall X.A \in \mathcal{SN}_{\forall X.A}$; then for all B, we have $x^C\vec{u}B \in \mathcal{SN}_{A[B/X]}$, and, by IH, $x^C\vec{u}B \in U_B$. Hence, since B is any, we conclude $x^C\vec{u} \in \tilde{\forall}B.U_B$.

□

As noted previously, the main distinction of Parigot's technique is that candidates have structure we can work with. Candidate constructors (Definition 3.1) pose specific restrictions over its terms, so each candidate of a same type have different restrictions. We are interested in being able to prove belonging of a term to a candidate without relying on neutrality (since we do not have reliable neutral terms). So instead of building the reducibility tree for neutral terms, we ask for more information from the candidate. That information is the set of all its possible *eliminations*, in a way such that proving the term is $\mathcal{SN}_*$applied with any of these is the same as proving the term belongs to the candidate. This is, we want to be able to characterize any candidate as a set of eliminators. The following definitions go in that direction.

Definition 3.4 (Orthogonal of a candidate)**.** We define the notion of orthogonals of a candidate F as the sets $X \subseteq (\mathcal{SN}_* \cup K)^{<\omega}$ such that $F = X\tilde{\Rightarrow}_A\mathcal{SN}_*$. We define the function $\cdot^\perp$ that produces the greatest orthogonal of a candidate, by induction on its belonging derivation.

$$\left(\overline{\mathcal{SN}_A}\right)^\perp = \varepsilon$$

$$\left(\frac{U \in \mathcal{R}_A \quad V \in \mathcal{R}_B}{U\tilde{\to}V \in \mathcal{R}_{A\to B}}\right)^\perp = U \times V^\perp$$

$$\left(\frac{\{F_i \mid i \in I\} \subseteq \mathcal{R}_A}{\bigcap\{F_i \mid i \in I\} \in \mathcal{R}_A}\right)^\perp = \bigcup\{F_i^\perp \mid i \in I\}$$

$$\left(\frac{U_B \in \mathcal{R}_{A[B/X]} \text{ (for all } B \in K)}{\tilde{\forall}B.U_B \in \mathcal{R}_{\forall X.A}}\right)^\perp = \bigcup_{B\in K}(\{B\} \times (U_B)^\perp)$$

Note: we will write $F^\perp$ instead of $F \in \mathcal{R}_A^\perp$, since when we deal with specific candidates we are actually dealing with their belonging derivation.

Lemma 3.5 (Orthogonality characterizes candidates)**.** *Let $F \in \mathcal{R}_A$. Then $F = F^\perp\tilde{\Rightarrow}_A\mathcal{SN}_*$.*

Proof. By induction on $F \in \mathcal{R}_A$.

1. $\mathcal{SN}_A$: trivial since $\mathcal{SN}_A = \varepsilon\tilde{\to}\mathcal{SN}_A$.
2. $\dfrac{U \in \mathcal{R}_A \quad V \in \mathcal{R}_B}{U\tilde{\to}V \in \mathcal{R}_{A\to B}}$: by IH $V = V^\perp\tilde{\Rightarrow}_B\mathcal{SN}_*$. Since $U \subseteq \mathcal{SN}_*$, we have $U \times V^\perp \subseteq (\mathcal{SN}_* \cup K)^{<\omega}$. Finally:

$$\begin{aligned}
U\tilde{\to}V &= \{t \in \mathcal{T}_{A\to B} \mid \forall s \in U.\ ts \in V\} \\
&= \{t \in \mathcal{T}_{A\to B} \mid \forall s \in U.\ ts \in V^\perp\tilde{\Rightarrow}_B\mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{A\to B} \mid \forall s \in U.\ \forall\vec{u} \in V^\perp.\ ts\vec{u} \in \mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{A\to B} \mid \forall (s,\vec{u}) \in U \times V^\perp.\ t(s,\vec{u}) \in \mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{A\to B} \mid \forall\vec{v} \in U \times V^\perp.\ t\vec{v} \in \mathcal{SN}_*\} \\
&= U \times V^\perp\tilde{\Rightarrow}_{A\to B}\mathcal{SN}_*
\end{aligned}$$

3. $\dfrac{\{F_i \mid i \in I\} \subseteq \mathcal{R}_A}{\bigcap\{F_i \mid i \in I\} \in \mathcal{R}_A}$: by IH $F_i = F_i^\perp\tilde{\Rightarrow}_A\mathcal{SN}_*$. Then

$$\begin{aligned}
\bigcap\{F_i \mid i \in I\} &= \bigcap\{F_i^\perp\tilde{\Rightarrow}_A\mathcal{SN}_* \mid i \in I\} \\
&= \left(\bigcup\{F_i^\perp \mid i \in I\}\right)\tilde{\Rightarrow}_A\mathcal{SN}_*
\end{aligned}$$

4. $\dfrac{U_B \in \mathcal{R}_{A[B/X]} \text{ (for all } B \in K)}{\tilde{\forall}B.U_B \in \mathcal{R}_{\forall X.A}}$: by IH we have $U_B = (U_B)^\perp\tilde{\Rightarrow}_{A[B/X]}\mathcal{SN}_*$. Then:

$$\tilde{\forall}B.U_B = \{t \in \mathcal{T}_{\forall X.A} \mid \forall B.\ tB \in U_B\}$$

$$
\begin{aligned}
&= \{t \in \mathcal{T}_{\forall X.A} \mid \forall B.\ tB \in (U_B)^{\perp} \tilde{\Rightarrow}_{A[B/X]} \mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{\forall X.A} \mid \forall B.\ \forall \vec{u} \in (U_B)^{\perp}.\ tB\vec{u} \in \mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{\forall X.A} \mid \forall (B, \vec{u}) \in \{B\} \times (U_B)^{\perp}.\ t(B, \vec{u}) \in \mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{\forall X.A} \mid \forall \vec{v} \in \bigcup_{B \in K} (\{B\} \times (U_B)^{\perp}).\ t\vec{v} \in \mathcal{SN}_*\} \\
&= \bigcup_{B \in K} (\{B\} \times (U_B)^{\perp}) \tilde{\Rightarrow}_{\forall X.A} \mathcal{SN}_* \qquad \square
\end{aligned}
$$

As stated before, the interpretation of a type is the function that tells us which are the restrictions that its terms must satisfy in order to later prove termination. In Parigot style, we use the candidate constructors to build the set, and then we prove that the set built is indeed a candidate for the type.

Definition 3.6 (Interpretation of types)**.** We define the set $[\![A]\!]_\rho$ by induction on A.

$$
\begin{aligned}
[\![X]\!]_\rho &= \rho(X) \\
[\![A \to B]\!]_\rho &= [\![A]\!]_\rho \tilde{\to} [\![B]\!]_\rho \\
[\![\forall X.A]\!]_\rho &= \tilde{\forall} B. \bigcap \{[\![A]\!]_{\rho \cdot [F/X]} | F \in \mathcal{R}_B\}
\end{aligned}
$$

Intuitively, we follow the type constructors and include restrictions accordingly:

- for variables, we simply provide the candidate in the context;
- for arrows, we ask to remain reducible when applied to a reducible term;
- for universal quantifiers, we ask to remain reducible when applied to an arbitrary type, under all the reducibility candidates for the given type.

Definition 3.7 (Terms, Types, and Candidates mappings)**.** Let $\theta : Var \to \mathcal{T}_*$, $\sigma : TVar \to K$, and $\rho : TVar \to \mathcal{R}_*$. We define the following notions of validity, noted $\vDash$:

1. $\sigma, \rho \vDash A$ iff for all free type variable X in A, it holds that $\rho(X) \in \mathcal{R}_{\sigma(X)}$.
2. $\theta, \sigma, \rho \vDash \Gamma$ iff for all x^A in Γ, it holds that $\theta(x^{\sigma(A)}) \in [\![A]\!]_\rho$.

Lemma 3.8 (Interpretation is a candidate)**.** *Let σ and ρ such that $\sigma, \rho \vDash A$. Then $[\![A]\!]_\rho \in \mathcal{R}_{\sigma(A)}$.*

Proof. By induction on A:

1. X: $[\![X]\!]_\rho$ equals $\rho(X)$, which belongs to $\mathcal{R}_{\sigma(X)}$ by hypothesis.
2. $A \to B$: by IH, $[\![A]\!]_\rho \in \mathcal{R}_{\sigma(A)}$ and $[\![B]\!]_\rho \in \mathcal{R}_{\sigma(B)}$. Then, by Definition 3.2, $[\![A]\!]_\rho \tilde{\to} [\![B]\!]_\rho = [\![A \to B]\!]_\rho \in \mathcal{R}_{\sigma A \to \sigma B} = \mathcal{R}_{\sigma(A \to B)}$.
3. $\forall X.A$: let B and $F \in \mathcal{R}_B$. By IH, $[\![A]\!]_{\rho \cdot [F/X]} \in \mathcal{R}_{\sigma \cdot [B/X] A} = \mathcal{R}_{(\sigma A)[B/X]}$. We have then that $\{[\![A]\!]_{\rho \cdot [F/X]} \mid F \in \mathcal{R}_B\} \subseteq \mathcal{R}_{(\sigma A)[B/X]}$, so, by Definition 3.2, $\bigcap \{[\![A]\!]_{\rho \cdot [F/X]} \mid F \in \mathcal{R}_B\} \subseteq \mathcal{R}_{(\sigma A)[B/X]}$. Then, again by Definition 3.2, since previous statement holds for any type B, we have $\Pi B. \bigcap \{[\![A]\!]_{\rho \cdot [F/X]} \mid F \in \mathcal{R}_B\} = [\![\forall X.A]\!]_\rho \in \mathcal{R}_{\forall X. \sigma A} = \mathcal{R}_{\sigma(\forall X.A)}$. $\square$

Lemma 3.9 (Substitution)**.** $[\![A[B/X]]\!]_\rho = [\![A]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$

Proof. By induction on A. $\square$

Lemma 3.10 ($\mathcal{SN}$ –closure under head expansion)**.**

1. *If $s \in \mathcal{SN}_*$ and $t[s/x^A]\vec{u} \in \mathcal{SN}_*$ then $((\lambda x^A.t)s)\vec{u} \in \mathcal{SN}_*$*
2. *If $B \in K$ and $t[B/X]\vec{u} \in \mathcal{SN}_*$ then $((\Lambda X.t)B)\vec{u} \in \mathcal{SN}_*$*

Proof. By induction on $|s| + |t| + |\vec{u}|$, with $|\cdot|$ the longest reduction chain from a given term. $\square$

Lemma 3.11 (Adequacy)**.** *If $\Gamma \vdash t : A$ and $\theta, \sigma, \rho \vDash \Gamma$, then $\theta(\sigma(t)) \in [\![A]\!]_\rho$.*

Proof. By induction on the typing derivation.

1. $\dfrac{}{x : A \in \Gamma}$: by hypothesis, $\theta\sigma x \in [\![A]\!]_\rho$.
2. $\dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash ts : B}$: by IH we have $\theta\sigma t \in [\![A \to B]\!]_\rho = [\![A]\!]_\rho \tilde{\to} [\![B]\!]_\rho$ and $\theta\sigma s \in [\![A]\!]_\rho$, so $(\theta\sigma t)(\theta\sigma s) \in [\![B]\!]_\rho$. By substitution definition, $\theta\sigma(ts) \in [\![B]\!]_\rho$.
3. $\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B}$: to prove that $\theta\sigma(\lambda x^A.t) = \lambda x^{\sigma A}.\theta\sigma t \in [\![A \to B]\!]_\rho = [\![A]\!]_\rho \tilde{\to} [\![B]\!]_\rho$, we must check that for all $s \in [\![A]\!]_\rho$, we have $(\lambda x^{\sigma A}.\theta\sigma t)s \in [\![B]\!]_\rho$. By IH, $\theta\sigma(t)[s/x^{\sigma A}] \in [\![B]\!]_\rho$. By Lemma 6.9, we have $[\![B]\!]_\rho \in \mathcal{R}_{\sigma B}$. Thus, by Lemma 6.5, $[\![B]\!]_\rho = [\![B]\!]_\rho^{\perp} \tilde{\Rightarrow}_{\sigma B} \mathcal{SN}_*$. By definition of $\tilde{\Rightarrow}_{\sigma B}$ we have that, for all $\vec{u} \in [\![B]\!]_\rho^{\perp}$, it holds that $\theta\sigma t[s/x^{\sigma A}]\vec{u} \in \mathcal{SN}_*$. By Lemma 3.10, we have $(\lambda x^{\sigma A}.\theta\sigma(t))s\vec{u} \in \mathcal{SN}_*$. Hence, $(\lambda x^{\sigma A}.\theta\sigma t)s$ satisfies the belonging conditions of $[\![B]\!]_\rho^{\perp} \tilde{\Rightarrow}_B \mathcal{SN}_* = [\![B]\!]_\rho$, and we conclude $\lambda x^{\sigma A}.\theta\sigma t \in [\![A \to B]\!]_\rho$.
4. $\dfrac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash tB : A[B/X]}$: by IH $\theta\sigma t \in [\![\forall X.A]\!]_\rho$, so for all C and $F \in \mathcal{R}_C$, we have $(\theta\sigma t)C \in [\![A]\!]_{\rho \cdot [F/X]}$. To prove that $\theta\sigma(tB) = \theta\sigma t \sigma B \in [\![A[B/X]]\!]_\rho$, we consider two cases depending on X occurring free in A:
   a. $X \in FVA$: since $\sigma, \rho \vDash A[B/X]$, it also holds that $\sigma, \rho \vDash B$. By Lemma 6.9, we have $[\![B]\!]_\rho \in \mathcal{R}_{\sigma B}$. By IH, $(\theta\sigma t)\sigma B \in [\![A]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$. Since $[\![A]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$ equals $[\![A[B/X]]\!]_\rho$ by Lemma 6.10, we conclude.
   b. $X \notin FVA$: then $A[B/X] = A$, so we must prove that $\theta\sigma t \sigma B \in [\![A]\!]_\rho$. Let F be any of $\mathcal{R}_{\sigma B}$, *e.g.* $\mathcal{SN}_{\sigma B}$. Then, by IH, $(\theta\sigma t)(\sigma B) \in [\![A]\!]_{\rho \cdot [F/X]}$. Note that, for all A' such that $Y \notin FVA'$, it holds that $[\![A']\!]_{\rho' \cdot [G/Y]} = [\![A']\!]_{\rho'}$. Then $[\![A]\!]_{\rho \cdot [F/X]} = [\![A]\!]_\rho$, so we conclude.
5. $\dfrac{\Gamma \vdash t : A \quad X \notin FV(\Gamma)}{\Gamma \vdash \Lambda X.t : \forall X.A}$: we must prove that $\theta\sigma(\Lambda X.t) \in [\![\forall X.A]\!]_\rho$, *i.e.* $(\Lambda X.\theta\sigma t)B \in [\![A]\!]_{\rho \cdot [F/X]}$ for all B and $F \in \mathcal{R}_B$. Since θ remains unaffected by $[B/X]$ due to $X \notin FV(A)$, we have $(\theta\sigma t)[B/X] = \theta(\sigma \cdot [B/X])t$. Furthermore, $\theta, (\sigma \cdot [B/X]), (\rho \cdot [F/X]) \vDash \Gamma$, so, by IH, $\theta(\sigma \cdot [B/X])t \in [\![A]\!]_{\rho \cdot [F/X]}$. By Lemma 6.5, it holds that $(\theta(\sigma \cdot [B/X])t)\vec{u} = (\theta\sigma t)[B/X]\vec{u} \in \mathcal{SN}_*$ for all $\vec{u} \in [\![A]\!]_{\rho \cdot [F/X]}^{\perp}$. By Lemma 3.10, we have $(\Lambda X.\theta\sigma t)B\vec{u} \in$

$\mathcal{SN}_*$. Hence, $(\Lambda X.\theta\sigma t)B$ satisfies the belonging conditions of $[\![A]\!]^{\perp}_{\rho\cdot[F/X]} \tilde{\Rightarrow}_{\sigma A} \mathcal{SN}_* = [\![A]\!]_{\rho\cdot[F/X]}$, and we conclude $\theta\sigma(\Lambda X.t) \in [\![\forall X.A]\!]_\rho$.

□

4 System F modulo isomorphisms

Polymorphic System I (PSI) [10] is an extension of System I [3] to polymorphic types. In this section we present the system.

The syntax of types coincides with that of System F [8, Chapter 11] with pairs:

$$A \quad ::= \quad X \mid A \to A \mid A \times A \mid \forall X.A$$

where $X \in \mathcal{TV}ar$, a set of type variables.

In System I, variables can only have so-called *prime* types, which are those that are not equivalent to a product. In order to properly maintain this limitation in a System F based calculus, we must restrict type application: each type appearing at the level of terms must be prime. We denote prime types with $\mathbb{A}, \mathbb{B}, \mathbb{C}, \ldots$, and the set of prime types with $\mathbb{K}$.

Terms are then:

$$t \ ::= \ x^{\mathbb{A}} \mid \lambda x^{\mathbb{A}}.t \mid tt \mid \langle t,t\rangle \mid \pi_A t \mid \Lambda X.t \mid t\mathbb{A}$$

Usual projection $\pi_1 t$ does not make sense in these systems. Consider $\pi_1\langle x^{\mathbb{A}}, y^{\mathbb{B}}\rangle$; since the pair has type $\mathbb{A}\times\mathbb{B}$ which is equivalent to $\mathbb{B}\times\mathbb{A}$, first element can be both x and y. Commutativity and associativity make pairs positions indistinguishable. The projection that makes sense here is *over types*: $\pi_{\mathbb{A}}\langle x^{\mathbb{A}}, y^{\mathbb{B}}\rangle$ is indeed x. When both elements of the pair are of the same type, the system behaves non–deterministically. Determinism can be easily recovered by adding type constants $\mathbb{1}^{\mathbb{K}}$ and $\mathbb{2}^{\mathbb{K}}$, inhabited by 1 and 2 respectively, and defining $\pi_1\langle t,s\rangle$ as $(\pi_{\mathbb{1}^{\mathbb{K}}\to A}\langle\lambda x^{\mathbb{1}^{\mathbb{K}}}.t, \lambda x^{\mathbb{2}^{\mathbb{K}}}.s\rangle)1$.

The typing rules are:

$$\frac{}{\Gamma, x:\mathbb{A}\vdash x:\mathbb{A}} \qquad \frac{\Gamma\vdash t:A \qquad A\equiv B}{\Gamma\vdash t:B}$$

$$\frac{\Gamma, x:\mathbb{A}\vdash t:B}{\Gamma\vdash \lambda x^{\mathbb{A}}.t:\mathbb{A}\to B} \qquad \frac{\Gamma\vdash t:A\to B \qquad \Gamma\vdash s:A}{\Gamma\vdash ts:B}$$

$$\frac{\Gamma\vdash t:A \qquad \Gamma\vdash s:B}{\Gamma\vdash\langle t,s\rangle:A\times B} \qquad \frac{\Gamma\vdash t:A\wedge B}{\Gamma\vdash\pi_A(t):A}$$

$$\frac{\Gamma\vdash t:A \qquad X\notin FTV(\Gamma)}{\Gamma\vdash\Lambda X.t:\forall X.A} \qquad \frac{\Gamma\vdash t:\forall X.A}{\Gamma\vdash t\mathbb{B}:A[\mathbb{B}/X]}$$

Note that the arrow introduction rule puts a prime type to the left, but the elimination rule allows any type to the left. This happens because the type of the function can be obtained by rule ≡.

The type and term equivalences are:

$$A\wedge B \equiv B\wedge A \tag{1}$$
$$A\wedge(B\wedge C)\equiv(A\wedge B)\wedge C \tag{2}$$
$$A\Rightarrow(B\wedge C)\equiv(A\Rightarrow B)\wedge(A\Rightarrow C) \tag{3}$$
$$(A\wedge B)\Rightarrow C\equiv A\Rightarrow B\Rightarrow C \tag{4}$$
$$\text{If } X\notin FTV(A),\quad \forall X.(A\Rightarrow B)\equiv A\Rightarrow\forall X.B \tag{5}$$
$$\forall X.(A\wedge B)\equiv\forall X.A\wedge\forall X.B \tag{6}$$

$$\langle r,s\rangle \rightleftarrows \langle s,r\rangle \qquad (\text{COMM})$$
$$\langle r,\langle s,t\rangle\rangle \rightleftarrows \langle\langle r,s\rangle,t\rangle \qquad (\text{ASSO})$$
$$\lambda x^A.\,\langle r,s\rangle \rightleftarrows \langle\lambda x^A.\,r, \lambda x^A.\,s\rangle \qquad (\text{DIST}_\lambda)$$
$$\langle r,s\rangle t \rightleftarrows \langle rt, st\rangle \qquad (\text{DIST}_{\text{app}})$$
$$r\langle s,t\rangle \rightleftarrows rst \qquad (\text{CURRY})$$
$$\text{If } X\notin FTV_A(A),\quad \Lambda X.\lambda x^A.r \rightleftarrows \lambda x^A.\Lambda X.r \qquad (\text{P-COMM}_{\forall_i\Rightarrow_i})$$
$$(\lambda x^A.\,r)[B] \rightleftarrows \lambda x^A.\,r[B] \qquad (\text{P-COMM}_{\forall_e\Rightarrow_i})$$
$$\Lambda X.\langle r,s\rangle \rightleftarrows \langle\Lambda X.r, \Lambda X.s\rangle \qquad (\text{P-DIST}_{\forall_i\wedge_i})$$
$$\langle r,s\rangle[A] \rightleftarrows \langle rA, sA\rangle \qquad (\text{P-DIST}_{\forall_e\wedge_i})$$
$$\pi_{\forall X.A}(\Lambda X.r) \rightleftarrows \Lambda X.\pi_A(r) \qquad (\text{P-DIST}_{\forall_i\wedge_e})$$
$$\text{If } r:\forall X.(B\wedge C),\quad (\pi_{\forall X.B}r)[A] \rightleftarrows \pi_{[X:=A]B}(r[A]) \qquad (\text{P-DIST}_{\forall_e\wedge_e})$$

Reduction is given by the rules corresponding to each elimination, preceeded by arbitrary term equivalences.

$$\text{If } \Gamma\vdash s:A,\quad (\lambda x^A.\,r)s \to [x:=s]r \qquad (\beta_\lambda)$$
$$(\Lambda X.r)[A] \to [X:=A]r \qquad (\beta_\Lambda)$$
$$\text{If } \Gamma\vdash r:A,\quad \pi_A(\langle r,s\rangle)\to r \qquad (\pi)$$

$$\rightsquigarrow \ ::= \rightleftarrows^* \circ \to$$

$$\frac{\lambda x^A.r \rightleftarrows \lambda x^A.s}{r\rightleftarrows s} \quad \frac{rt\rightleftarrows st}{r\rightleftarrows s} \quad \frac{tr\rightleftarrows ts}{r\rightleftarrows s}$$

$$\frac{\langle t,r\rangle\rightleftarrows\langle t,s\rangle}{r\rightleftarrows s} \quad \frac{\pi_A(r)\rightleftarrows\pi_A(s)}{r\rightleftarrows s} \quad \frac{\langle r,t\rangle\rightleftarrows\langle s,t\rangle}{r\rightleftarrows s}$$

$$\frac{\Lambda X.r\rightleftarrows\Lambda X.s}{r\rightleftarrows s} \quad \frac{rA\rightleftarrows sA}{r\rightleftarrows s}$$

$$\frac{\lambda x^A.r \rightsquigarrow \lambda x^A.s}{r\rightsquigarrow s} \quad \frac{rt\rightsquigarrow st}{r\rightsquigarrow s} \quad \frac{tr\rightsquigarrow ts}{r\rightsquigarrow s}$$

$$\frac{\langle t,r\rangle\rightsquigarrow\langle t,s\rangle}{r\rightsquigarrow s} \quad \frac{\pi_A(r)\rightsquigarrow\pi_A(s)}{r\rightsquigarrow s} \quad \frac{\langle r,t\rangle\rightsquigarrow\langle s,t\rangle}{r\rightsquigarrow s}$$

$$\frac{\Lambda X.r\rightsquigarrow\Lambda X.s}{r\rightsquigarrow s} \quad \frac{rA\rightsquigarrow sA}{r\rightsquigarrow s}$$

Definition 4.1 (Measure on types)**.** Let $\mathsf{m}(A)$ be a measure stable by isomorphisms, *i.e.* such that if $A\equiv B$ then $\mathsf{m}(A) = \mathsf{m}(B)$.

5 Issues with isomorphisms and classical reducibility

The core idea of reducibility is, as we described earlier, to determine what are the circumstances that the terms of a given type must *resist* with "good" behaviour, and what "good" behaviour means. The way to determine these is to follow the structure of types when we have them, and to parameterize and find out in a stepwise manner when we do not have a structure to follow.

The first challenge is that the structure of types is not stable through isomorphisms. For instance, in System F a pair $\langle \Lambda X.t, \Lambda X.s\rangle$ can only be projected, so the circumstance to resist is simply projection; in Polymorphic System I, the same term can also be applied to a type $\mathbb{D}$, so the circumstances to resist now depend not only on its type structure, but on the structure of all its isomorphic types. We must define reducibility and interpretation in such a way that all these circumstances are properly considered.

The second challenge is that one of the main properties of reducibility candidates from Girard's approach, the so-called CR3, requires neutrality of terms to be stable, which does not hold for terms equated by isomorphisms. A neutral term is one that is not an introduction, *e.g.*variables and eliminations. Polymorphic System I equates $\langle(\Lambda X.t)\mathbb{D}, (\Lambda X.s)\mathbb{D}\rangle$, an introduction, to $\langle \Lambda X.t, \Lambda X.s\rangle\mathbb{D}$, an elimination. CR3 is used when *testing* that an introduction term behaves well under its immediate restrictions, by showing that if it is neutral and all its reducts are well-behaved, then it is also well-behaved, where well-behaved means recursively (on the type) reducible. Since we cannot rely on this property, we take Parigot's approach, which considers all the possible ways to apply the term and shows $\mathcal{SN}$ for all of them. To do so, Parigot gives candidates an inductive structure, in order to be able to determine what are all the ways that the term can be applied. We come across the first challenge again: the equivalence on types that required us to make terms resist not only their type eliminations, but also those of their isomorphic ones, also requires us to relate the candidates of isomorphic types in the family of candidates.

A third challenge, which is more general and not particularly tied to reducibility, is that terms have more one-step reducts in Polymorphic System I than in System F. Consider $(\Lambda X.\langle t, s\rangle)\mathbb{D}$. In System F, it has three kinds of reducts: those in t, s, and the type application. In Polymorphic System I, it is equivalent to $\langle(\Lambda X.t)\mathbb{D}, (\Lambda X.s)\mathbb{D}\rangle$, which has four kinds of reducts: those in t, s, and now two type applications instead of one. The proof requires us to know which are all the one-step reducts of a term, so we must understand all the possible ways in which the term equivalence makes redexes arise.

6 Reducibility candidates modulo isomorphisms

In this section we sketch the structure of the reducibility proof adapted to the equivalences induced by isomorphisms, explaining what changes must be done to each part and presenting some finished and in progress lemmas and proofs.

We recall that candidate constructors are the *building blocks* of reducibility candidates, and that they are defined according to the ways the system allows eliminations for each type. Although Polymorphic System I allows unusual eliminations for each type, this will modify how the family of candidates is populated, not the constructors, which remain the same. The only change we introduce here is the candidate constructor corresponding to our type-based pair projection.

Definition 6.1 (Constructors of potential candidates)**.**

$$
\begin{array}{rcl}
\prod_{i=1}^{n} F_i & = & \{t \in \mathcal{T}_{A_1\times\ldots\times A_n} \mid \forall i \in 1..n.\ \pi_{A_i} t \in F_i\} \\
 & & \text{with } F_i \subseteq \mathcal{T}_{A_i} \\
F \tilde{\to} G & = & \{t \in \mathcal{T}_{A\to B} \mid \forall u \in F.\ ts \in G\} \\
 & & \text{with } F \subseteq \mathcal{T}_A \text{ and } G \subseteq \mathcal{T}_B \\
\tilde{\forall}\mathbb{B}.U_{\mathbb{B}} & = & \{t \in \mathcal{T}_{\forall X.A} \mid \forall \mathbb{B} \in \mathbb{K}.\ t\mathbb{B} \in U_{\mathbb{B}}\} \\
 & & \text{with } (U_{\mathbb{B}})_{\mathbb{B}\in\mathbb{K}} \text{ family s.t.} \\
 & & U_{\mathbb{B}} \subseteq \mathcal{T}_{A[\mathbb{B}/X]} \text{ for all } \mathbb{B} \in \mathbb{K} \\
S \tilde{\Rightarrow}_A G & = & \{t \in \mathcal{T}_A \mid \forall \vec{u} \in S.\ t\vec{u} \in G\} \\
 & & \text{with } S \subseteq (\mathcal{T}_* \cup K \cup \{\overline{\pi_B}\})^{<\omega} \text{ and } G \subseteq \mathcal{T}
\end{array}
$$

We write $F_1 \tilde{\times} F_2$ for $\prod_{i=1}^{2} F_i$. We use $\overline{\pi_A}$ to refer to the projection as a postfix operator.

We stated that the intuition for candidates is to determine which restrictions should be satisfied by the terms of a given type, where the restrictions come from the interactions under which the term should remain reducible. In regular systems, these interactions are mainly the eliminations of the principal constructor of the type. In systems modulo isomorphisms, each type allows the eliminations of all its isomorphic types, for instance when a pair of functions is directly applied without a prior projection. The family of reducibility candidates must reflect the new restrictions. To do so, we introduce a new belonging rule: if we have two candidates F and G of two isomorphic types A and B, then the intersection of the candidates is a candidate that belongs to the set of candidates of both types. This way we are asking the terms in the new candidate to *resist* more interactions, thus satisfying more restrictions: those of F in $\mathcal{R}_A$ and those of G in $\mathcal{R}_B$.

Definition 6.2 (Family $(\mathcal{R}_A)_{A\in K}$ of sets of *reducibility candidates* of type A)**.**

$$
\frac{}{\mathcal{SN}_A \in \mathcal{R}_A} \qquad \frac{U \in \mathcal{R}_A \quad V \in \mathcal{R}_B}{U \tilde{\to} V \in \mathcal{R}_{A\to B}}
$$

$$
\frac{X \subseteq \mathcal{R}_A}{\bigcap X \in \mathcal{R}_A} \qquad \frac{U_B \in \mathcal{R}_{A[B/X]} \text{ (for all } B \in K)}{\tilde{\forall} B.U_B \in \mathcal{R}_{\forall X.A}}
$$

$$\frac{F \in \mathcal{R}_A \quad G \in \mathcal{R}_B}{F \tilde{\times} G \in \mathcal{R}_{A\times B}} \qquad \frac{F \in \mathcal{R}_A \quad G \in \mathcal{R}_B \quad A \equiv B}{F \cap G \in \mathcal{R}_A}$$

Lemma 6.3 (Candidates termination and inhabitation)**.** *If $F \in \mathcal{R}_A$, then $F \subseteq \mathcal{SN}_A$ and $x^A \in F$.*

Proof. By induction on the derivation of $F \in \mathcal{R}_A$. We extend the proof of Lemma 3.3 by adding the two new cases:

1. $\dfrac{F \in \mathcal{R}_A \quad G \in \mathcal{R}_B}{F \tilde{\times} G \in \mathcal{R}_{A\times B}}$: We recall that $F \tilde{\times} G$ is the set of terms t of type $A \times B$ that, when projected to A (resp. B), belong to F (resp. G). By IH $\pi_A t \in F \subseteq \mathcal{SN}_A$ and $\pi_B t \in G \subseteq \mathcal{SN}_B$. Since both projections are $\mathcal{SN}$, then t is also $\mathcal{SN}$. Let $x^C \vec{u} \in \mathcal{SN}_{A\times B}$. By IH $\pi_A(x^C \vec{u}) \in F$ and $\pi_B(x^C \vec{u}) \in G$, so by Definition 6.2 $x^C \vec{u} \in F \tilde{\times} G$.
2. $\dfrac{F \in \mathcal{R}_A \quad G \in \mathcal{R}_B \quad A \equiv B}{F \cap G \in \mathcal{R}_A}$: Let $t \in F \cap G$. Then $t \in F$ so, by IH, $t \in \mathcal{SN}_A$. Let $x^C \vec{u} \in \mathcal{SN}_A$. By IH $x^C \vec{u} \in F$. Since $A \equiv B$, it holds that $x^C \vec{u} \in \mathcal{SN}_B$, and by IH $x^C \vec{u} \in G$. Therefore $x^C \vec{u} \in F \cap G$.

□

Definition 6.4 (Orthogonal of a candidate)**.** We define the notion of orthogonals of a candidate F as the sets $X \subseteq (\mathcal{SN}_* \cup K \cup \{\overline{\pi_A}\})^{<\omega}$ such that $F = X \tilde{\Rightarrow}_A \mathcal{SN}_*$. We define the function $\cdot^\perp$ that produces the greatest orthogonal of a candidate, by induction on its belonging derivation. The differences with Definition 3.4 are only the newly added cases.

$$\left(\overline{\mathcal{SN}_A}\right)^\perp = \varepsilon$$

$$\left(\frac{U \in \mathcal{R}_A \quad V \in \mathcal{R}_B}{U \tilde{\to} V \in \mathcal{R}_{A\to B}}\right)^\perp = U \times V^\perp$$

$$\left(\frac{\{F_i \mid i \in I\} \subseteq \mathcal{R}_A}{\bigcap\{F_i \mid i \in I\} \in \mathcal{R}_A}\right)^\perp = \bigcup\{F_i^\perp \mid i \in I\}$$

$$\left(\frac{U_B \in \mathcal{R}_{A[B/X]} \text{ (for all } B \in K)}{\tilde{\forall} B.U_B \in \mathcal{R}_{\forall X.A}}\right)^\perp = \bigcup_{B\in K}(\{B\} \times (U_B)^\perp)$$

$$\left(\frac{F \in \mathcal{R}_A \quad G \in \mathcal{R}_B}{F \tilde{\times} G \in \mathcal{R}_{A\times B}}\right)^\perp = (\overline{\pi_A} \times F^\perp) \cup (\overline{\pi_B} \times G^\perp)$$

$$\left(\frac{F \in \mathcal{R}_A \quad G \in \mathcal{R}_B \quad A \equiv B}{F \cap G \in \mathcal{R}_A}\right)^\perp = F^\perp \cup G^\perp$$

Note: we will write $F^\perp$ instead of $F \in \mathcal{R}_A^\perp$, since when we deal with specific candidates we are actually dealing with their belonging derivation.

Lemma 6.5 (Orthogonality characterizes candidates)**.** *Let $F \in \mathcal{R}_A$. Then $F = F^\perp \tilde{\Rightarrow}_A \mathcal{SN}_*$.*

Proof. By induction on $F \in \mathcal{R}_A$. We extend the proof from Lemma 3.5 with the new cases.

1. $\dfrac{F \in \mathcal{R}_A \quad G \in \mathcal{R}_B}{F \tilde{\times} G \in \mathcal{R}_{A\times B}}$: by IH $F = F^\perp \tilde{\Rightarrow}_A \mathcal{SN}_*$ and $G = G^\perp \tilde{\Rightarrow}_B \mathcal{SN}_*$.

$$\begin{aligned}
F \tilde{\times} G &= \{t \in \mathcal{T}_{A\times B} \mid \pi_A t \in F, \pi_B t \in G\} \\
&= \{t \in \mathcal{T}_{A\times B} \mid \pi_A t \in F^\perp \tilde{\Rightarrow}_A \mathcal{SN}_*, \pi_B t \in G^\perp \tilde{\Rightarrow}_B \mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{A\times B} \mid \forall \vec{u}_A \in F^\perp . \pi_A t \vec{u}_A \in \mathcal{SN}_*, \forall \vec{u}_B \in G^\perp . \pi_B t \vec{u}_B \in \mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{A\times B} \mid \forall (\overline{\pi_C}, \vec{u}_C) \in (\overline{\pi_A} \times F^\perp) \cup (\overline{\pi_B} \times G^\perp) . t \overline{\pi_C} \vec{u}_C \in \mathcal{SN}_*\} \\
&= \{t \in \mathcal{T}_{A\times B} \mid \forall \vec{v} \in (\overline{\pi_A} \times F^\perp) \cup (\overline{\pi_B} \times G^\perp) . t \vec{v} \in \mathcal{SN}_*\} \\
&= (\overline{\pi_A} \times F^\perp) \cup (\overline{\pi_B} \times G^\perp) \tilde{\Rightarrow}_{A\times B} \mathcal{SN}_*
\end{aligned}$$

2. $\dfrac{F \in \mathcal{R}_A \quad G \in \mathcal{R}_B \quad A \equiv B}{F \cap G \in \mathcal{R}_A}$: by IH $F = F^\perp \tilde{\Rightarrow}_A \mathcal{SN}_*$ and $G = G^\perp \tilde{\Rightarrow}_B \mathcal{SN}_*$.

$$\begin{aligned}
F \cap G &= F^\perp \tilde{\Rightarrow}_A \mathcal{SN}_* \cap G^\perp \tilde{\Rightarrow}_B \mathcal{SN}_* \\
&= (F^\perp \cup G^\perp) \tilde{\Rightarrow}_A \mathcal{SN}_*
\end{aligned}$$

Note that the set of terms of type A coincides with the set of terms of type B due to $A \equiv B$.

□

In System F, we followed the type constructors to define the interpretation of a type. In Polymorphic System I, we need to follow the type constructors of all the types isomorphic to A. Definition 6.1 states what is the direct restriction that terms of a given type should satisfy. We will consider the equivalence class of A, and ask the terms in $[\![A]\!]_\rho$ to satisfy, for all the types B isomorphic to A, the direct restriction of the principal type constructor of B together with those of the interpretation of its inner types. For instance, to define $[\![X \times Y]\!]_\rho$, we will ask terms in $X \times Y$ to satisfy the restrictions posed by $\rho(X) \tilde{\times} \rho(Y)$, and also those posed by $\rho(Y) \tilde{\times} \rho(X)$, since $Y \times X$ is the only type isomorphic to $X \times Y$.

Definition 6.6 (Interpretation of types)**.** We define the set $[\![A]\!]_\rho$ by induction on $\mathsf{m}(A)$.

1. In the base case we have $A = X$. We define $[\![X]\!]_\rho = \rho(X)$.
2. In the inductive case we $[\![A]\!]_\rho$ as the set of terms t such that:
   - for all B, C with $A \equiv B \times C$, we have $t \in [\![B]\!]_\rho \tilde{\times} [\![C]\!]_\rho$;
   - for all B, C with $A \equiv B \to C$, we have $t \in [\![B]\!]_\rho \tilde{\to} [\![C]\!]_\rho$;
   - for all B with $A \equiv \forall X.B$, we have $t \in \tilde{\forall}\mathbb{D}. \bigcap\{[\![B]\!]_{\rho \cdot [F/X]} | F \in \mathcal{R}_\mathbb{D}\}$ for all $\mathbb{D}$.

Remark 6.7 (Stability of interpretation through isomorphisms)*.* $A \equiv B$ implies $[\![A]\!]_\rho = [\![B]\!]_\rho$.

Definition 6.8 (Terms, Types, and Candidates mappings)**.** Let $\theta : Var \to \mathcal{T}_*$, $\sigma : TVar \to K$, and $\rho : TVar \to \mathcal{R}_*$. We define the following notions of validity, noted $\vDash$:

1. $\sigma, \rho \vDash A$ iff for all free type variable X in A, it holds that $\rho(X) \in \mathcal{R}_{\sigma(X)}$.

2. $\theta, \sigma, \rho \vDash \Gamma$ iff for all x^A in Γ, it holds that $\theta(x^{\sigma(A)}) \in [\![A]\!]_\rho$.

Lemma 6.9 (Interpretation is a candidate). *Let σ and ρ such that $\sigma, \rho \vDash A$. Then $[\![A]\!]_\rho \in \mathcal{R}_{\sigma(A)}$.*

Proof. By induction on $\mathsf{m}(A)$.

1. In the base case we have $A = X$, so by hypothesis $\rho(X) \in \mathcal{R}_{\sigma X}$.
2. In the inductive case we consider all the possible equivalences:
   - if $A \equiv B \times C$, then, by IH, $[\![B]\!]_\rho \in \mathcal{R}_{\sigma(B)}$ and $[\![C]\!]_\rho \in \mathcal{R}_{\sigma(C)}$. Then, by Definition 6.2, $[\![B]\!]_\rho \tilde{\times} [\![C]\!]_\rho \in \mathcal{R}_{\sigma B \times \sigma C} = \mathcal{R}_{\sigma(B \times C)}$.
   - if $A \equiv B \to C$, then, by IH, $[\![B]\!]_\rho \in \mathcal{R}_{\sigma(B)}$ and $[\![C]\!]_\rho \in \mathcal{R}_{\sigma(C)}$. Then, by Definition 6.2, $[\![B]\!]_\rho \tilde{\to} [\![C]\!]_\rho \in \mathcal{R}_{\sigma B \to \sigma C} = \mathcal{R}_{\sigma(B \to C)}$.
   - if $A \equiv \forall X.B$, then let C and $F \in \mathcal{R}_C$. By IH, $[\![B]\!]_{\rho \cdot [F/X]} \in \mathcal{R}_{\sigma \cdot [C/X]B} = \mathcal{R}_{(\sigma B)[C/X]}$. We have then that $\{[\![B]\!]_{\rho \cdot [F/X]} \mid F \in \mathcal{R}_C\} \subseteq \mathcal{R}_{(\sigma B)[C/X]}$, so, by Definition 6.2, $\bigcap\{[\![B]\!]_{\rho \cdot [F/X]} \mid F \in \mathcal{R}_C\} \subseteq \mathcal{R}_{(\sigma B)[C/X]}$. Then, again by Definition 6.2, since previous statement holds for any type C, we have $\tilde{\forall} C. \bigcap\{[\![B]\!]_{\rho \cdot [F/X]} \mid F \in \mathcal{R}_C\} \in \mathcal{R}_{\forall X.\sigma B} = \mathcal{R}_{\sigma(\forall X.B)}$. □

   We proved, for all types B isomorphic to A, that the set formed by IH (*e.g.*$[\![B_1]\!]_\rho$, $[\![B_2]\!]_\rho$) and the candidate constructor (*e.g.*$\tilde{\to}$) of its main type constructor (*e.g.*$\to$), is a candidate of $\mathcal{R}_{\sigma B}$. Let us call that set $\mathcal{B}$. To show that $[\![A]\!]_\rho$ is indeed a candidate of $\mathcal{R}_{\sigma A}$, we can apply the previous reasoning to A, since $\equiv$ is reflexive, obtaining the candidate $\mathcal{A}$ of $\mathcal{R}_{\sigma A}$. If we take $\mathcal{A} \cap \mathcal{B}$, we get a candidate of $\mathcal{R}_{\sigma A}$ by the isomorphic candidates rule. If we repeatedly apply the rule for all $\mathcal{C}$ corresponding to each C isomorphic to A, we get $\mathcal{A} \cap \mathcal{C}_1 \cap \ldots \cap \mathcal{C}_n$. That set is precisely $[\![A]\!]_\rho$, so we conclude.

Lemma 6.10 (Substitution). $[\![A[B/X]]\!]_\rho = [\![A]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$

Proof. By induction on $\mathsf{m}(A)$. For the base case, we have A equal to a type variable, and we consider two possibilities, whether it coincides with X or not.

- If $A \equiv X$, then $[\![X[B/X]]\!]_\rho = [\![B]\!]_\rho = \rho \cdot [[\![B]\!]_\rho / X](X) = [\![X]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$.
- If $A \equiv Y$, then $[\![Y[B/X]]\!]_\rho = [\![Y]\!]_\rho = \rho Y = \rho \cdot [[\![B]\!]_\rho / X] Y = [\![Y]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$.

For the inductive case, $[\![A]\!]_\rho$ is built considering all possible types equivalent to A. So we check back and forth that, in all these cases, the equality holds. Let $t \in [\![A[B/X]]\!]_\rho$.

- If $A \equiv C \times D$, then $t \in [\![C[B/X]]\!]_\rho \tilde{\times} [\![D[B/X]]\!]_\rho$. By IH, $t \in [\![C]\!]_{\rho \cdot [[\![B]\!]_\rho / X]} \tilde{\times} [\![D]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$.
- If $A \equiv C \to D$, then $t \in [\![C[B/X]]\!]_\rho \tilde{\to} [\![D[B/X]]\!]_\rho$. By IH, $t \in [\![C]\!]_{\rho \cdot [[\![B]\!]_\rho / X]} \tilde{\to} [\![D]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$.
- if $A \equiv \forall Y.C$, then $t \in \tilde{\forall} D. \bigcap\{[\![C[B/X]]\!]_{\rho \cdot [F/Y]} | F \in \mathcal{R}_D\}$. By IH, $t \in \tilde{\forall} D. \bigcap\{[\![C]\!]_{\rho \cdot [F/Y] \cdot [[\![B]\!]_{\rho \cdot [F/Y]} / X]} | F \in \mathcal{R}_D\}$

Then, by Definition 6.6, $t \in [\![A]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$, so $[\![A[B/X]]\!]_\rho \subseteq [\![A]\!]_{\rho \cdot [[\![B]\!]_\rho / X]}$. The symmetric inclusion is proven similarly. □

A key part of the Adequacy lemma is the recursive analysis of one–step reducts of terms, proving that all of them are $\mathcal{SN}_*$, in order to prove that the common anti–reduct is also $\mathcal{SN}_*$. Although this is not a particularly complex part, the amount of equivalences that arise on terms produces several new one–step reducts for each term constructor, so the analysis has a small combinatorial explosion. The most essential task here is to understand what are all the possible one–step reducts of all term constructors. To do so, we first characterize the class of equivalence of term constructors. For instance, to which other terms can a type abstraction be equivalent.

Lemma 6.11 (The class of type abstractions). *If $\Lambda X.t' \rightleftarrows^n s$, then s is equal to:*

1. *$\Lambda X.s'$ with $t' \rightleftarrows^m s'$ and $m \leq n$*
2. *$\lambda x^A.s'$ with $t' \rightleftarrows^{m_1} \lambda x^A.r$, $s' \rightleftarrows^{m_2} \Lambda X.r$, $m_1 + 1 + m_2 \leq n$, and $X \notin FV(A)$*
3. *$\langle s'_1, s'_2 \rangle$ with $t' \rightleftarrows^{m_1} \langle r_1, r_2 \rangle$, $s'_i \rightleftarrows^{m_{2_i}} \Lambda X.r_i$, and $m_1 + 1 + m_{2_1} + m_{2_2} \leq n$*
4. *$\pi_{\forall X.A} s'$ with $t' \rightleftarrows^{m_1} \pi_A r$, $s' \rightleftarrows^{m_2} \Lambda X.r$, and $m_1 + 1 + m_2 \leq n$*

Proof. By induction on the equivalence derivation $\Lambda X.t' \rightleftarrows^n s$. □

The characterization lemmas for the equivalence class of the other term constructors are a work in progress. Once we have them, we will be able to characterize the possible one–step reducts of all terms. These will enable us to reproduce the $\mathcal{SN}$ closure under head expansion lemma in Polymorphic System I, and finally to proceed with the Adequacy lemma.

WIP Lemma 6.12 (Adequacy). *If $\Gamma \vdash t : A$ and $\theta, \sigma, \rho \vDash \Gamma$, then $\theta(\sigma(t)) \in [\![A]\!]_\rho$.*

WIP Theorem 6.13 (Strong normalization). *If $\Gamma \vdash t : A$, then $t \in \mathcal{SN}$.*

References

[1] Beniamino Accattoli and Alejandro Díaz-Caro. 2020. Functional Pearl: The Distributive λ-Calculus. In *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12073)*, Keisuke Nakano and Konstantinos Sagonas (Eds.). Springer, 33–49. https://doi.org/10.1007/978-3-030-59025-3_3

[2] Roberto Di Cosmo. 1995. *Isomorphisms of types: from λ-calculus to information retrieval and language design.* Birkhauser, Switzerland.

[3] Alejandro Díaz-Caro and Gilles Dowek. 2019. Proof Normalisation in a Logic Identifying Isomorphic Propositions. *LIPIcs (FSCD'19)* 131 (2019), 14:1–14:23.

[4] Alejandro Díaz-Caro and Gilles Dowek. 2023. Extensional proofs in a propositional logic modulo isomorphisms. *Theor. Comput. Sci.* 977 (2023), 114172. https://doi.org/10.1016/J.TCS.2023.114172

[5] Alejandro Díaz-Caro and Pablo E. Martínez López. 2015. Isomorphisms considered as equalities: Projecting functions and enhancing partial application through an implementation of λ^+. *ACM IFL* 2015, 9 (2015), 1–11.

[6] Jean-Yves Girard. 1971. Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types. In *Proceedings of the Second Scandinavian Logic Symposium*, J.E. Fenstad (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63–92. https://doi.org/10.1016/S0049-237X(08)70843-7

[7] Jean-Yves Girard. 1972. Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur. https://api.semanticscholar.org/CorpusID:117631778

[8] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Cambridge U.P., UK.

[9] Michel Parigot. 1997. Proofs of Strong Normalisation for Second Order Classical Natural Deduction. *J. Symb. Log.* 62, 4 (1997), 1461–1479. https://doi.org/10.2307/2275652

[10] Cristian Sottile, Alejandro Díaz-Caro, and Pablo E. Martínez López. 2020. Polymorphic System I. In *IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020*, Olaf Chitil (Ed.). ACM, 127–137. https://doi.org/10.1145/3462172.3462198

[11] William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

Foundations of Gradual Abstract Interpretation (draft paper)

ANONYMOUS AUTHOR(S)

Static program analyses are valuable tools for testing and lightweight verification of software, offering automated predictions of program behavior. Soundness is desirable, but non-trivial program properties are undecidable, so sound analyses must rely on over-approximation. This often leads to imprecision, such as false alarms or unnecessary rejections of valid programs, which limits the usability of sound analyses in practice. To mitigate this problem, some authors have proposed *soundiness*: analysis designers can deliberately under-approximate certain features, provided that assumptions are made explicit. However, soundiness alone does not ensure that assumptions hold at runtime, leaving results potentially invalid. We explore an alternative approach based on gradual techniques: optimistic assumptions are combined with runtime checks, producing not only analysis results but also elaborated programs that detect violations dynamically. This gradual style allows analyses to recover precision without abandoning reliability. We explore how the framework of abstract interpretation can be infused with this gradual approach to produce gradual analyses. We develop the foundations of Gradual Abstract Interpretation (GAI) and formalize its metatheory for a small language in Rocq.

1 Introduction

Static program analyses are important tools in software development. They help in both testing and lightweight verification of programs. Their push-button nature is attractive: they can automatically report the presence of bugs or check that a program satisfies a given property without requiring user interaction.

A central concern for such tools is soundness. A sound analysis guarantees that no bugs are missed and that no program is incorrectly verified as satisfying a given property. Soundness is desirable because an unsound tool can mislead developers, either by overlooking errors or by validating programs that do not behave as prescribed by their specification.

The main framework for designing sound static analyses is abstract interpretation [Cousot and Cousot 1977]. This framework provides a systematic method for approximating program behavior in a way that preserves soundness. However, by Rice's theorem [Rice 1953], every non-trivial semantic property of programs is undecidable. Therefore, analyses cannot be both precise and sound. If soundness is required, program behavior must be over-approximated. Over-approximation ensures that all possible traces of execution of a program are considered to make a prediction. This might include paths that are actually impossible, introducing imprecision. An analysis that is too imprecise becomes impractical. For instance, if it reports too many false alarms or rejects too many valid programs, developers may start to disregard the alarms and error reports.

In practice, several sources make over-approximation particularly troublesome. Some low-level details *leak* into the analysis, such as the overflow behavior of machine integers. Certain language features are also difficult to handle statically, for example dynamic evaluation, reflection, or user input. Moreover, programs often depend on external libraries or APIs for which precise models are not available.

To address these difficulties, the notion of *soundiness* has been proposed by Livshits et al. [2015]. In soundy analyses, designers intentionally under-approximate some features in order to improve precision. This approach accepts a degree of unsoundness but requires analysis designers to state explicitly which features are under-approximated. The drawback is that soundiness itself does not guarantee that these assumptions are respected during program execution. Which means that users might act on invalid results if they misuse the tool.

In this work, we address the problem in a *gradual* style, taking inspiration from the principles of *gradual typing* [Siek and Taha 2006]. The approach relies on optimistic under-approximations, which are validated through the insertion of runtime checks that dynamically verify whether the assumptions hold. In this way, the output of a gradual analysis is not just an analysis result but also an elaborated program. The instrumented program contains runtime checks that raise errors as soon as an assumption is violated, ensuring that the analysis results are sound approximations of the elaborated program's behavior.

Contributions. With this work, we make the following contributions:

- We explore how the framework of abstract interpretation can be infused with the ideas that come from gradual typing to design *gradual analyses*.
- We develop the foundations of Gradual Abstract Interpretation (GAI) and study its metatheory.
- We provide a mechanized formalization in Rocq of a proof of concept for a small language.

Outline. The rest of the paper is organized as follows. In §2 we illustrate the essence of this work with a running example. The small language that we consider for our development is defined in §3, where we also present a sound static analysis for it. In §4 we present *soundiness* in more detail, discussing how we could use that approach to increase precision of the static analysis and ponder its problems. The foundations of GAI are introduced in §5, where we start with an overview of the technical aspects of the approach (§5.1), followed by the presentation of the formalization (§5.2) and the development of the metatheory (§5.3). In §6 we discuss how we envision GAI to be used in practice, as well as future extensions to this work. Finally, we conclude in §8.

2 Running Example

To illustrate the essence of this work, let us consider the following program, where we define a recursive function which computes the sum of numbers from 1 to its argument, and then apply it to 4.

```
0 def sum(n):
1   if (n <= 1):
2     return 1
3   else:
4     return n + sum(n - 1)
5
6 sum(4)
```

Suppose we wish to do a sign analysis on this program. That is, predict the sign of the values that the program will work with at runtime. We can achieve this with an abstract interpreter which uses the lattice shown in Fig. 1 as its abstract domain.

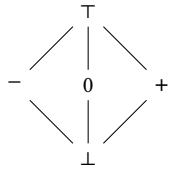


Fig. 1. Hasse diagram of the simple sign lattice.

For the sake of illustration, we will also suppose that the language has bounded integers. This means that arithmetic operations might overflow. Consequently, the addition of two positive numbers could result in a negative value. For the analysis to be sound, the abstract addition used by the abstract interpreter must satisfy $+ \oplus + = \top$. In other words, adding two positive numbers could result in a value of any sign due to potential overflows.

In the initial steps of abstract interpretation, the abstract result of the call `sum(n - 1)` in line 4 will be +. If the interpreter is path sensitive, it could even refine its information to know that `n` is positive in the else branch[1]. However, the result of the addition will be $\top$ nonetheless. As a consequence, once the interpretation stabilizes, the conclusion will be that the call `sum(4)` in line 6 can return a value of any sign (i.e. $\top$).

This imprecision is undesirable, as in this context we know that an overflow will not occur and that the call will return a positive result. In fact, we could be using this analysis during the development of a system in a particular domain where we will always work with small integers. Therefore, the precision of the analysis will suffer unnecessarily. What can we do to remedy this?

We could follow the *soundy* approach of [Livshits et al. 2015] and design our analysis assuming that no overflows will occur. In the spirit of *soundiness*, we would need to be explicit about this assumption and inform the users of the analysis about it. Our newfound optimism would allow us, in particular, to under-approximate the behavior

[1]Path sensitivity is intractable in the general case. However, for simple guard conditions such as `n <= 1`, the fact that `n` is strictly positive in the else branch can indeed be derived automatically.

of the addition. Therefore, the abstract operation would be such that $+ \oplus + = +$. This is enough to recover precision in our previous example, and the abstract interpreter will conclude that the call `sum(4)` returns a positive value.

The problem with this approach is that, although we have warned the users about our assumption, the analysis provides no mechanism to verify that the assumption is respected. Consequently, users are left to fend for themselves, and they could end up acting on incorrect results produced by an accidental misuse of the analysis.

An alternative route to address imprecision, inspired by gradual typing [Siek and Taha 2006; Siek et al. 2015], is to insert runtime checks in the points where the analysis leverages optimistic assumptions. In our example program, the assumption is used in line 4. Therefore, a runtime check would be inserted to guard the addition, yielding an elaborated program like the following, where $\mathsf{check}_{>0}(\cdot)$[2] raises an error if its argument is non-positive.

```
0  def sum(n):
1    if (n <= 1):
2      return 1
3    else:
4      return check>0(n + sum(n - 1))
5
6  sum(4)
```

A *gradual* analysis would, therefore, not only give a more precise result, but also an elaborated program whose runtime behavior is faithful to the prediction. The users of such an analysis can now test the elaborated program in realistic scenarios to make sure that the assumptions are being respected, getting a runtime error as soon as possible if that is not the case. This gives them the opportunity and valuable information to correct the original program.

3 Base Language & Static Analysis

In this section we define a simple language and a sound abstract interpreter for it, which we also refer to as *static analysis*. Both of these are the base for our study of the foundations of Gradual Abstract Interpretation (GAI). All the definitions, theorems and notable results presented in this paper have been formalized in the Rocq prover.

The base language, defined in Fig. 2, contains integer and boolean values, identifiers, addition and $<$ comparison for integers, and an `if` expression. In our formalization, integers are modeled using the abstractions of fixed-length binary integers provided by CompCert [Leroy 2009], which correctly model the overflowing behavior of arithmetic operations. We also define concrete contexts (ρ) as maps from identifiers to values, and sets of contexts ($\bar{\rho}$).

The concrete semantics of the language is defined by the big-step reduction relation $[\![e]\!]\ \rho \mapsto v$. As standard practice in abstract interpretation, the collecting semantics $[\![e]\!]_C\ \bar{\rho} \mapsto \bar{v}$ is defined for sets of concrete contexts, and its purpose is to *collect* the values which an expression reduces to in each context of the set. Fig. 2 shows some of the rules that define these relations.

The abstract domain used by our sound abstract interpreter is illustrated in Fig. 3. It is an extension of the lattice presented in §2, and includes new abstract values to represent any integer ($\mathbb{Z}$) and booleans ($\mathbb{B}$). Thus, the $\top$ element now represents any value, integer or boolean. We will refer to this lattice as L, and use $\sqsubseteq$, $\sqcup$ and $\sqcap$ to represent its partial order relation, join and meet operations, respectively.

To complete the abstract domain, we define abstract contexts (σ) as maps from variables to abstract values. The relations and operations of L extend naturally to abstract contexts through pointwise definitions, and we will overload the symbols to avoid verbosity.

The connection between the concrete and abstract domains is established through the abstraction functions α, $\alpha_{\bar{v}}$, α_ρ and $\alpha_{\bar{\rho}}$, defined for values, sets of values, contexts and sets of contexts respectively, as illustrated also in Fig. 3. Note that both α_ρ and $\alpha_{\bar{\rho}}$ produce a single abstract context as a result.

With the previous definitions in place, we are ready to introduce the abstract interpreter. Some of its rules are illustrated in Fig. 4. The most relevant for our discussion is AAdd, which shows that the abstract interpreter handles the addition with the sound abstract addition $\oplus_s$. As

[2] For the addition of two strictly positive numbers, this check is enough to guarantee that no overflow occurred, as an overflow cannot cause the result to «loop around» yielding a strictly positive result.

$$
\begin{array}{rcl}
v & ::= & z \in \mathbb{Z} \mid b \in \mathbb{B} \\
e & ::= & v \mid x \in \textsc{Var} \mid e + e \mid e < e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e
\end{array}
$$

$$
\begin{array}{rcl}
\rho & \in & \textsc{Ctx} = \textsc{Var} \rightarrow \textsc{Val} \\
\bar{\rho} & \in & \mathcal{P}(\textsc{Ctx}) \\
\bar{v} & \in & \mathcal{P}(Val)
\end{array}
$$

$\boxed{[\![e]\!]\ \rho \mapsto v}$

$$
\ldots \qquad \frac{[\![e_1]\!]\ \rho \mapsto z_1 \qquad [\![e_2]\!]\ \rho \mapsto z_2}{[\![e_1 + e_2]\!]\ \rho \mapsto z_1 + z_2}\ \textsc{EAdd}
$$

$\boxed{[\![e]\!]_C\ \bar{\rho} \mapsto \bar{v}}$

$$
\ldots \qquad \frac{[\![e]\!]\ \rho \mapsto v \qquad [\![e]\!]_C\ \bar{\rho} \mapsto \bar{v}}{[\![e]\!]_C\ \{\rho\} \cup \bar{\rho} \mapsto \{v\} \cup \bar{v}}\ \textsc{CollectCons}
$$

Fig. 2. Definition of base language: syntax, semantic domains and concrete semantics. The language features integer and boolean values, identifiers, addition and < comparison for integers, and an `if` expression. For illustration, we show some of the rules that define the relations of the concrete semantics.

discussed in §2, this operation accounts for potential overflows, returning $\mathbb{Z}$ (any integer) when both operands are positive or both are negative. Its definition is also extended to work with the new abstract domain. For instance, when any of the operands of $\oplus_s$ is a boolean, it returns $\perp$ to signal an invalid addition.

We establish the soundness of this static analysis in Theorem 3.1, which states that the result returned by the abstract interpreter for an expression e is a sound approximation of the values which e can reduce to. The theorem is illustrated pictorially in Fig. 5.

Theorem 3.1 (Soundness of Static Analysis). *If* $[\![e]\!]_C\ \bar{\rho} \mapsto \bar{v}$ *and* $[\![e]\!]_a\ \alpha_{\bar{\rho}}(\bar{\rho}) \mapsto a$ *then* $\alpha_{\bar{v}}(\bar{v}) \sqsubseteq a$.

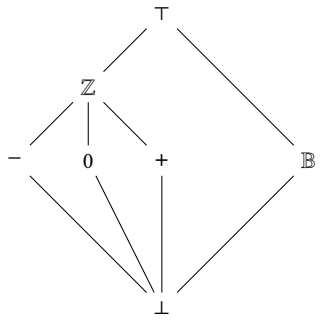
This formal result is desirable, however, as illustrated in §2, it comes at the cost of the precision of the abstract interpreter. As a consequence of the definition of the sound abstract addition $\oplus_s$, even for a simple expression such as `2 + 3`, the static analysis returns $\mathbb{Z}$, where we would want the result to be +.

4 Soundiness

One way of increasing the precision of the static analysis presented in the previous section is to follow the *soundy* approach [Livshits et al. 2015]. As introduced in §2, a soundy analysis increases precision by under-approximating certain features. In other words, these kind of analyses *optimistically assume* that some features are used in a specific way and/or that some behaviors do not occur at runtime.

For example, we could turn the static analysis from §3 into a soundy analysis by switching the sound abstract addition $\oplus_s$ with an optimistic version $\oplus_o$ which optimistically assumes that no overflow can occur. For instance, this operation would say that adding any two positive integers results in a positive value, i.e. $+ \oplus_o + = +$.

Soundy analyses are characterized by what is referred to as a *sound core*. This is the fragment of the language whose behavior is over-approximated by the analysis; in other words, the subset of programs for which the soundy analysis returns sound results. In contrast, the soundy analysis



$$\sigma \in \textsc{AbsCtx} = \textsc{Var} \to L$$

$$\alpha : \textsc{Val} \to L$$

$$\alpha(v) = \begin{cases} \mathbb{B} & , v \in \mathbb{B} \\ sign(v) & , v \in \mathbb{Z} \end{cases}$$

$$\alpha_{\bar{v}} : \mathcal{P}(\textsc{Val}) \to L$$

$$\alpha_{\bar{v}}(\bar{v}) = \bigsqcup_{v \in \bar{v}} \alpha(v)$$

$$\alpha_\rho : \textsc{Ctx} \to \textsc{AbsCtx}$$

$$\alpha_\rho(\rho) = \{x \mapsto a \mid \alpha(\rho(x)), x \in \textsc{Dom}(\rho)\} \cup \{x \mapsto \bot \mid x \notin \textsc{Dom}(\rho)\}$$

$$\alpha_{\bar{\rho}} : \mathcal{P}(\textsc{Ctx}) \to \textsc{AbsCtx}$$

$$\alpha_{\bar{\rho}}(\bar{\rho}) = \bigsqcup_{\rho \in \bar{\rho}} \rho$$

Fig. 3. Definition of the abstract domain as the extended sign lattice L which includes boolean values. Abstract contexts are defined as maps from variables to abstract values of L. The connection between the concrete and abstract domains is defined by the abstraction functions α, α_ρ and $\alpha_{\bar{\rho}}$.

could return either sound or unsound results for programs that lie outside of the sound core.

Analysis designers ought to declare which features are not part of the sound core, and describe how they are under-approximated, i.e. describe what assumptions are made about the behavior of those features. For the soundy version of our analysis, we would declare that the addition is under-approximated by assuming that it does not overflow. Our responsibility as analysis designers would end

$$\boxed{[\![e]\!]_a\ \sigma \mapsto a}$$

$$\frac{[\![e_1]\!]_a\ \sigma \mapsto a_1 \qquad [\![e_2]\!]_a\ \sigma \mapsto a_2}{[\![e_1 + e_2]\!]_a\ \sigma \mapsto a_1 \oplus_s a_2}\ \textsc{AAdd}$$

$$\frac{[\![e_t]\!]_a\ \sigma \mapsto a_t \qquad [\![e_f]\!]_a\ \sigma \mapsto a_f}{[\![\texttt{if}\ c\ \texttt{then}\ e_t\ \texttt{else}\ e_f]\!]_a\ \sigma \mapsto a_t \sqcup a_f}\ \textsc{AIf}$$

Fig. 4. Relevant rules of the abstract interpreter which uses lattice L as the abstract domain. Interpretation of the addition is performed using the sound abstract addition $\oplus_s$. For the `if`, the interpreter joins the results of both branches.

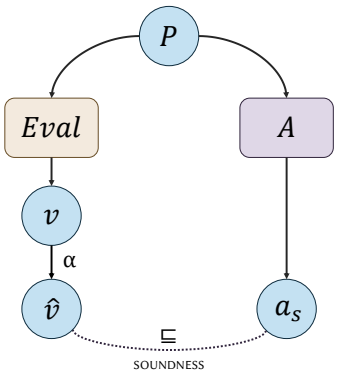


Fig. 5. The results of the abstract interpreter soundly approximate the values that a program reduces to.

there. The users of the analysis need to pay attention to how they use the feature so that they can assess whether their program is in the sound core or not, so that they can correctly interpret the results of the analysis. However, determining whether a program satisfies the assumptions of the soundy analysis might not be straightforward, and an incorrect appraisal of the situation might lead a user to trust invalid results.

In practical analyses, under-approximations are justified by domain knowledge. They reflect the way in which features are actually used. For example, in a study by Richards et al. [2011], the authors show that the `eval` construct in JavaScript, which allows dynamic execution of arbitrary code, is employed in very specific patterns by expert users.

As discussed, this kind of domain knowledge can be levaraged to increase the precision of analyses. But it can also guide us on how to check whether the under-approximated features are being used in a way that satisfies the optimistic assumptions. For example, when computing an addition of two positive numbers, we can check whether an overflow occurred by testing if the result was positive.

Once we notice that, we can approach the design of analyses using the *gradual* approach. That is, whenever the analysis treats a feature optimistically, it inserts a runtime check so that the assumption can be validated during execution. Such a *gradual analysis* would not only return a more precise prediction, but also an elaborated program that the user can run in realistic scenarios to test whether the original program actually satisfies the optimistic assumptions leveraged by the analysis.

In the next section, we explore how we can infuse the theory of abstract interpretation with the techniques that come from gradual typing, in order to produce gradual analyses such as the one described above.

5 Gradual Abstract Interpretation (GAI)

In this section we develop the theory of the gradual approach to program analysis based on abstract interpretation. All aspects of the formalization have been mechanized in the Rocq prover.

5.1 Approach

As illustrated in § 2, the essence of GAI is the insertion of runtime checks to validate the optimistic assumptions leveraged by an abstract interpreter to increase the precision of its results. Thus, a gradual abstract interpreter (also referred to as gradual analysis), yields an elaborated program along with its result.

The way we introduce optimism in an abstract interpreter is quite simple. Each language construct that is going to be handled gradually has two corresponding abstract operations: a sound one and an optimistic one. With these, the gradual analysis can detect an opportunity to be optimistic by performing both operations and comparing the sound result a_s with the optimistic one a_o. If $a_s \not\sqsubseteq a_o$, that is, when the optimistic result is strictly more precise than (or unrelated to) the sound one, then the analysis can be optimistic. This condition also signals the need for a runtime check. If not, the analysis simply returns the sound result and does not insert any check.

The final result is obtained as the precision meet $a_s \sqcap a_o$, thereby ensuring the most precise under-approximation. If the sound and optimistic results are unrelated but have a non-empty intersection, the meet yields the intersection. And if the intersection is empty, the meet results in $\bot$, which lets us know that the check will always fail. Simply returning the optimistic result a_o would miss those cases.

To illustrate the approach, let us see how the gradual version of the static analysis of § 3 would operate. Firstly, it would have both a sound abstract addition $\oplus_s$ and an optimistic one $\oplus_o$, such as the one discussed in § 4. When analyzing the addition of two positive numbers, the analysis notes that $+ \oplus_s + = \mathbb{Z} \not\sqsubseteq + = + \oplus_o +$. Therefore, the final result is $\mathbb{Z} \sqcap + = +$ and, because the analysis is being optimistic, it inserts a runtime check in the elaborated program, in order to validate the assumption during execution.

5.2 Formalization

We begin the formalization of GAI by extending the base language presented in § 3 with a construct called `assume`. This expression is an abstract representation of a runtime check. An `assume` takes a subject expression e and an abstract value a, which we refer to as the *assumption*. During execution, e is reduced and if the abstraction of the resulting value is at least as precise as the assumption, the execution continues with the computed value. Otherwise, execution terminates with an error. For simplicity, errors are represented by the special value $\bot$. These extensions to the base language and reduction are presented in Fig. 6.

We can also extend the static analysis presented in § 3 to handle the `assume` expression. As presented in the rule AAssume, and consistent with the discussion in § 5.1, the analysis uses the meet operation to combine the result of

[3] To simplify the presentation, we omit the propagation rules for errors. They are defined as expected, i.e. when any of the operators of an operation is an error, the operation results in an error. Likewise, when the condition of an `if` is an error, the expression reduces to an error (no branch is evaluated).

$$
\begin{array}{rcl}
v & ::= & \ldots \mid \bot \\
e & ::= & \ldots \mid \texttt{assume}\ e\ a
\end{array}
$$

$$\frac{[\![e]\!]\ \rho \mapsto v \qquad \alpha\ v \sqsubseteq a}{[\![\texttt{assume}\ e\ a]\!]\ \rho \mapsto v}\ \textsc{EAssumePass}$$

$$\frac{[\![e]\!]\ \rho \mapsto v \qquad \alpha\ v \not\sqsubseteq a}{[\![\texttt{assume}\ e\ a]\!]\ \rho \mapsto \bot}\ \textsc{EAssumeFail}$$

Fig. 6. Base language and reduction rules extended with the `assume` construct[3].

$$\frac{[\![e]\!]_a\ \sigma \mapsto a_e}{[\![\texttt{assume}\ e\ a]\!]_a\ \sigma \mapsto a_e \sqcap a}\ \textsc{AAssume}$$

Fig. 7. Static analysis rules extended for the `assume` construct.

the analysis for the subject expression and the assumption. This handling preserves soundness of the static analysis as stated in Theorem 3.1.

With the presented extensions, we are ready to define the gradual abstract interpreter. We write $\sigma \vdash s \rightsquigarrow e : a$ to mean that, when a gradual abstract interpreter analyze a source expression s in the abstract context σ, it returns the elaborated expression e and the result a. Both the source expression s and the elaborated one e are terms of the same extended language, but we use different metavariables to make it evident in text when we are talking about one kind of expression or the other.

The structure of the rules of the static analysis are preserved by the gradual analysis except for the addition, where we now have two rules (GAddSafe and GAddOptimistic) to detect an opportunity for optimism as described in § 5.1. The sound abstract addition used in these rules ($\oplus_s$) is the same operation used by the static analysis in AAdd.

The gradual analysis is parameterized by a set of optimistic abstract operations, *ops*, yielding the notation $\sigma \vdash \langle ops \rangle\ s \rightsquigarrow e : a$. Likewise, the optimistic abstract addition is written $\oplus_o^{ops}$. This parameterization allows us to further fine-tune the precision of the gradual analysis, and also provides a means to decouple the theory from the actual implementation of the optimistic operations. Consequently, it allows us to study the formal requirements for those optimistic operations.

As shown in rule GAddOptimistic, when optimism is applied, the analysis inserts an `assume`, representing the runtime check. The assumption that is checked during execution is the same abstract value that the analysis returns as a result ($a_s \sqcap a_o$). This ensures that this result soundly approximates the reduction of the elaborated expression.

5.3 Metatheory

One of the objectives of a gradual analysis is to increase the precision of the results by leveraging optimistic assumptions. As we noted in the previous section, the gradual analysis preserves the structure of the rules of the static analysis, modulo the gradual handling of the addition. This means that the gradual analysis produces the same results as the static analysis for all expressions except for the addition. However, in this case there are two options. Either the gradual analysis proceeds optimistically, returning a result that is more precise than the one that the static one returns ($a_s \sqcap a_o \sqsubseteq a_s$), or it uses no optimism, returning the same result as the static analysis. Therefore, irrespective of the route that the gradual analysis takes, its results will be at least as precise as the results of the static one. And, in some cases, it yields results that are indeed more precise. We establish this in Theorem 5.1.

Theorem 5.1 (Gradual Analysis is More Precise). *If* $\sigma \vdash s \rightsquigarrow e : a_g$ *and* $[\![s]\!]_a\ \sigma \mapsto a_s$ *then* $a_g \sqsubseteq a_s$.

Note that whenever we use the notation that leaves the set of optimistic operations implicit, it means that the property in question is quantified for all possible sets of operations. Therefore, the formal result is valid for all instantiations of the gradual analysis.

Another interesting property of the gradual analysis is that it is idempotent. That is, if the analysis of a source expression yields a result a and an elaborated expression e, then analyzing e returns the same result a. In other words, the gradual analysis cannot apply further optimism. This property is interesting because when the gradual analysis

$$\boxed{\sigma \vdash \langle ops \rangle\ s \rightsquigarrow e : a}$$

$$\frac{\sigma \vdash \langle ops \rangle\ s_1 \rightsquigarrow e_1 : a_1 \qquad \sigma \vdash \langle ops \rangle\ s_2 \rightsquigarrow e_2 : a_2 \qquad a_1 \oplus_s a_2 \sqsubseteq a_1 \oplus_o^{ops} a_2}{\sigma \vdash \langle ops \rangle\ s_1 + s_2 \rightsquigarrow e_1 + e_2 : a_1 \oplus_s a_2}\ \textsc{GAddSafe}$$

$$\frac{\sigma \vdash \langle ops \rangle\ s_1 \rightsquigarrow e_1 : a_1 \qquad \sigma \vdash \langle ops \rangle\ s_2 \rightsquigarrow e_2 : a_2 \qquad a_1 \oplus_s a_2 \not\sqsubseteq a_1 \oplus_o^{ops} a_2}{\begin{array}{c}\sigma \vdash \langle ops \rangle\ s_1 + s_2 \rightsquigarrow \texttt{assume}\ \ (e_1 + e_2)\ (a_s \sqcap a_o) : a_s \sqcap a_o \\ \text{where } a_s = a_1 \oplus_s a_2 \text{ and } a_o = a_1 \oplus_o^{ops} a_2\end{array}}\ \textsc{GAddOptimistic}$$

Fig. 8. Relevant rules of the gradual analysis. The addition is handled by two rules to detect an oportunity for optimism.

cannot be optimistic, it returns the same results as the static analysis.

As a consequence, analyzing an elaborated expression with the static analysis returns the same result that the gradual analysis returns when analyzing the corresponding source expression. This is stated in Lemma 5.2.

Lemma 5.2 (Static Analysis on Elaborated Program). *If* $\sigma \vdash s \rightsquigarrow e : a$, *then* $[\![e]\!]_a\ \sigma \mapsto a$

With this lemma, we can appeal to the soundness of the static analysis to establish the notion of *gradual soundness* (Theorem 5.3), which states that the results of the gradual analysis are sound approximations of the behavior of the elaborated expressions.

Theorem 5.3 (Gradual Soundness). *If* $\alpha_{\bar{\rho}}(\bar{\rho}) \vdash s \rightsquigarrow e : a$ *and* $[\![e]\!]_C\ \bar{\rho} \mapsto \bar{v}$ *then* $\alpha_{\bar{v}}(\bar{v}) \sqsubseteq a$.

All the previous connections between the gradual and static analyses are depicted in Fig. 9, along with the results stated by Theorem 5.1 and Theorem 5.3.

Respecting the expectations of users. In gradual typing, one of the central concerns is to allow users to navigate the static-to-dynamic spectrum without breaking their expectations regarding typing and reduction. This is captured by the gradual guarantees [Siek et al. 2015]. Similar considerations also apply in the context of GAI, with some nuance, in three dimensions, as explained hereafter.

Error-Approximation of Source Expressions. Firstly, the elaboration needs to preserve the reduction behavior of the

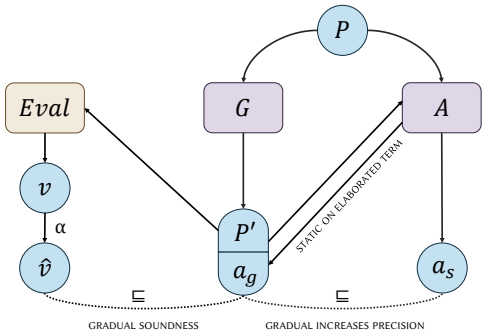


Fig. 9. Relations between the gradual and static analyses. Analyzing an elaborated program with the static analysis yields the same prediction that the gradual analysis returned for the corresponding source program. The results of the gradual analysis are sound approximations of the behavior of elaborated expressions, and are at least as precise as the results of the static analysis.

source expressions. In other words, if the source expression reduces to a value, then the elaborated expression must also reduce to a related value or fail with an error raised by a runtime check. This property is inspired by the dynamic gradual guarantee (DGG) of gradual typing, and we call it *error approximation* (following New and Ahmed [2018]).

In our context, this property is established from the well-formedness of the elaboration, defined by the relation $\textsc{ElabWF}_\sigma\ s\ e$, whose presentation we omit for brevity. This relation essentially requires the elaboration to preserve the syntactic structure of the source expression s, and only allows it to introduce `assumes` in e. We prove that the elaborated expressions produced by the gradual analysis

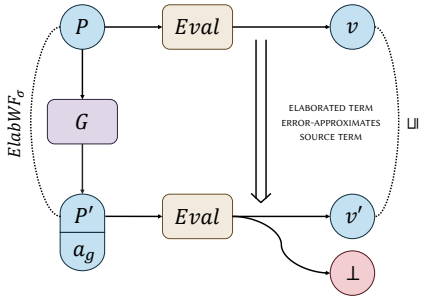


Fig. 10. Well-formedness of the elaboration and error-approximation of the source term by the elaborated term.

are well-formed with respect to their source expressions (Theorem 5.4).

Theorem 5.4 (Elaboration of Gradual Analysis is Well-Formed). *If* $\sigma \vdash s \rightsquigarrow e : a$ *then* $\textsc{ElabWF}_\sigma\ s\ e$.

We also prove that, whenever $\textsc{ElabWF}_\sigma\ s\ e$ holds, e error-approximates s (Theorem 5.5).

Theorem 5.5 (Elaborated Expression Error-Approximates Source). *If* $\textsc{ElabWF}_{\alpha_\rho(\rho)}\ s\ e$ *then,*

(1) *if* $[\![s]\!]\ \rho \mapsto v_s$ *then* $[\![e]\!]\ \rho \mapsto v_e$ *and* $v_e \sqsubseteq v_s$.
(2) *if* $[\![e]\!]\ \rho \mapsto v_e$ *and* $v_e \neq \bot$ *then* $[\![s]\!]\ \rho \mapsto v_s$ *and* $v_e \sqsubseteq v_s$.

The second part of Theorem 5.5 is important, because it ensures that if a user runs an elaborated program in a given context and the program reduces without failing, then the corresponding source program can be run safely in the same context, reducing correctly. In other words, it verifies that in that context, no assumptions were violated, and the results given by the gradual analysis are also sound approximations of the behavior of the source program (without runtime checks). The previous results are depicted in Fig. 10.

Monotonicity with Respect to Expression Precision. The second dimension in which a gradual analysis honors the expectations of users is with respect to the precision of the analyzed programs. For this, we define a precision relation for expressions. We say that $\sigma_1,\ \sigma_2 \vdash e_1 \preccurlyeq e_2$ holds when both expressions are syntactically equal modulo the pressence of `assumes`, as handled by the rules shown in Fig. 11. The relation is indexed by two abstract contexts, σ_1 and σ_2, to allow relating expressions that have the same identifiers bound to related values.

In rule LTEEAssume, we relate two `assumes` expressions when their subjects and assumptions are related by precision.

Notable are the cases where we try to relate expressions with asymmetric presence of `assumes`. When an `assume` appears on the left expression, the premise in rule LTEEAssumeL tells us (by inductive reasoning) that the analysis result and reduction for e_1 are more precise than the analysis result and reduction of e_2. And, since an `assume` can only increase precision, that sole premise is enough to establish that surrounding e_1 with an `assume` is more precise than e_2.

The rule for the opposite case (LTEEAssumeR) has more requirements. Since an `assume` can increase precision, the premise $\sigma_1,\ \sigma_2 \vdash e_1 \preccurlyeq e_2$ is not enough to establish $\sigma_1,\ \sigma_2 \vdash e_1 \preccurlyeq$ `assume` $e_2\ a2$. We also need to show that the result and reduction for e_1 are more precise than the assumption on the right. We achieve this with the help of the static analysis, which we can then relate to the gradual analysis with Lemma 5.2.

Monotonicity with Respect to Precision of Operations. The third dimension is the precision of the optimistic abstract operations with which a gradual analysis is instantiated. In other words, the analysis result and elaboration need to be monotonic with respect to the precision of the operations.

To this end, we define a precision relation for operations, and say that $ops_1 \sqsubseteq ops_2$ holds when, for the same inputs, the operations in ops_1 yield results that are as precise as those in ops_2.

With the previous definitions we establish adaptations of the static and dynamic gradual guarantees of gradual typing, which embody the monotonicity of the gradual analysis, elaboration, and reduction with respect to the precision of terms and operations.

$$\boxed{\sigma_1,\ \sigma_2 \vdash e_1 \preccurlyeq e_2}$$

$$\frac{\sigma_1,\ \sigma_2 \vdash e_1 \preccurlyeq e_2 \qquad a_1 \sqsubseteq a_2}{\texttt{assume}\ e_1\ a_1 \preccurlyeq \texttt{assume}\ e_2\ a_2}\ \textsc{LTEEAssume} \qquad \frac{\sigma_1,\ \sigma_2 \vdash e_1 \preccurlyeq e_2}{\texttt{assume}\ e_1\ a \preccurlyeq e_2}\ \textsc{LTEEAssumeL}$$

$$\frac{\sigma_1,\ \sigma_2 \vdash e_1 \preccurlyeq e_2 \qquad [\![e_1]\!]_a\ \sigma_1 \mapsto a_1 \qquad a_1 \sqsubseteq a_2}{e_1 \preccurlyeq \texttt{assume}\ e_2\ a_2}\ \textsc{LTEEAssumeR}$$

Fig. 11. Relevant rules of the precision relation for expressions.

The Static Gradual Guarantee (SGG) establishes that when the gradual analysis is applied to expressions related by precision, the elaborated expressions and analysis results are also related by precision.

Theorem 5.6 (SGG). *If* $\sigma_1,\ \sigma_2 \vdash e_1 \preccurlyeq e_2$, $ops_1 \sqsubseteq ops_2$, $\sigma_1 \vdash \langle ops_1 \rangle\ e_1 \rightsquigarrow e_1' : a_1$ *and* $\sigma_2 \vdash \langle ops_2 \rangle\ e_2 \rightsquigarrow e_2' : a_2$, *then*

(1) $\sigma_1,\ \sigma_2 \vdash e_1' \preccurlyeq e_2'$
(2) $a_1 \sqsubseteq a_2$

The Dynamic Gradual Guarantee (DGG) captures the fact that reduction is monotonic with respect to expression precision. Similar to the case for well-formedness of the elaboration, the DGG establishes that the most precise expression error-approximates the least precise one.

Theorem 5.7 (DGG). *If* $\alpha_\rho(\rho_1),\ \alpha_\rho(\rho_2) \vdash e_1 \preccurlyeq e_2$, $ops_1 \sqsubseteq ops_2$, $\alpha_\rho(\rho_1) \vdash \langle ops_1 \rangle\ e_1 \rightsquigarrow e_1' : a_1$ *and* $\alpha_\rho(\rho_2) \vdash \langle ops_2 \rangle\ e_2 \rightsquigarrow e_2' : a_2$, *then*

(1) *if* $[\![e_2']\!]\ \rho_2 \mapsto v_2$ *then* $[\![e_1']\!]\ \rho_1 \mapsto v_1$ *and* $v_1 \sqsubseteq v_2$.
(2) *if* $[\![e_1']\!]\ \rho_1 \mapsto v_1$ *and* $v_1 \neq \bot$ *then* $[\![e_2']\!]\ \rho_2 \mapsto v_2$ *and* $v_1 \sqsubseteq v_2$.

These results are illustrated in Fig. 12.

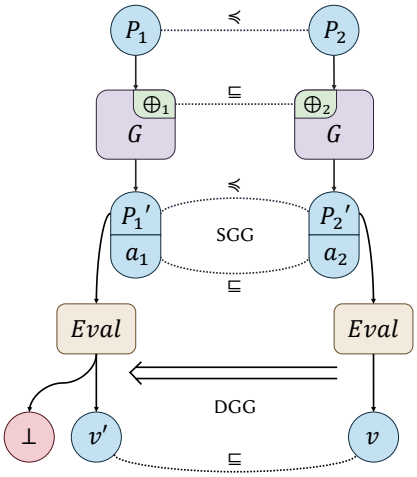


Fig. 12. Gradual guarantees for the parametric gradual analysis.

6 Discussion

We now discuss how we envision GAI to be used in practice and some considerations about future work.

6.1 GAI in Practice

One of the objectives of GAI is to allow users to check that the programs they analyze indeed respect the optimistic assumptions leveraged by an analysis. On this front, we imagine that users of gradual analyses would add them to their development processes as follows.

During development, a user will write programs and analyze them with a gradual analysis. She will assess if the results are satisfactory, but also test the elaborated program produced by the analysis in real scenarios, in order to check that assumptions are not violated and confirm the validity of the results. If any of the runtime checks fail, this signals that the program needs to be corrected.

Once the results of the analysis are satisfactory, and there is enough certainty that assumptions are not violated, the user can trust that the original program satisfies the assumptions of the analysis in the contexts where it was tested, and that the results of the analysis are valid under

those conditions. In this way, the user has more assurances that if she decides to run the original program in production, which is computationally cheaper than running an elaborated program with many (potentially expensive) runtime checks, it will behave as expected.

This mode of operation relies on the properties established by Theorem 5.4 and Theorem 5.5.

Of course, testing the elaborated program in a number of representative scenarios does not guarantee that the assumptions will not be broken in a scenario that was not considered. But this is still an improvement over soundiness, because a gradual analysis gives users a way to check that they are respecting the optimistic assumptions.

6.2 Fixpoint Algorithm & Future Work

To simplify our exploration, our formal development considers a very simple language which makes the fixpoint algorithm of abstract interpretation unnecessary. We plan on extending this language with first-order functions, in order to assess the correct interaction of the fixpoint algorithm with the essence of the proposed gradual approach.

This being said, there are some clues that suggest that the ideas presented in this work are compatible with the fixpoint algorithm. In particular, one of the requirements to guarantee termination of the fixpoint algorithm and ensure that an analysis finds a solution, is that all the abstract operations should be monotonic. In the case of the optimistic operations, we already require them to be monotonic. Apart from that, the only other language construct that we handle in a non-standard way is the `assume`. The abstract interpreter analyses the subject of the `assume` and combines the result with the assumption (using the *meet*). So the corresponding abstract operation for that construct is the *meet*, which is monotonic on both arguments by definition. And, as long as the lattice used as the abstract domain has finite height, or that an appropriate widening operator is provided, the algorithm should converge to a solution.

7 Related Work

The application of the techniques from gradual typing to other reasoning approaches has already been explored in the literature, in particular in gradual program verification [Bader et al. 2018; Wise et al. 2020] and gradual program analysis for null pointers [Estep et al. 2021]. In both lines of work, following the approach of gradual type systems, optimism is driven by imprecision in annotated information (be it nullability, or logical predicates). These approaches exploit the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016] to lift static reasoning to operate over imprecise information. In contrast, GAI introduces optimism at the level of the operations used by the abstract interpreter, as exemplified with the optimistic addition. Combining this approach to GAI with explicit, possibly incomplete and/or imprecise annotations is an interesting direction for future work.

8 Conclusion

We study an extension of the framework of abstract interpretation with techniques inspired by gradual typing. The purpose is two-fold: to produce analyses that leverage optimistic assumptions in order to increase precision, and to provide a way to check whether those assumptions are respected by the analyzed programs.

We show that these objectives are achievable with a mechanized formalization of the foundations of Gradual Abstract Interpretation for a small language in the Rocq prover. We have discussed how gradual analyses could become an integral part of the development life-cycle, and pondered the compatibility of the ideas put forth by this work with the fixpoint algorithm, the cornerstone of the framework of abstract interpretation.

Future work includes the extension of the proof of concept with first-order functions, to formally study the interaction of the fixpoint algorithm with GAI.

References

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2018)*, Işil

Dillig and Jens Palsberg (Eds.), Vol. 10747. Springer-Verlag, Los Angeles, CA, USA, 25–46.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

Sam Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine. 2021. Gradual Program Analysis for Null Pointers. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021)*, Manu Sridharan and Anders Møller (Eds.). Aarhus, Denmark.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM Press, St Petersburg, FL, USA, 429–442. See erratum: https://www.cs.ubc.ca/ rxg/agt-erratum.pdf.

Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (jan 2015), 44–46. doi:10.1145/2644805

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proceedings of the ACM on Programming Languages* 2, ICFP (Sept. 2018), 73:1–73:30.

Henry G. Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366. doi:10.1090/S0002-9947-1953-0053041-6

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European Conference on Object-Oriented Programming* (Lancaster, UK) *(ECOOP'11)*. Springer-Verlag, Berlin, Heidelberg, 52–78.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. 81–92.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual Verification of Recursive Heap Data Structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 228:1–228:28.

Type Checking Dependently Sorted Nominal Signatures

Maribel Fernández
King's College London
UK
maribel.fernandez@kcl.ac.uk

Miguel Pagano
Univ. Nac. Córdoba
Argentina
miguel.pagano@unc.edu.ar

Nora Szasz
Universidad ORT
Uruguay
szasz@ort.edu.uy

Álvaro Tasistro
Universidad ORT
Uruguay
tasistro@ort.edu.uy

ABSTRACT

Dependently sorted nominal signatures generalise standard many-sorted first-order signatures by including nominal abstraction constructs and sorts that depend on terms. They can provide a suitable basis for a logical framework with a distinctive first-order flavour, provided a type checking algorithm can be defined. In this paper we show that type-checking is decidable: we present a type-checking algorithm for dependently sorted nominal signatures that can be directly implemented in a functional programming language. We prove its correctness and completeness with respect to the sorting system.

KEYWORDS

Nominal Terms, Type-checking, Dependent Types, Functional Programming.

ACM Reference Format:
Maribel Fernández, Miguel Pagano, Nora Szasz, and Álvaro Tasistro. 2025. Type Checking Dependently Sorted Nominal Signatures. In *Proceedings of 37th Symposium on Implementation and Application of Functional Languages (IFL 2025).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Nominal languages [2, 5, 6] provide support for the specification of data structures that include bound names and for the formalisation of their properties. Dependently sorted nominal signatures [1] can be seen as a generalisation of nominal signatures where sorts can now depend on terms. The resulting *dependently sorted language* inherits the distinctive first-order algebraic flavour of nominal signatures, and can serve as a basis for a logical framework [4]. More precisely, dependently sorted nominal signatures provide a foundation for a theory of expressions subject to alpha-conversion, on top of which a system of dependent sorts is built that respects alpha-equivalence. Names carry sorts of data —but not abstraction (or "higher-order") sorts. This yields a limited form of computation associated to the elimination of abstractions (concretion), that can be solved at the level of syntactic meta-definitions. Thus the language becomes a first-order dependent sorts system.

Previous work showed that dependently sorted nominal signatures provide an adequate foundation for a logical framework: specifications of first and higher-order logic, and lambda-calculi (untyped, simply typed) have been defined, together with corresponding induction principles (that are directly derived thanks to the nominal foundation). However, for the system to be usable in practice, a type checking algorithm is needed.

In this paper, we describe an algorithm to check whether a term is well sorted with respect to a set of declarations (that assign a type to each function symbol and sort constructor) and a context (that assigns sorts to the free names occurring in the term. This algorihtm can be directly implemented in a functional language: an implementation in Haskell is in progress. We prove that the type checker is sound and complete with respect to the sorting system previously defined [1]. This is a first step towards building a (first-order) functional programming language or a software verification tool based on nominal syntax.

2 SYNTAX

In this section we give a short overview of the system presented in LFMTP 2025 [1].

2.1 Grammar

Consider a countably infinite set of name sorts, each one inhabited by a countably infinite set of names (*atoms*). Let a, b, c, range over atoms. Let also there be countably infinite sets of *parameters* $X, Y, Z, \ldots \in \mathbb{X}$; *term constructors*, $f, g \ldots \in \mathbb{F}$; and *sort constructors* $\mathcal{F}, \mathcal{G}, \ldots \in \mathbb{C}$. Following Gabbay's *permutative convention* [3]: a, b range over *distinct atoms*. The notation $\bar{t}$ refers to a vector of terms $t_0, \ldots, t_n$ with $n \geqslant 0$; given a term t' and $\bar{t} = t_0, \ldots, t_n$, we use $\bar{t}, t'$ to represent the vector $t_0, \ldots, t_n, t'$.

IFL 2025, October 1-3, 2025, Montevideo, Uruguay

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Sorts γ and terms t are generated by the grammar below. We will use M to stand for either.

$$\begin{array}{lll}
\gamma ::= \mathcal{F}\,\overline{t} & & \textit{data sorts}\\
\quad \mid \ll a : \mathcal{F}\,\overline{t} \gg \gamma & & \textit{abstraction sorts}\\
t ::= a & & \textit{atom}\\
\quad \mid X\overline{[t]} & & \textit{parameter with term concretions}\\
\quad \mid f\,\overline{t} & & \textit{application}\\
\quad \mid \ll a : \mathcal{F}\,\overline{t} \gg t & & \textit{abstraction}
\end{array}$$

Sorts are built using sort constructors or abstractions and can depend on terms, which can be atoms, parameters, application of a term constructor to a tuple of terms, or the abstraction of an atom on a term. We say an *expression* (sort or term) is *ground* iff it contains no parameters. When a parameter X has no concretions, we omit the square brackets. As stated earlier, parameters are intended for declarations, as shown in the previous section, while sorting judgments (i.e., the language generated by the system to be given in the next section) involve only ground expressions.

2.2 Operations and Relations

We define the action of permutations on sorts and terms. Here, $\overline{\pi \cdot t}$ denotes the vector $\pi \cdot t_0, \ldots, \pi \cdot t_n$.

Definition 1 (Permutation Action). *A* permutation π *is a bijection on the set of atoms,* $\mathbb{A}$*, with finite domain. We represent permutations as lists of swappings* $(a\ b)$*. The identity permutation is written* id.

$$\begin{array}{c}
\pi \cdot a \triangleq \pi(a)\\
\pi \cdot \ll a : \mathcal{F}\,\overline{s} \gg M \triangleq \ll \pi(a) : \mathcal{F}\,\overline{\pi \cdot s} \gg (\pi \cdot M)\\
\pi \cdot (X\overline{[t]}) \triangleq X\overline{[\pi \cdot t]}\\
\pi \cdot f\,\overline{t} \triangleq f\,\overline{\pi \cdot t}\\
\pi \cdot \mathcal{F}\,\overline{t} \triangleq \mathcal{F}\,\overline{\pi \cdot t}
\end{array}$$

To define alpha-equivalence, we first introduce the freshness relation. Call $a \# M$ a *freshness constraint.*

Definition 2 (Freshness Relation). *A* freshness judgement *has the form* $\vdash a \# M$*. To derive freshness judgements we use the following rules. A premise* $\vdash a \# \overline{t}$ *is to be expanded as* $\vdash a \# t_0, \ldots,$ $\vdash a \# t_n$*.*

$$(atm)^{\#}\ \frac{}{\vdash a \# b} \qquad (cns)^{\#}\ \frac{\vdash a \# \overline{t}}{\vdash a \# \mathcal{F}\,\overline{t}} \qquad (app)^{\#}\ \frac{\vdash a \# \overline{t}}{\vdash a \# f\,\overline{t}}$$

$$(ab_{aa})^{\#}\ \frac{\vdash a \# \mathcal{F}\,\overline{t}}{\vdash a \# \ll a : \mathcal{F}\,\overline{t} \gg M} \qquad (ab_{ab})^{\#}\ \frac{\vdash a \# M \qquad \vdash a \# \mathcal{F}\,\overline{t}}{\vdash a \# \ll b : \mathcal{F}\,\overline{t} \gg M}$$

$$(var)^{\#}\ \frac{\vdash a \# \overline{t}}{\vdash a \# X\overline{[t]}}$$

The main difference with respect to the freshness relation for standard nominal terms is the introduction of new rules $(ab_{aa})^{\#}$, $(ab_{ab})^{\#}$, $(cns)^{\#}$, and $(var)^{\#}$, the rule for concretion, which checks freshness only in terms and not in the parameter. As will be commented again later, parameters stand for arbitrary *closed* ground terms of the target language.

Definition 3 (Alpha-equivalence Relation). *An* α*-*equivalence judgement *has the form* $\vdash M \approx_\alpha N$*, where* M *and* N *are ground. We introduce now the rules defining this relation. A premise* $\vdash \overline{s} \approx_\alpha \overline{t}$*, where* $\overline{s}$ *and* $\overline{t}$ *must always be of the same size, is to be expanded in an element-wise manner into premises* $\vdash s_i \approx_\alpha t_i$*.*

$$(atm)^{\alpha}\ \frac{}{\vdash a \approx_\alpha a} \qquad (cns)^{\alpha}\ \frac{\vdash \overline{s} \approx_\alpha \overline{t}}{\vdash \mathcal{F}\,\overline{s} \approx_\alpha \mathcal{F}\,\overline{t}}$$

$$(app)^{\alpha}\ \frac{\vdash \overline{s} \approx_\alpha \overline{t}}{\vdash f\,\overline{s} \approx_\alpha f\,\overline{t}}$$

$$(ab_{aa})^{\alpha}\ \frac{\vdash \mathcal{F}\,\overline{t} \approx_\alpha \mathcal{F}\,\overline{u} \qquad \vdash M \approx_\alpha M'}{\vdash \ll a : \mathcal{F}\,\overline{t} \gg M \approx_\alpha \ll a : \mathcal{F}\,\overline{u} \gg M'}$$

$$(ab_{ab})^{\alpha}\ \frac{\vdash \mathcal{F}\,\overline{t} \approx_\alpha \mathcal{F}\,\overline{u} \qquad \vdash M \approx_\alpha (a\ b) \cdot M' \qquad \vdash a \# M'}{\vdash \ll a : \mathcal{F}\,\overline{t} \gg M \approx_\alpha \ll b : \mathcal{F}\,\overline{u} \gg M'}$$

This definition of alpha-equivalence generalises the standard one for nominal terms. For simplicity, we omit a rule for parameters, which is not essential but would facilitate the writing of declarations.

Lemma 1 (Equivariance). *If* $\vdash a \# M$ *then* $\vdash \pi \cdot a \# \pi \cdot M$*. Similarly if* $\vdash M \approx_\alpha N$ *then* $\vdash \pi \cdot M \approx_\alpha \pi \cdot N$*.*

Proof. Straightforward induction. □

Freshness is stable by α-equivalence:

Lemma 2. *If* $\vdash a \# M$ *and* $\vdash M \approx_\alpha N$ *then* $\vdash a \# N$*.*

Proof. By induction on the freshness relation. Use equivariance. □

Lemma 3. $\approx_\alpha$ *is a congruence.*

Proof. Induction on the definition of $\approx_\alpha$. □

Definition 4 (Atom Substitution). *We write* $[a \mapsto t]$ *for the operation that substitutes the atom* a *by the term* t*. This is defined on expressions as follows:*

$$\begin{array}{rl}
a\,[a \mapsto t] \triangleq & t\\
b\,[a \mapsto t] \triangleq & b\\
(f\,\overline{s})\,[a \mapsto t] \triangleq & f\,(\overline{s\,[a \mapsto t]})\\
(\mathcal{F}\,\overline{s})\,[a \mapsto t] \triangleq & \mathcal{F}\,(\overline{s\,[a \mapsto t]})\\
(X\overline{[t]})\,[a \mapsto t'] \triangleq & X\overline{[t\,[a \mapsto t']]}\\
(\ll a : \mathcal{F}\,\overline{t} \gg M)\,[a \mapsto t] \triangleq & \ll a : \mathcal{F}\,\overline{t\,[a \mapsto t]} \gg M\\
(\ll b : \mathcal{F}\,\overline{t} \gg M)\,[a \mapsto t] \triangleq & \ll c : \mathcal{F}\,\overline{t\,[a \mapsto t]} \gg ((b\ c) \cdot M)\,[a \mapsto t]\\
& \qquad (\vdash c \# M, \vdash c \# t).
\end{array}$$

Some explanations are in order: to avoid capturing unabstracted atoms, when an atom substitution acts upon an abstraction or abstraction sort (last case above), a suitable alpha-equivalent representative of the latter is first chosen. Any implementation of this definition as a recursive function must accommodate a suitable mechanism for the generation of names; this is most easily achieved by the threading of global state throughout the function or by the use of a global choice function that returns the next available name.

Atom substitutions work uniformly on alpha-equivalence classes.

Lemma 4. *If* $\vdash M \approx_\alpha N$ *and* $\vdash t \approx_\alpha u$ *then* $\vdash M[a \mapsto t] \approx_\alpha N[a \mapsto u]$

Proof. Induction on the derivation of $\vdash M \approx_\alpha N$. □

A concretion $w[t]$ is a partial operation: if w is an abstraction $\ll a : \mathcal{F}\,\overline{s} \gg u$, then its concretion to t evaluates to the body of the abstraction, u, where the abstracted atom is substituted by t. If w is the parameter X (possibly with other concretions suspended in it), the concretion remains "suspended" (until X is instantiated). Under the sorting system of the next section, concretion of a parameter will be well-sorted only if the parameter is of an (appropriate) abstraction sort.

Definition 5 (Concretion). *Concretion is a partial operation:*

$$(\ll a : \mathcal{F}\,\overline{s} \gg u)[t] \triangleq u\,[a \mapsto t]$$
$$(X\overline{[t]})[t'] \triangleq X[\overline{t}, t']$$

Definition 6 (Parameter Instantiation). *A* parameter instantiation *is a finite mapping from parameters to terms, and it acts on expressions as just grafting (i.e., without a control of capture), subject to the condition that each parameter to be replaced is in the domain of the instantiation.*

2.3 Sorting judgements

We use five forms of judgements: 1) Well-formedness of signature Σ, formally $\vdash \Sigma$ sig-ok (Fig. 1a); 2) Well-formedness of telescopes $\mathscr{T}$ under a valid signature, $\vdash_\Sigma \mathscr{T}$ tel-ok(Fig. 1b); 3) Well-formedness of contexts of atoms (Fig. 1c), $\mathscr{T} \vdash_\Sigma \Gamma$ ctx-ok; 4) Well-formedness of sorts (Fig. 1d), $\mathscr{T}; \Gamma \vdash_\Sigma \gamma$ sort; and 5) Well-sortedness of terms (Fig. 1e), $\mathscr{T}; \Gamma \vdash_\Sigma t : \gamma$.

As indicated above, the sorts are either *data sorts* or *abstraction sorts*. Data sorts are introduced by *sort constructors* $\mathcal{F}$, and these can only introduce data sorts, never an abstraction sort —the latter being formed exclusively by the binder $\ll\_ : \_ \gg \_$. Similarly, terms of the data sorts are formed by *(term) constructors* f, and terms of abstraction sorts exclusively by the corresponding binder. *Signatures* are sequences of *declarations* of sort and term constructors. As already explained, a declaration specifies the sorts of the corresponding parameters and a freshness context. These parameter declarations are called *telescopes*. The word *context* is reserved for *atom contexts*, Γ, necessary to sort abstractions.

As already stated, the intention is that the system is used for generating well-formed *ground* sorts and terms. The rules given below define well-formed scripts of declarations (i.e. signatures), which involve not only ground expressions but also expressions with parameters.

First, notice the use of *freshness contexts* (Δ) in declarations. They involve conditions of the form $a \# X$, where the atom a is to appear bound in the declaration and X is any parameter of the declaration.

This defines the side condition on well-formedness of the contexts Δ. The rules check the validity of the freshness conditions whenever a declaration is put into use, i.e. in rules (data) and (constr). There the constructor employed must be declared in the signature with a telescope $\mathscr{T}'$ and freshness context Δ, as stated in the side condition. Then a fresh version of $\mathscr{T}'$, as well as of Δ, are created by employing new atoms so as to avoid possible collisions with unabstracted atoms in the expression being checked. We call this new telescope $\mathscr{T}'_\#$, and the new context $\Delta_\#$. Then it is checked that the tuple $\overline{t}$ of arguments *fits* the telescope $\mathscr{T}'_\#$ and at the same time the conditions in $\Delta_\#$ are satisfied, with the mentioned parameters instantiated accordingly by the tuple $\overline{t}$ —which we write $(\Delta_\#)_{\overline{t}}$. That a tuple of terms *fits* a telescope has the (obvious) meaning that: a) The telescopes and the context are well-formed. b) They are of the same length. c) Each term has the sort attached to its corresponding parameter, instantiated on the preceding terms in the tuple.

An equally valid alternative is that the freshness conditions are rather *imposed* by the system, i.e. a freshness declaration is to be interpreted as an *assumption* on part of the user about the employment of names in the (ground) expressions to be generated. The conditions can be imposed by the system by generating in each case a sample chosen among all the alpha-equivalent expressions satisfying the sorting rules that also respects the freshness conditions. For this to work, it is essential that the system is closed under alpha-equivalence —which will be shown presently— and that the freshness conditions are only on bound atoms —which is already imposed in the well-formation of declarations.

In the rule (constr) we use the notation $\overline{u}_{\overline{t}}$, which stands for the instantiation of the parameters of the tuple of terms $\overline{u}$ with the tuple $\overline{t}$.

Finally, let us remark that, as stated in rule (fun-sig) and (constel), valid telescopes and target sorts of term constructors cannot depend on (unabstracted) atoms. Also note that in the rules we omit premises that can be deduced from some explicitly mentioned premise.

3 PROPERTIES OF THE SORTING SYSTEM

In this section we analyse in more detail the properties of the dependently sorted system given above, before designing a type checking algorithm for it.

Lemma 5 (Concretion is well-typed). *Let* t, t' *be terms,* γ *a sort, and* a *an atom. If* $\mathscr{T}; \Gamma \vdash_\Sigma t : \ll a : \mathcal{F}\,\overline{s} \gg \gamma$, *and* $\mathscr{T}; \Gamma \vdash_\Sigma t' : \mathcal{F}\,\overline{s}$; *then* $t[t']$ *is well-defined and* $\mathscr{T}; \Gamma \vdash_\Sigma t[t'] : \gamma[a \mapsto t']$.

Notice that the converse does not hold: consider the case of t being an abstraction $\ll a : \mathcal{F}\,\overline{s} \gg u$ where a does not occur free in u.

Proof. One can prove that a term has an abstraction sort if and only if it is an abstraction or a meta-variable (possibly with some concretions). Those are the cases when the concretion $t[t']$ is well-defined.

The well-sortedness of the concretion is trivial when t is a meta-variable. The case for the abstraction depends on the substitution lemma. □

Lemma 6 (Substitution lemma). *If* $\mathscr{T}; \Gamma, (a : \mathcal{F}\,\overline{s}), \Gamma' \vdash_\Sigma t : \gamma$ *and* $\mathscr{T}; \Gamma \vdash_\Sigma u : \mathcal{F}\,\overline{s}$, *then* $\mathscr{T}; \Gamma, (\Gamma'[a \mapsto u]) \vdash_\Sigma t[a \mapsto u] : \gamma[a \mapsto u]$.

The inversion lemma gives information about the shape of the sort of some term; when the term has parameters, then we can state precisely the shape of the term because the sort system is

$$\text{(empty-sig)}\ \frac{}{\vdash \langle\rangle\ \text{sig-ok}} \qquad \text{(sort-sig)}\ \frac{\vdash_\Sigma \mathscr{T}\ \text{tel-ok}}{\vdash \Sigma, \langle \mathcal{F} : \mathscr{T} \to \text{data}\,;\, \Delta\rangle\ \text{sig-ok}} \begin{cases} \mathcal{F} \notin \text{dom}(\Sigma) \\ \Delta\ \text{well-formed} \end{cases}$$

$$\text{(fun-sig)}\ \frac{\mathscr{T};\cdot \vdash_\Sigma \mathcal{F}\,\overline{t}\ \ \text{sort}}{\vdash \Sigma, \langle f : \mathscr{T} \to \mathcal{F}\,\overline{t}\,;\, \Delta\rangle\ \text{sig-ok}} \begin{cases} f \notin \text{dom}(\Sigma) \\ \Delta\ \ \text{well-formed} \end{cases}$$

(a) *Rules for signatures.*

$$\text{(empty-tel)}\ \frac{\vdash \Sigma\ \text{sig-ok}}{\vdash_\Sigma \cdot\ \text{tel-ok}} \qquad \text{(cons-tel)}\ \frac{\mathscr{T};\cdot \vdash_\Sigma \gamma\ \ \text{sort}}{\vdash_\Sigma \mathscr{T}, (X : \gamma)\ \text{tel-ok}}\ X \notin \text{dom}(\mathscr{T})$$

(b) *Rules for telescope formation.*

$$\text{(emp-ctx)}\ \frac{\vdash_\Sigma \mathscr{T}\ \text{tel-ok}}{\mathscr{T} \vdash_\Sigma \cdot\ \text{ctx-ok}} \qquad \text{(cons-ctx)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \mathcal{F}\,\overline{t}\ \ \text{sort}}{\mathscr{T} \vdash_\Sigma \Gamma, (a : \mathcal{F}\,\overline{t})\ \text{ctx-ok}}\ a \notin \text{dom}(\Gamma)$$

(c) *Rules for well-formed contexts.*

$$\text{(data)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \overline{t}\ \text{fits}\ \mathscr{T}'_{\#}[(\Delta_{\#})_{\overline{t}}]}{\mathscr{T};\Gamma \vdash_\Sigma \mathcal{F}\,\overline{t}\ \ \text{sort}} \begin{cases} \mathcal{F} \in \text{dom}(\Sigma) \\ \Sigma(\mathcal{F}) = \mathscr{T}' \to \text{data}; \Delta \end{cases}$$

$$\text{(abs-*)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \mathcal{F}\,\overline{t}\ \ \text{sort} \quad \mathscr{T};(\Gamma, b : \mathcal{F}\,\overline{t}) \vdash_\Sigma (a\ b)\cdot\gamma\ \ \text{sort}}{\mathscr{T};\Gamma \vdash_\Sigma \ll a : \mathcal{F}\,\overline{t} \gg \gamma\ \ \text{sort}} \begin{cases} b \notin \text{dom}(\Gamma) \\ b \# \gamma \end{cases}$$

(d) *Rules for well-formed sorts.*

$$\text{(atm)}\ \frac{\mathscr{T} \vdash_\Sigma \Gamma\ \text{ctx-ok}}{\mathscr{T};\Gamma \vdash_\Sigma a : \Gamma(a)}\ a \in \text{dom}(\Gamma) \qquad \text{(var1)}\ \frac{\mathscr{T} \vdash_\Sigma \Gamma\ \text{ctx-ok}}{\mathscr{T};\Gamma \vdash_\Sigma X : \mathscr{T}(X)}\ X \in \text{dom}(\mathscr{T})$$

$$\text{(var2)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma X\overline{[t]} : \ll a : \mathcal{F}\,\overline{s} \gg \gamma \qquad \mathscr{T};\Gamma \vdash_\Sigma t' : \mathcal{F}\,\overline{s}}{\mathscr{T};\Gamma \vdash_\Sigma X[\overline{t}, t'] : \gamma[a \mapsto t']}$$

$$\text{(constr)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \overline{t}\ \text{fits}\ \mathscr{T}'_{\#}[(\Delta_{\#})_{\overline{t}}]}{\mathscr{T};\Gamma \vdash_\Sigma f\,\overline{t} : \mathcal{F}\,(\overline{u}_{\overline{t}})} \begin{cases} f \in \text{dom}(\Sigma) \\ \Sigma(f) = \mathscr{T}' \to \mathcal{F}\,\overline{u}; \Delta \end{cases}$$

$$\text{(abs)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \mathcal{F}\,\overline{t}\ \ \text{sort} \qquad \mathscr{T};(\Gamma, b : \mathcal{F}\,\overline{t}) \vdash_\Sigma (a\ b)\cdot t : (a\ b)\cdot\gamma}{\mathscr{T};\Gamma \vdash_\Sigma \ll a : \mathcal{F}\,\overline{t} \gg t : \ll a : \mathcal{F}\,\overline{t} \gg \gamma} \begin{cases} b \notin \text{dom}(\Gamma) \\ b \# \{t, \gamma\} \end{cases}$$

$$\text{(conv)}\ \frac{\cdot;\Gamma \vdash_\Sigma t : \gamma}{\cdot;\Gamma \vdash_\Sigma t : \gamma'}\ \gamma \approx_\alpha \gamma'$$

(e) *Rules for well-sorted terms.*

Figure 1: Sorting System

syntax-directed. For ground terms the inversion lemma has almost the same statement: we replace $\gamma = \gamma'$ by $\gamma \approx_\alpha \gamma'$.

Lemma 7 (Inversion of sorting). *Let $\mathscr{T} \neq \cdot$.*

(1) If $\mathscr{T};\Gamma \vdash_\Sigma a : \gamma$, then $\gamma = \Gamma(a)$.
(2) If $\mathscr{T};\Gamma \vdash_\Sigma X : \gamma$, then $\gamma = \mathscr{T}(X)$.
(3) If $\mathscr{T};\Gamma \vdash_\Sigma X[\overline{t}, t'] : \gamma$, then there exist $F, \overline{s}, \gamma'$ such that $\mathscr{T};\Gamma \vdash_\Sigma X\overline{[t]} : \ll a : \mathcal{F}\,\overline{s} \gg \gamma'$, $\mathscr{T};\Gamma \vdash_\Sigma t' : \mathcal{F}\,\overline{s}$, and $\gamma = \gamma'[a \mapsto t']$.
(4) If $\mathscr{T};\Gamma \vdash_\Sigma f\,\overline{t} : \gamma$, then $f \in \text{dom}(\Sigma)$ and there exists $\mathcal{F}$, $\overline{u}$, $\mathscr{T}'$, and Δ, such that $\Sigma(f) = \mathscr{T}' \to \mathcal{F}\,\overline{u}; \Delta$ and $\gamma = \mathcal{F}\,(\overline{u}_{\overline{t}})$.
(5) If $\mathscr{T};\Gamma \vdash_\Sigma \ll a : \mathcal{F}\,\overline{t} \gg t : \gamma$, then there exist γ' such that $\gamma = \ll a : \mathcal{F}\,\overline{t} \gg \gamma'$, $\mathscr{T};\Gamma \vdash_\Sigma \mathcal{F}\,\overline{t}$ sort and $\mathscr{T};(\Gamma, b : \mathcal{F}\,\overline{t}) \vdash_\Sigma (a\ b)\cdot t : (a\ b)\cdot\gamma'$, for any b such that $b \notin \text{dom}(\Gamma)$ and $b \# \{t, \gamma'\}$.

Proof. When $\mathscr{T} \neq \cdot$, there is only one rule that can be used to conclude $\mathscr{T};\Gamma \vdash_\Sigma t : \gamma$. For each item, we *invert* that rule to obtain the shape of γ.

The proof of the inversion lemma for ground terms proceeds by induction on the derivation of the sorting judgment. We show the proof for $t = a$. If the last rule used is (atm), then we have $\gamma = \Gamma(a)$; since $\approx_\alpha$ is reflexive, we conclude $\gamma \approx_\alpha \Gamma(a)$. If the last rule used is (conv), then by inductive hypothesis we deduce $\gamma \approx_\alpha \Gamma(a)$. Since $\approx_\alpha$ is symmetric and transitive, we have $\gamma' \approx_\alpha \Gamma(a)$. □

To get uniqueness of typing (up-to α-equivalence) we need to proceed by induction on the size of terms.

Lemma 8 (uniqueness of typing up-to α-equivalence). *If $\cdot;\Gamma \vdash_\Sigma t : \gamma_1$, and $\cdot;\Gamma \vdash_\Sigma t : \gamma_2$, then $\vdash \gamma_1 \approx_\alpha \gamma_2$.*

Proof. Notice that t can be an atom a, an application $f\,\bar{t}$, or an abstraction $\ll a : \mathcal{F}\,\bar{t} \gg t$. In the first two cases, the result follows directly from inversion because there exists a precise sort γ such that $\gamma_1 \approx_\alpha \gamma$ and $\gamma_2 \approx_\alpha \gamma$; therefore $\gamma_1 \approx_\alpha \gamma_2$.

When $t = \ll a : \mathcal{F}\,\bar{t} \gg t$, then by inversion we get γ_1' and γ_2' such that $\gamma_1 \approx_\alpha \ll a : \mathcal{F}\,\bar{t} \gg \gamma_1'$ and $\gamma_2 \approx_\alpha \ll a : \mathcal{F}\,\bar{t} \gg \gamma_2'$; moreover $\mathscr{T}; (\Gamma, b : \mathcal{F}\,\bar{t}) \vdash_\Sigma (a\ b) \cdot t : (a\ b) \cdot \gamma_1'$ and $\mathscr{T}; (\Gamma, b : \mathcal{F}\,\bar{t}) \vdash_\Sigma (a\ b) \cdot t : (a\ b) \cdot \gamma_2'$. Then by inductive hypothesis on $(a\ b) \cdot t$ we have $(a\ b) \cdot \gamma_1' \approx_\alpha (a\ b) \cdot \gamma_2'$. □

Lemma 9 (Compatibility with instantiations). *If $\mathscr{T}'; \Gamma \vdash_\Sigma t : \gamma$, and $\mathscr{T}; \Gamma' \vdash_\Sigma \sigma$ fits $\mathscr{T}'$ with $\Gamma \bowtie \Gamma'$, then $\mathscr{T}; \Gamma \cup \Gamma' \vdash_\Sigma t\,\sigma : \gamma\,\sigma$.*

4 TYPE-CHECKING ALGORITHM

The rules of the sorting system are "almost" syntax directed. In fact, notice that for terms with parameters (non-ground terms) they are syntax-directed, because the rule (conv) is only for ground terms. In the following, we introduce syntax-directed rules for incrementally checking signatures, telescopes, contexts, sorts, and terms.

Definition 7 (Sort-checking). *Given a signature Σ, telescope $\mathscr{T}$, context Γ, and a well-formed sort γ, $\mathscr{T}; \Gamma \vdash_\Sigma \gamma$ sort, we define the following relations that are syntax-directed and can thus be understood as the definition of a sort-checking algorithm: $\mathscr{T}; \Gamma \vdash_\Sigma t \Leftarrow \gamma$, in Fig. 2, and $\mathscr{T}; \Gamma \vdash_\Sigma t \Rightarrow \gamma$, in Fig. 3.*

Notice the preconditions assuming the well-formedness of signatures, telescopes, and sorts. This illustrates that first we check the signature, once we know it is well-formed (initially the empty signature is obviously well-formed), to extend it with a new constructor (either sort or term constructor) we check the telescope with respect to that signature. So, in judgment $\vdash_\Sigma \mathscr{T} \Leftarrow$, we can assume $\vdash \Sigma$ sig-ok.

When we check that an instantiation fits in a telescope, in Fig. 6, we change the perspective on telescopes; instead of seeing them as snoc-lists we assume them to be cons-lists. Here we also check that the instantiation fulfills the freshness condition; to check $\vdash \Delta|_X[X \mapsto t] \Leftarrow$ we get every freshness condition $a \# X$ in Δ and check $\vdash a \# t$.

4.1 Properties of the type-checker

Theorem 1 (Correctness and completeness). *The type-checker, Fig. 2 and Fig. 3 is correct and complete with respect to the typing system Fig. 1e.*

5 CONCLUSIONS

Dependently sorted nominal signatures constitute:

- A logical framework in which it is possible to develop formal and certified proofs of properties of, among others, programming languages. In [1] we have sketched how some basic meta-theory of lambda calculi can be developed.
- A basis for a (first-order) functional language built on top a of nominal syntax and that, therefore, can treat binding as a natural syntactic feature of expressions and let them operate accordingly.

We are just starting with the implementation of the type-checker in Haskell. We aim to have a first prototype ready for the conference.

REFERENCES

[1] Maribel Fernández, Miguel Pagano, Nora Szasz, and Alvaro Tasistro. Dependently sorted nominal signatures. In *Proceedings of LFMTP 2025: Logical Frameworks and Meta Languages: Theory and Practice.*, 2025.

[2] Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence.* PhD thesis, University of Cambridge, UK, March 2001.

[3] Murdoch J. Gabbay. Nominal terms and nominal logics: from foundations to meta-mathematics. In *Handbook of Philosphical Logic*, volume 17, pages 79–178. Kluwer, 2011.

[4] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science (LICS 1987)*, pages 194–204. IEEE Computer Society Press, New York, June 1987.

[5] Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Proc. 4th Int'l Symposium on Theoretical Aspects of Computer Software (TACS 2001)*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer, 2001.

[6] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal Unification. *Theoretical Computer Science*, 323(1–3):473–497, September 2004.

$$\text{(inf:atm)}\ \frac{\mathscr{T} \vdash_\Sigma \Gamma\ \text{ctx-ok}}{\mathscr{T};\Gamma \vdash_\Sigma a \Rightarrow \Gamma(a)}\ a \in \mathrm{dom}(\Gamma) \qquad \text{(inf:var1)}\ \frac{\mathscr{T} \vdash_\Sigma \Gamma\ \text{ctx-ok}}{\mathscr{T};\Gamma \vdash_\Sigma X \Rightarrow \mathscr{T}(X)}\ X \in \mathrm{dom}(\mathscr{T})$$

$$\text{(inf:var2)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma X[\overline{t}] \Rightarrow\!\ll a : \mathcal{F}\ \overline{s}\gg \gamma \qquad \mathscr{T};\Gamma \vdash_\Sigma t' \Leftarrow \mathcal{F}\ \overline{s}}{\mathscr{T};\Gamma \vdash_\Sigma X[\overline{t}, t'] \Rightarrow \gamma[a \mapsto t']}$$

$$\text{(inf:constr)}\ \frac{\Sigma(f) = \mathscr{T}' \to \mathcal{F}\ \overline{u}; \Delta \quad \mathscr{T};\Gamma \vdash_\Sigma \overline{t} \Leftarrow\!\!\!\Leftarrow_\Delta \mathscr{T}'_\#[(\Delta_\#)_{\overline{t}}]}{\mathscr{T};\Gamma \vdash_\Sigma f\ \overline{t} \Rightarrow \mathcal{F}\ (\overline{u}_{\overline{t}})}$$

$$\text{(inf:abs)}\ \frac{\mathscr{T};(\Gamma, b : \mathcal{F}\ \overline{t}) \vdash_\Sigma (a\ b) \cdot t \Rightarrow (a\ b) \cdot \gamma}{\mathscr{T};\Gamma \vdash_\Sigma \ll a : \mathcal{F}\ \overline{t}\gg t \Rightarrow\!\ll a : \mathcal{F}\ \overline{t}\gg \gamma}\ \begin{cases} b \notin \mathrm{dom}(\Gamma) \\ b \,\#\, \{t, \gamma\} \end{cases}$$

Figure 2: ***Syntax-directed rules for sort inference.***

$$\text{(gnd:check)}\ \frac{\cdot;\Gamma \vdash_\Sigma t \Rightarrow \gamma}{\cdot;\Gamma \vdash_\Sigma t \Leftarrow \gamma'}\ \gamma \approx_\alpha \gamma' \qquad \text{(check)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma t \Rightarrow \gamma}{\mathscr{T};\Gamma \vdash_\Sigma t \Leftarrow \gamma}$$

Figure 3: ***Sort-checking rules.***

$$\text{(chk-empty-sig)}\ \frac{}{\vdash \langle\rangle\ \Leftarrow}$$

$$\text{(chk-sort-sig)}\ \frac{\vdash \Sigma\ \Leftarrow \quad \vdash_\Sigma \mathscr{T}\ \Leftarrow \quad \mathcal{F} \notin \mathrm{dom}(\Sigma) \quad \Delta\ \text{well-formed}}{\vdash \Sigma, \langle \mathcal{F} : \mathscr{T} \to \mathrm{data}\,;\, \Delta\rangle\ \Leftarrow}$$

$$\text{(chk-fun-sig)}\ \frac{\vdash \Sigma\ \Leftarrow \quad \vdash_\Sigma \mathscr{T}\ \Leftarrow \quad \mathscr{T};\cdot \vdash_\Sigma \mathcal{F}\ \overline{t} \Leftarrow \mathrm{sort} \quad \mathcal{F} \notin \mathrm{dom}(\Sigma) \quad \Delta\ \text{well-formed}}{\vdash \Sigma, \langle f : \mathscr{T} \to \mathcal{F}\ \overline{t}\,;\, \Delta\rangle\ \Leftarrow}$$

Figure 4: ***Rules for checking signatures.***

$$\text{(check-empty-tel)}\ \frac{}{\vdash_\Sigma \cdot\ \Leftarrow} \qquad \text{(check-cons-tel)}\ \frac{\vdash_\Sigma \mathscr{T}\ \Leftarrow \quad \mathscr{T};\cdot \vdash_\Sigma \gamma \Leftarrow \mathrm{sort}}{\vdash_\Sigma \mathscr{T}, (X : \gamma)\ \Leftarrow}\ X \notin \mathrm{dom}(\mathscr{T})$$

Figure 5: ***Rules for checking telescopes.***

$$\text{(check-empty-fit)}\ \frac{}{\mathscr{T};\Gamma \vdash_\Sigma ()\ \Leftarrow\!\!\!\Leftarrow_\Delta \cdot}$$

$$\text{(check-cons-fit)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma t \Leftarrow \gamma \quad \vdash \Delta|_X[X \mapsto t]\ \Leftarrow \quad \mathscr{T};\Gamma \vdash_\Sigma \overline{t'} \Leftarrow\!\!\!\Leftarrow_\Delta \mathscr{T}'[X \mapsto t]}{\mathscr{T};\Gamma \vdash_\Sigma t, \overline{t'} \Leftarrow\!\!\!\Leftarrow_\Delta (X : \gamma), \mathscr{T}'}\ X \notin \mathrm{dom}(\mathscr{T})$$

Figure 6: ***Rules for checking instantiations.***

$$\text{(chk-emp-ctx)}\ \frac{}{\mathscr{T} \vdash_\Sigma \cdot\ \Leftarrow} \qquad \text{(chk-cons-ctx)}\ \frac{\mathscr{T} \vdash_\Sigma \Gamma\ \Leftarrow \quad \mathscr{T};\Gamma \vdash_\Sigma \mathcal{F}\ \overline{t} \Leftarrow \mathrm{sort}}{\mathscr{T} \vdash_\Sigma \Gamma, (a : \mathcal{F}\ \overline{t})\ \Leftarrow}\ a \notin \mathrm{dom}(\Gamma)$$

Figure 7: ***Rules for checking contexts.***

$$\text{(chk-data)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \overline{t} \Leftarrow\!\!\!\Leftarrow_\Delta \mathscr{T}'_\#[(\Delta_\#)_{\overline{t}}] \quad \Sigma(\mathcal{F}) = \mathscr{T}' \to \mathrm{data}; \Delta}{\mathscr{T};\Gamma \vdash_\Sigma \mathcal{F}\ \overline{t} \Leftarrow \mathrm{sort}}$$

$$\text{(chk-abs-*)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \mathcal{F}\ \overline{t} \Leftarrow \mathrm{sort} \quad \mathscr{T};(\Gamma, b : \mathcal{F}\ \overline{t}) \vdash_\Sigma (a\ b) \cdot \gamma \Leftarrow \mathrm{sort}}{\mathscr{T};\Gamma \vdash_\Sigma \ll a : \mathcal{F}\ \overline{t}\gg \gamma \Leftarrow \mathrm{sort}}\ \begin{cases} b \notin \mathrm{dom}(\Gamma) \\ b \,\#\, \gamma \end{cases}$$

Figure 8: ***Rules checking sorts.***