A Multi-Stage Language with Intensional Analysis

Marcos Viera Alberto Pardo

Instituto de Computación
Universidad de la República
Montevideo - Uruguay
{mviera,pardo}@fing.edu.uy

Abstract

This paper presents the definition of a language with reflection primitives. The language is a homogeneous multi-stage language that provides the capacity of code analysis by the inclusion of a pattern matching mechanism that permits inspection of the structure of quoted expressions and their destruction into component subparts. Quoted expressions include an explicit annotation of their context which is used for dynamic inference of type, where a dynamic typing discipline based on Hinze and Cheney's approach is used for typing quoted expressions.

This paper follows the approach of Sheard and Pasalic about the use of the meta-language Ωmega as a tool for language design. In this sense, it is shown how to represent the syntax, the static as well as the dynamic semantics of the proposed language in terms of Ωmega constructs.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory – Semantics, Syntax; D.3.4 [*Programming Languages*]: Processors – Code Generation

General Terms Design, Languages, Theory

Keywords Reflection, Multi-stage Programming, Intensional Analysis, Dynamics

1. Introduction

With the evolution of computer systems and their growing complexity it has become more and more important to take into account the way to improve their flexibility. In order to provide systems with the ability to evolve during its own execution, programming languages should support *reflection*, understanding it as the ability to "reason about itself". Friedman and Wand [7] introduced the concepts of *reification* and *reflection* to define the processes of converting an interpreter component into an object which the program can manipulate and its inverse, respectively. For instance, one of the components that can be reified is the program code. This sort of reification can be performed by a *quotation* mechanism.

Multi-stage languages [23] are typed languages with quotation constructs, analogues to those of Lisp, which define execution stages. These constructs are bracket, escape and run. Brackets (`[|_|]`) reify the surrounded expression lifting it into the next stage. For example, while `2 + 2` evaluates to `4`, `[| 2 + 2 |]` evaluates to the representation of the piece of code `2 + 2`. Reified expressions can be constructed from others by using escape. Escape (`$(_)`) evaluates the expression surrounded to a code piece and then splices it into the expression where it occurs. Thus, the evaluation of `[| 2 + $([| 2 + 2 |]) |]` returns the representation of `2 + 4`. Run evaluates its arguments to a piece of code and executes it. This implements the inverse operation of reification.

It is essential to have reflection that programs can reason about their own state and manipulate it. According to Sheard [17] meta-programs can be classified into two categories: *analyzers* and *generators*. Program analyzers are an important class of meta-programs that can be used among other things to optimize, transform, maintain and reason about complex systems. Most of multi-stages languages, like MetaML [27], lay in the category of program generators. However, a formal treatment of program analyzers features has not being sufficiently developed.

In this paper, we propose a multi-stage language with *intensional analysis*, understanding intensional analysis as the ability of a homogeneous meta-system to observe the structure of its object-programs. This is carried out by a pattern matching mechanism that is used to inspect the structure of quoted expressions and destruct them into their component subparts.

In most multi-stage languages the type of quoted expressions is $\langle\tau\rangle$ (or `cod` τ), meaning code of τ, for τ the type of the expression being quoted. This typing statically ensures that dynamically generated programs are type-safe, but excludes some functions that destruct or traverse the structure of expressions. Other approaches [20, 8, 22] assign the same type `cod` to all quoted expressions, performing their type checking at run-time. Such languages make a tradeoff between static and dynamic typing. We follow these ideas using the techniques proposed by Cheney and Hinze [5] and Baars and Swierstra [2] for encoding dynamic typing. So, our language somehow relaxes static safety in favour of retaining flexibility.

Our type for quoted expressions is of the form $\texttt{cod}^{\Gamma}$, being Γ a type context reflecting, like Nanevsky [11, 12, 13] *names*, the free variables of the expression. When an expression is quoted, its type context needs to be explicitly annotated as it is necessary for dynamic type inference.

We follow the approach of Sheard and Pasalic [19, 18, 21, 14] about the use of Ωmega as a tool for language design. Languages are encoded as object-program representation that enforces the semantic invariants of scoping and typing rules. The type system of Ωmega then guarantees that all meta-programs respect these additional object-language properties. In the following subsection we briefly describe some of the Ωmega features we use, for further information see the mentioned works.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010. . . $5.00.

1.1 Ωmega

Ωmega is based on Haskell, although it is strict and doesn't have a class system. Some of its most important features are the so-called Generalized Algebraic Data Types and an extensible kind systems, which make it possible to state and enforce interesting properties of programs using the type system.

Generalized Algebraic Data Types (GADTs) are a generalization of Algebraic Data Types (ADTs). GADTs remove the restriction for parameterized ADTs which states that the range of every constructor must be a polymorphic instance of the type constructor being defined. This is possible by introducing an alternative syntax for data types declarations, where the type being defined is given an explicit kind, and every constructor is given an explicit type. For example, the type constructor `Term` has kind `*0 ~> *0`, taking types to types, and represents a typed object-language:

```
data Term :: *0 ~> *0 where
    Const :: a -> Term a
    Pair  :: Term a -> Term b -> Term (a,b)
    App   :: Term (a -> b) -> Term a -> Term b
```

The only restriction on constructors' type is that their range must be a fully applied instance of the type being defined. For example, the range of the constructor `Pair` is a non-polymorphic instance of `Term`. Observe that the type argument of `Term` is used to stand for the object level type of the represented term.

In the same way types classify values, types are classified by kinds. Kinds are implicit in functional languages like Haskell, and can only be either the *base kind* (`*0`), which classifies types, or *higher order kinds* (κ_1 `~>` κ_2), which classifies type constructors. In Ωmega, new kinds can be introduced by a `kind` declaration, which is analogous to a `data` declaration. Instead of introducing value constructors, a kind declaration introduces type constructors that produce types classified by that kind.

Sheard [18] proposes using Ωmega to explore the design of new languages by encoding language semantics as meta-programs. The language is defined as a GADT. Each GADT represents a judgment, and its constructors encode the typing rules. Type parameters may have an arbitrary structure, because of definition of new kinds, and correspond to static semantics properties. These properties are checked and maintained by Ωmega's type system. Ωmega's type system also guarantees that meta-level programs maintain object level type-safety. A *big step semantics* can be defined as an interpreter or evaluation function, or a *small step semantics* can be defined in terms of substitutions over the term language. The typing of this function maintains object level type-safety.

1.2 Structure of the paper

The paper is organized as follows. In Section 2 we introduce a language with reflection primitives and present its static semantics. Section 3 shows how the static semantics of the language is encoded in Ωmega. Section 4 describes a big-step semantics of the language in the form of an Ωmega function. We discuss related work in Section 5. Finally, Section 6 draws some conclusions.

2. Language

The aim of this paper is the proposal of a language with linguistic reflection primitives that permit us to perform type-safe intensional code analysis. In this section we define the syntax and static semantics of that language.

2.1 Basic Calculus

The core of the language is a Church-style [1] simply typed λ-calculus, with the following syntax:

Types	$\tau \in \mathsf{T}$	$::=$	$\mathtt{int} \mid \mathtt{bool} \mid \tau \to \tau \mid \tau \times \tau$
Contexts	$\Gamma \in \mathsf{G}$	$::=$	$\cdot \mid \Gamma, \tau$
Ctxt. Stacks	$P \in \mathsf{GS}$	$::=$	$\cdot \mid (P, \Gamma)$
Variables	$v \in \mathsf{V}$	$::=$	$\mathtt{z} \mid \mathtt{s}\ v$
Terms	$e \in \mathsf{E}$	$::=$	$b \mid i \mid (e_1, e_2) \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \mid$ $\lambda^\tau.\, e \mid \mathtt{fix}\ e \mid v^\tau \mid e_1\ e_2 \mid$ $\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$

A type τ can be a base type `int` or `bool`, a function type $\tau \to \tau$ or a binary product $\tau \times \tau$. De Bruijn indices [6] are used to encode variables bindings, so variables are natural numbers and type contexts are sequences of types.

The typing judgment is of the form $P; \Gamma \vdash e : \tau$, and reads "expression e has type τ in local context Γ under stack P". The presence of context stacks in the typing rules of Figure 1 is the only difference from standard λ-calculus, but can be ignored until we explain their use in the multi-stage extension.

$$\frac{}{P; \Gamma \vdash i : \mathtt{int}}\ \text{LInt} \qquad \frac{}{P; \Gamma \vdash b : \mathtt{bool}}\ \text{LBool}$$

$$\frac{P; \Gamma \vdash e_1 : \tau_1 \quad P; \Gamma \vdash e_2 : \tau_2}{P; \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \text{Pair}$$

$$\frac{P; \Gamma \vdash e : \tau_1 \times \tau_2}{P; \Gamma \vdash \mathtt{fst}\ e : \tau_1}\ \text{Fst} \qquad \frac{P; \Gamma \vdash e : \tau_1 \times \tau_2}{P; \Gamma \vdash \mathtt{snd}\ e : \tau_2}\ \text{Snd}$$

$$\frac{P; \Gamma, \tau_1 \vdash e : \tau_2}{P; \Gamma \vdash \lambda^{\tau_1}.e : \tau_1 \to \tau_2}\ \text{Abs} \qquad \frac{P; \Gamma, \tau \vdash e : \tau}{P; \Gamma \vdash \mathtt{fix}\ e : \tau}\ \text{Fix}$$

$$\frac{P; \Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad P; \Gamma \vdash e_2 : \tau_2}{P; \Gamma \vdash e_1\ e_2 : \tau_1}\ \text{App} \qquad \frac{P; \Gamma \vdash v : \tau}{P; \Gamma \vdash v^\tau : \tau}\ \text{Var}$$

$$\frac{P; \Gamma \vdash e_1 : \mathtt{bool} \quad P; \Gamma \vdash e_2 : \tau \quad P; \Gamma \vdash e_3 : \tau}{P; \Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}\ \text{Cond}$$

Figure 1. Basic λ-calculus typing

In the typing rules for variables, the Base rule projects the 0-th index type and the Weak rule for $\mathtt{s}\ n$ projects recursively the $(n+1)$-th type.

$$\frac{}{P; \Gamma, \tau \vdash \mathtt{z} : \tau}\ \text{Base} \qquad \frac{P; \Gamma \vdash v : \tau_1}{P; \Gamma, \tau_2 \vdash \mathtt{s}\ v : \tau_1}\ \text{Weak}$$

Figure 2. Typing rules for variables

2.2 Multi-stage Extension

We include some staging annotations as part of the language to build and combine pieces of code, partitioning the execution of programs into computational stages.

Types	$\tau \in \mathsf{T}$	$::=$	$\ldots \mid \mathtt{cod}^\Gamma$
Explicit Substitutions	$\Theta \in \mathsf{S}$	$::=$	$\Uparrow(\Theta) \mid e/ \mid \uparrow^\tau$
Terms	$e \in \mathsf{E}$	$::=$	$\ldots \mid \mathtt{[\|}e\mathtt{\|]\{}\Gamma\mathtt{\}} \mid \mathtt{\$}(e)^\tau \mid$ $e[\Theta] \mid \mathtt{run}\ e_1 \mathtt{\|} e_2$

Annotations include brackets, escape, run and explicit substitution. We don't include Cross-Stage Persistence in our language. Like in $reFL^{ect}$ [8, 9], this decision is based on the observations of Taha [23] that intensional analysis requires reductions not to be allowed in higher levels, which leads to a loss of confluence if cross-stage persistence is included.

The typing rules for the staged terms (Figure 3) are inspired by the "sliding band" of type contexts proposed by Sheard [18], except for the "future" stack of contexts which is unnecessary without cross-stage persistence. The "past" stack contains the contexts of the past stages that could be accessed when an escape is applied to the current context.

2.2.1 Dynamic Typing and Explicit Contexts

The type for quoted expressions is $\texttt{cod}^{\Gamma^f}$, where Γ^f is a type context reflecting the free variables of the expression. When an expression is quoted, a context including the free variables of the expression must be passed explicitly. Observe that Γ^f doesn't need to be minimal. That is, if Γ^p represents all free variables in the bracketed expression Γ^f must fulfill the relation $\Gamma^f = (\Gamma^p, \Gamma^c)$, meaning that all free variables in the expression have to be in Γ^f, but some others (Γ^p) could be added.

Unlike most multi-stage languages this type doesn't include the type of the expression, so the escape annotation judgment could type wrong formed expressions. For example the expression

$$(\lambda^{\texttt{cod}^{(\cdot)}}.\texttt{[|\$(\#0}^{\texttt{cod}^{(\cdot)}}\texttt{)}^{\texttt{int}}\texttt{|]\{}\cdot\texttt{\}) [| True |]\{}\cdot\texttt{\}}$$

is well typed, because the requirement that the bound code must be an integer expression cannot be checked statically. The type checking of this kind of expressions is deferred until run-time, and *ill-typed* quoted expressions evaluates to the well-typed value `[| Fail |]`.

The `run` annotation is similar to the one proposed in [22], where a run-time type checking and unification is done to decide if code expression is executed. In the Run rule, the type of the executed quoted expression e_1 must be the type of e_2, called the *exception expression*. If its type is not the expected one or type checking fails then e_2 is evaluated. The Run rule assures that only closed code can be evaluated by allowing only expressions with type $\texttt{cod}^{(\cdot)}$, that is, without free variables.

$$\frac{(P,\Gamma);\Gamma^f \vdash e : \tau}{P;\Gamma \vdash \texttt{[|}e\texttt{|]\{}\Gamma^f\texttt{\}} : \texttt{cod}^{\Gamma^f}}\ \text{Br}$$

$$\frac{P;\Gamma^p \vdash e : \texttt{cod}^{\Gamma}}{(P,\Gamma^p);\Gamma \vdash \texttt{\$(}e\texttt{)}^{\tau} : \tau}\ \text{Esc} \qquad \frac{P;\Gamma \vdash e_1 : \texttt{cod}^{(\cdot)} \quad P;\Gamma \vdash e_2 : \tau}{P;\Gamma \vdash \texttt{run}\ e_1\texttt{|}e_2 : \tau}\ \text{Run}$$

$$\frac{\Gamma' \vdash \Theta \Rightarrow \Gamma'' \qquad P;\Gamma \vdash e : \texttt{cod}^{\Gamma'}}{P;\Gamma \vdash e[\Theta] : \texttt{cod}^{\Gamma''}}\ \text{Subst}$$

Figure 3. Multi-stage extension typing

2.2.2 Explicit Substitution

An explicit substitution operator over quoted expressions is included in order to provide a simple way of capturing free variables. We use the notation for substitutions of $\lambda\nu$ [3], adding an explicit annotation of the new type in the case of shifting.

The typing judgment for substitutions is of the form $\Gamma \vdash \Theta \Rightarrow \Gamma'$. It relates a type context and a substitution with a "resulting" type context. Therefore, a substitution Θ over an expression typed in local context Γ results in an expression typed in local context Γ'. The typing rules are shown in Figure 4.

$$\frac{P;\Gamma \vdash e : \tau}{\Gamma, \tau \vdash e/ \Rightarrow \Gamma}\ \text{Slash} \qquad \frac{}{\Gamma \vdash \uparrow^{\tau} \Rightarrow \Gamma, \tau}\ \text{Shift}$$

$$\frac{\Gamma \vdash \Theta \Rightarrow \Gamma'}{\Gamma, \tau \vdash \Uparrow(\Theta) \Rightarrow \Gamma', \tau}\ \text{Lift}$$

Figure 4. Explicit substitution typing

Given an expression e of type τ in a local context Γ under any past stack P, a slash ($e/$) replaces the first variable by e and decrements the indexes of the remaining variables by one. Shift ($\uparrow^{\tau}$) increments the indexes of all variables by one and appends the type τ at index 0. Applying lift ($\Uparrow$), the 0-index type τ remains unchanged and the substitution Θ is applied to the rest of the context. For example, the expression

$$\begin{aligned}&(\texttt{[|\#0}^{(\texttt{int},\texttt{bool})\rightarrow\texttt{bool}}\ (\texttt{\#1}^{\texttt{int}}, \texttt{\#2}^{\texttt{bool}})\texttt{|]}\\ &\quad\texttt{\{bool, int, (int, bool)} \rightarrow \texttt{bool\}})\texttt{[}\Uparrow(9/)\texttt{]}\end{aligned}$$

would reduce to a code, with type $\texttt{cod}^{\cdot,\texttt{bool},(\texttt{int},\texttt{bool})\rightarrow\texttt{bool}}$, corresponding to the expression: $(\texttt{\#0}^{(\texttt{int},\texttt{bool})\rightarrow\texttt{bool}}\ (9, \texttt{\#1}^{\texttt{bool}}))$.

2.3 Intensional Analysis Extension

In order to provide *intensional code analysis* we extend the calculus with an alternation primitive, similar to the one proposed in [8], where variables are bound by a pattern matching mechanism.

Terms	$e \in \mathsf{E}$	::=	$\ldots \mid \lambda p.\, e_1 \mid e_2$
Patterns	$p \in \mathsf{P}$	::=	$i \mid b \mid (p_1, p_2) \mid \bullet^{\tau} \mid \_ \mid \texttt{[|}pc\texttt{|]}$
Code Patterns	$pc \in \mathsf{PC}$	::=	$\texttt{\$(}\bullet\texttt{)}^{\tau} \mid \texttt{\$(lit)}^{\tau} \mid \_ \mid \texttt{fail} \mid$
			$(pc_1, pc_2) \mid \texttt{fst}\ pc \mid \texttt{snd}\ pc \mid$
			$\lambda^{\tau}.\, pc \mid \texttt{fix}^{\tau}\ pc \mid$
			$v^{\tau} \mid pc_1\ pc_2 \mid$
			$\texttt{if}\ pc_1\ \texttt{then}\ pc_2\ \texttt{else}\ pc_3 \mid$
			$\texttt{[|\_|]\{}\Gamma\texttt{\}} \mid pc[\Theta] \mid$
			$\texttt{run}\ pc_1 \mid pc_2$

The semantics of patterns is inspired by the pattern matching mechanism defined by Pasalic and Linger [15]. In that work, a pattern judgment $\Gamma \vdash p : \tau \Rightarrow \Gamma'$ involves an "input" type context Γ, a pattern p, which should match a value of type τ, and a resulting type context Γ'. This context extends Γ with the types of the pattern variables. Based on the fact that the only change possible to an "input" context is its extension with the free variables of p, and in order to simplify the dynamic semantics of substitutions over alternations (see section 4.2), we had omitted the "input" context in the pattern judgment. So the judgment is of the form $\vdash p : \tau \Rightarrow \Gamma$, meaning that a pattern p (matching a value of type τ) has the free variables contained in Γ.

The Alt rule for an alternation of type $\tau_1 \rightarrow \tau_2$ relates a pattern p, which should match a value of type τ_1 extending a context by Γ', an expression e_1 with type τ_2 in local context Γ, Γ' (Γ extended with Γ'), and an alternative expression e_2 of type $\tau_1 \rightarrow \tau_2$. If p matches a value of type τ_1, then e_1 is evaluated in local context Γ, Γ', otherwise e_2 is evaluated in local context Γ and applied to the matched value.

The simplest pattern is the pattern *any* (_) which matches any value of type τ and leaves the context unchanged. Another basic

$$\frac{\vdash p : \tau_1 \Rightarrow \Gamma' \quad P;\Gamma,\Gamma' \vdash e_1 : \tau_2 \quad P;\Gamma \vdash e_2 : \tau_1 \to \tau_2}{P;\Gamma \vdash \lambda p.\ e_1 \,|\, e_2 : \tau_1 \to \tau_2}\ \text{Alt}$$

Figure 5. Alternation typing

pattern is the (nameless) *variable binding pattern* ($\bullet^\tau$), which differs from the previous one in the type annotation and the extension of the context binding the value matched. More than one variable in a pattern could be bound. The PPair rule shows how variables are related to the indexes in the resulting context. Given a pair pattern (p_1, p_2), where p_1 and p_2 are related to Γ' and Γ'' respectively, its free variables are Γ', Γ''. So the variables of the furthest to the right subpattern (p_2) would be those of smaller indices in the context. This can be taken as a general rule for patterns with multiple variables.

$$\frac{}{\vdash \_ : \tau \Rightarrow \cdot}\ \text{PAny} \qquad \frac{}{\vdash \bullet^\tau : \tau \Rightarrow \cdot, \tau}\ \text{PVar}$$

$$\frac{}{\vdash i : \texttt{int} \Rightarrow \cdot}\ \text{PLInt} \qquad \frac{}{\vdash b : \texttt{bool} \Rightarrow \cdot}\ \text{PLBool}$$

$$\frac{\vdash p_1 : \tau_1 \Rightarrow \Gamma' \quad \vdash p_2 : \tau_2 \Rightarrow \Gamma''}{\vdash (p_1, p_2) : \tau_1 \times \tau_2 \Rightarrow \Gamma', \Gamma''}\ \text{PPair}$$

$$\frac{\Gamma^f \Vdash pc \Rightarrow \Gamma'}{\vdash \texttt{[|}pc\texttt{|]} : \texttt{cod}^{\Gamma^f} \Rightarrow \Gamma'}\ \text{PCod}$$

Figure 6. Pattern typing

Code analysis is carried out with the help of *code patterns*. Their typing rules are shown in Figure 7. The judgment $\Gamma^f \Vdash pc \Rightarrow \Gamma'$ expresses that a pattern pc, which should match a quoted expression with type $\texttt{cod}^{\Gamma^f}$, has the variables contained in Γ'.

Most code patterns consist in destructing the expression and applying code patterns to the subexpressions. The *any* (`_`) code pattern matches any code, while the *fail* (`fail`) code pattern matches only failed code. Both patterns leave the context unchanged.

The syntax of *variable binding* ($\texttt{\$(}\bullet\texttt{)}^\tau$) and *literal binding* ($\texttt{\$(lit)}^\tau$) code patterns suggest their semantics in the sense that they only match with expressions that when unquoted have type τ. Having fulfilled this constraint the former matches any value while the latter matches only quoted literal expressions. Both patterns extend the context with the code value matched.

The *variable constant* behaves like *literal constants* (i and b). It matches with code which quoted expression is exactly the variable v^τ, unchanging the context. The *any bracket* code pattern matches any brackets quoted expression with free variables Γ^{ff}. Given a quoted explicit substitution, $e[\Theta']$, the *substitution* code pattern requires Θ to be equal to $Theta'$ and matches the code pattern pc with e.

An example of code analysis is the following:

$$\begin{array}{l}\lambda\texttt{[|if }\#0^{\texttt{bool}}\texttt{ then \$(}\bullet\texttt{)}^{\texttt{int}}\texttt{ else \$(}\bullet\texttt{)}^{\texttt{int}}\texttt{|]} \\ \quad .\quad \#1^{\texttt{cod}^{\cdot,\texttt{bool}}}\texttt{[}True\texttt{/]} \\ \quad |\quad \lambda^{\texttt{cod}^{\cdot,\texttt{bool}}}.\texttt{[|0|]\{}\cdot\texttt{\}}\end{array}$$

$$\frac{}{\Gamma^f \Vdash \_ \Rightarrow \cdot}\ \text{PCPAny} \qquad \frac{}{\Gamma^f \Vdash \texttt{fail} \Rightarrow \cdot}\ \text{PCPFail}$$

$$\frac{}{\Gamma^f \Vdash \texttt{\$(}\bullet\texttt{)}^\tau \Rightarrow \cdot, \texttt{cod}^{\Gamma^f}}\ \text{PCPVar}$$

$$\frac{}{\Gamma^f \Vdash \texttt{\$(lit)}^\tau \Rightarrow \cdot, \texttt{cod}^{\Gamma^f}}\ \text{PCPLit}$$

$$\frac{}{\Gamma^f \Vdash i \Rightarrow \cdot}\ \text{PCLInt} \qquad \frac{}{\Gamma^f \Vdash b \Rightarrow \cdot}\ \text{PCLBool}$$

$$\frac{\Gamma^f \Vdash pc_1 \Rightarrow \Gamma' \quad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma''}{\Gamma^f \Vdash (pc_1, pc_2) \Rightarrow \Gamma', \Gamma''}\ \text{PCPair}$$

$$\frac{\Gamma^f \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash \texttt{fst}\ \ pc \Rightarrow \Gamma'}\ \text{PCFst}$$

$$\frac{\Gamma^f, \tau \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash \lambda^\tau.pc \Rightarrow \Gamma'}\ \text{PCAbs} \qquad \frac{\Gamma^f, \tau \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash \texttt{fix}^\tau\ .pc \Rightarrow \Gamma'}\ \text{PCFix}$$

$$\frac{\Gamma^f \Vdash pc_1 \Rightarrow \Gamma' \quad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma''}{\Gamma^f \Vdash pc_1\ pc_2 \Rightarrow \Gamma', \Gamma''}\ \text{PCApp} \qquad \frac{}{\Gamma^f \Vdash v^\tau \Rightarrow \cdot}\ \text{PCVar}$$

$$\frac{\Gamma^f \Vdash pc_1 \Rightarrow \Gamma' \quad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma'' \quad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma'''}{\Gamma^f \Vdash \begin{array}{l}\texttt{if }\ pc_1\ \texttt{ then }\ pc_2 \\ \texttt{else }\ pc_3 \Rightarrow \Gamma', \Gamma'', \Gamma'''\end{array}}\ \text{PCCond}$$

$$\frac{}{\Gamma^f \Vdash \texttt{[|\_|]}\{\Gamma^{ff}\} \Rightarrow \cdot}\ \text{PCBr}$$

$$\frac{\Gamma^f \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash pc[\Theta] \Rightarrow \Gamma'}\ \text{PCSubst}$$

$$\frac{\Gamma^f \Vdash pc_1 \Rightarrow \Gamma' \quad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma''}{\Gamma^f \Vdash \texttt{run }\ pc_1 \,|\, pc_2 \Rightarrow \Gamma', \Gamma''}\ \text{PCRun}$$

Figure 7. Code Pattern typing

This expression takes a code value with type $\texttt{cod}^{\cdot,\texttt{bool}}$ and returns one with type $\texttt{cod}^{(\cdot)}$. If the code passed is a quotation of an "if-then-else" expression, with condition $\#0^{\texttt{bool}}$, a code with the "then"" subexpression is returned, with a True literal in each occurrence of the variable $\#0^{\texttt{bool}}$. Otherwise, the returned value is a code of the literal 0.

3. Static Semantics as an Ωmega GADT

In this section we will encode the typing judgments of section 2 as Ωmega GADTs. A value of each datatype then represents a derivation of the encoded judgment. This ensures that the properties of the static semantics are checked and maintained by the meta-language type system.

The expression judgment $P;\Gamma \vdash e : \tau$ is represented by the multiple indexed type `(Exp p n t)`. The "past" stack P is tracked by the first index, a nested product type, which contains types of

kind `Row *0` [1]representing type contexts. The next index is a `Row *0` tracking the current context type Γ. Finally, `t` tracks the term type τ.

The Ωmega encoding of the rules showed in Figures 1, 3 and 4 is the following:

```
data Exp :: *0 ~> Row *0 ~> *0 ~> *0 where
    ELBool  :: Bool -> Exp p n Bool
    ELInt   :: Int -> Exp p n Int
    EPair   :: Exp p n t -> Exp p n s
            -> Exp p n (t,s)
    EPFst   :: Exp p n (t,s) -> Exp p n t
    EPSnd   :: Exp p n (t,s) -> Exp p n s
    EAbs    :: Rep s -> Exp p (RCons s n) t
            -> Exp p n (s->t)
    EFix    :: Exp p (RCons t n) t
            -> Exp p n t
    EApp    :: Exp p n (s->t) -> Exp p n s
            -> Exp p n t
    EVar    :: Var n t -> Rep t
            -> Exp p n t
    ECond   :: Exp p n Bool
            -> Exp p n t -> Exp p n t
            -> Exp p n t
    EBr     :: Exp (p,Env n) c t -> RepEnv c
            -> Exp p n (Cod c)
    ERun    :: Exp p n (Cod RNil) -> Exp p n t
            -> Exp p n t
    EEsc    :: Exp p b (Cod n) -> Rep t
            -> Exp (p, Env b) n t
    ESubst  :: Exp p n (Cod f) -> Subst f fc
            -> Exp p n (Cod fc)
    EAlt    :: Pat s c -> Exp p {eapp c n} t
            -> Exp p n (s->t)
            -> Exp p n (s->t)
```

Each constructor has the structure of a formal judgment. For example, `EApp` takes two arguments `Exp p n (s->t)` and `Exp p n s`. These arguments correspond to the judgments $P; \Gamma \vdash e_1 : \tau_2 \to \tau_1$ and $P; \Gamma \vdash e_2 : \tau_2$, respectively. If these can be supplied, the constructor results in the type `Exp p n t`, encoding $P; \Gamma \vdash e_1\ e_2 : \tau_2$.

In `EAbs`, `EVar` and `EEsc` a type must be annotated. This is done by an argument of type `Rep t`, the parametric type representation defined both by Cheney and Hinze [5] and Baars and Swierstra [2] for dynamic typing:

```
data Rep:: *0 ~> *0 where
    Int     :: Rep Int
    Bool    :: Rep Bool
    Arr     :: Rep a -> Rep b -> Rep(a -> b)
    Prod    :: Rep a -> Rep b -> Rep (a,b)
    Cod     :: RepEnv n -> (Rep (Cod n))
```

These type annotations are used to carry out the run-time type checking in the same way dynamic typing is handled in the works mentioned previously.

The `EVar` constructor includes the `Var n t` sub-judgment, where `VZ` and `VS` encode the rules Base and Weak of Figure 2.

```
data Var :: Row *0 ~> *0 ~> *0 where
    VZ      :: Var (RCons t env) t
    VS      :: Var env t -> Var (RCons s env) t
```

[1] `Row` is a kind that classifies list-like data structures at type level, its definition is: `kind Row (x::*1) = RCons x (Row x) | RNil`. So `Row *0` is a list of types that classifies values.

Observe that a context extension Γ, τ is represented by the `Row` constructor `(RCons t env)`.

The stacks of contexts are nested pairs. A type `Env`, which is indexed by a `Row *0`, is used to push a context. This is done because the pair constructor takes only types of kind *0.

```
data Env :: Row *0 ~> *0 where
    EnvNil  :: Env RNil
    EnvCons :: t -> Env r -> Env (RCons t r)
```

Multi-stage annotations involves expressions with type $\texttt{cod}^{\Gamma^f}$. The encoding of this type in Ωmega has the following definition:

```
data Cod :: Row *0 ~> *0 where
    Q   :: (forall p. Exp p n t) -> RepEnv n
        -> Cod n
    F   :: RepEnv n -> Cod n
```

Because of dynamic typing, it could happen that an expression evaluates to a bad formed code value. For this reason, the type `Cod` has two constructors: one for well formed quoted expressions and another for failed ones. A well formed code is an expression at level 0, typed in a given environment. A term at level 0 has no escapes at level 0. This is captured by requiring that the past contexts stack is universally quantified. Both in the case of well formed code like for failed code, a representation of the context is passed as an argument. This representation has type `RepEnv`.

```
data RepEnv:: Row *0 ~> *0 where
    REnvNil     :: RepEnv RNil
    REnvCons    :: Rep t -> RepEnv r
                -> RepEnv (RCons t r)
```

This type classifies lists of `Rep t` and is indexed by a `Row *0`. Type `RepEnv` is also used in the constructor `EBr` to represent the free variables of the expression.

The substitutions judgment is encoded by the datatype `Subst`. Like in the `Q` constructor for `Cod`, the expression passed to the `SSlsh` constructor must carry an universally quantified past contexts stack.

```
data Subst :: Row *0 ~> Row *0 ~> *0 where
    SSft    :: Rep t -> Subst n (RCons t n)
    SLft    :: Subst n c
            -> Subst (RCons t n) (RCons t c)
    SSlsh   :: (forall p. Exp p n t)
            -> Subst (RCons t n) n
```

So, the encoding for the explicit substitution example of section 2.2.2 is:

```
(ESubst
    (EBr
        (EApp
            (EVar VZ (Arr (Prod Int Bool) Bool))
            (EPair
                (EVar (VS VZ) Int)
                (EVar (VS VS VZ) Bool)))
        (REnvCons (Arr (Prod Int Bool) Bool)
            (REnvCons Int (REnvCons Bool REnvNil))))
    (SLft (SSlsh (ELInt 9))))
```

In the `EAlt` constructor we use the *type function* `eapp` to encode a list append constraint (Γ', Γ''). It can be proven by doing induction on the first argument that this function terminates.

```
eapp :: Row *0 ~> Row *0 ~> Row *0
{eapp RNil ys} = ys
{eapp (RCons x xs) ys} = RCons x {eapp xs ys}
{eapp {eapp xs ys} zs} = {eapp xs {eapp ys zs}}
```

The pattern judgment $\vdash p : \tau \Rightarrow \Gamma$ is encoded by the datatype `(Pat t n)`.

```
data Pat :: *0 ~> Row *0 ~> *0 where
    PLInt   :: Int -> Pat Int RNil
    PLBool  :: Bool -> Pat Bool RNil
    PPair   :: Pat t1 c1 -> Pat t2 c2
            -> Pat (t1,t2) {eapp c2 c1}
    PVar    :: Rep t -> Pat t (RCons t RNil)
    PAny    :: Pat t RNil
    PCod    :: PatCod f c -> Pat (Cod f) c
```

The constructor function `PCod` includes a sub-judgment for code patterns. The definition of the type `PatCod`, representing the code patterns judgment, is the following:

```
data PatCod :: Row *0 ~> Row *0 ~> *0 where
    PCPVar  :: Rep t
            -> PatCod f (RCons (Cod f) RNil)
    PCPAny  :: PatCod f RNil
    ...
    PCVar   :: Var vn t -> Rep t
            -> PatCod f RNil
    PCLInt  :: Int -> PatCod f RNil
    ...
    PCPair  :: PatCod f c1
            -> PatCod f c2
            -> PatCod f {eapp c2 c1}
    ...
    PCAbs   :: Rep s -> PatCod (RCons s f) c
            -> PatCod f c
    ...
    PCCond  :: PatCod f c1
            -> PatCod f c2
            -> PatCod f c3
            -> PatCod f {eapp3 c3 c2 c1}

    PCBr    :: RepEnv fp
            -> PatCod f (RCons (Cod f) RNil)
    PCRun   :: PatCod f c1 -> PatCod f c2
            -> PatCod f {eapp c2 c1}
    PCSubst :: PatCod fc c -> Subst f fc
            -> PatCod fc c
```

4. Dynamic Semantics as an Ωmega evaluator

Dynamic semantics for the language is given by a big-step semantics written as an evaluation function. The semantics shows that the evaluation of well typed terms doesn't go wrong.

The evaluation function has type `Exp p n t -> Env n -> t`. Given any well typed expression `Exp p n t` and an environment with shape `n`, `eval` returns a value with type `t`.

```
eval :: Exp p n t -> Env n -> t
eval (ELInt i) env = i
...
eval (EPair e1 e2) env = (eval e1 env,eval e2 env)
...
eval (EAbs t e) env = \ v -> eval e (EnvCons v env)
eval (EApp f x) env = (eval f env)(eval x env)
eval (EVar v t) env = evalVar v t env
...
eval (EFix e) env = lazy ((\ v ->
                        (eval e (EnvCons v env)))
                        (eval (EFix e) env))
eval (EBr e renv) env
            = case (bd (CountBrZ env) e) of
                (x,True) -> Q x renv
                (x,False) -> F renv
eval (ERun e1 e2) env
            = case (eval e1 env) of
                Q e REnvNil ->
                    case eqType (getType e)
                                (getType e2) of
                        Just Eq -> eval e EnvNil
                        Nothing -> eval e2 env
                _ -> eval e2 env
eval (ESubst e s) env
            = case (eval e env) of
                Q eb rb ->
                    case (evalSub s eb rb) of
                        (en,rn) -> Q en rn
                F rb -> F (evalSubR s rb)
eval (EAlt p e1 e2) env
            = \ v -> case (evalPat p v env) of
                Just env2 -> eval e1 env2
                Nothing -> (eval e2 env) v
```

This function is total excepting for de `EEsc` case, which is not evaluated. In an expression at level 0 will not be an escape, so the evaluation function must be defined to take expressions at level 0. This could be enforced defining an evaluation function that can only be applied to terms polymorphic in their past.

```
eval0 :: (forall p. Exp p n t) -> Env n -> t
eval0 exp env = eval exp env
```

To avoid infinite loops, the Ωmega construct for explicit laziness (`lazy`) is used in the evaluation of `EFix`.

4.1 Dynamic Type Checking and Building Code

The type checking is implemented by the unification function `eqType`, which takes two type representations, tests them for structural equality, and possibly returns a proof of their equivalence. Its signature is:

```
eqType :: Rep a -> Rep b -> Maybe(Equal a b)
```

During the evaluation of `ERun`, after a verification that the code is well formed, an unification between the types of the quoted expression `e` and the exception expression `e2` is made. If the unification succeeds, there's a witness that the type of `e` is the same as `e2`. So, the expression `e` is evaluated in the empty environment (static type-checking assures that `e` is closed). If the unification fails, the expression `e2` is evaluated in the environment `env`. The types of `e` and `e2` are obtained by the type inference function `getType`, which is based on the typing rules:

```
getType:: Exp p n t -> Rep t
getType (ELInt i) = Int
getType (ELBool b) = Bool
getType (EPair e1 e2) = Prod (getType e1)
                             (getType e2)
getType (EPFst e) = case (getType e) of
                        Prod r1 r2 -> r1
...
getType (EAbs t e) = Arr t (getType e)
getType (EApp e1 e2) = case (getType e1) of
                             Arr r1 r2 -> r2
getType (EVar v t) = t
...
getType (EBr e renv) = Cod renv
getType (ESubst e s) = case (getType e) of
                        Cod renv ->
                            Cod (evalSubR s renv)
```

```
getType (ERun e1 e2) = getType e2
getType (EEsc e t) = t
getType (EAlt p e1 e2) = getType e2
```

Observe that type annotations and the explicit type context `renv` are used in the type inference algorithm.

Evaluating `EBr` involves evaluating a code template in order to build an expression polymorphic in the past. This is done by the `bd` function, which is the one defined in [18] with the addition of dynamic type checking. Essentially, the function traverses an expression, generating a copy without embedded escapes at level 0, and a boolean expressing if the code produced is well typed. The `CountBr` argument counts the brackets surrounding the expression. When a `EBr` is found, the counter is incremented.

```
data CountBr :: *0 ~> *0 ~> *0 where
    CountBrZ :: a -> CountBr (b,a) c
    CountBrS :: CountBr a b -> CountBr (a,c) (b,c)

bd :: CountBr a z -> Exp a n t -> (Exp z n t,Bool)
bd env (ELInt i) = (ELInt i,True)
...
bd env (EPair e1 e2) =  let (x1,b1) = (bd env e1)
                            (x2,b2) = (bd env e2)
                        in (EPair x1 x2, b1 && b2)
...
bd env (EBr e renv)
        =   let (x,b) = bd (CountBrS env) e
            in (EBr x renv, b)
...
bd (CountBrZ env) (EEsc e t)
        = case (eval e env) of
            Q x renv ->
                case eqType (getType x) t of
                    Just Eq -> (x,True)
                    Nothing -> (getAny t,False)
            _ -> (getAny t,False)
bd (CountBrS r) (EEsc e t)
        =   let (x,b) = bd r e
            in (EEsc x t, b)
```

For `(EEsc e t)`, if brackets' counter is `(CountBrZ env)`, the expression `e` is evaluated and, if it is well formed and the type is what was expected, the resulting code is spliced. In other case a dummy expression with type `t` is generated by the function `getAny :: Rep t -> Exp p n t`.

4.2 Explicit Substitution

The explicit substitution evaluation is divided into two functions, which apply the substitution to the expression and to the context representation, respectively.

The core of the expression substitution is the one defined by Sheard and Pasalic [21], extended by passing the representation of the source environment.

```
evalSubE    :: Subst g gp -> Exp p g t -> RepEnv g
            -> Exp p gp t

evalSubE s (ELInt i) r = ELInt i
...
evalSubE s (EPair e1 e2) r = EPair (evalSubE s e1 r)
                                   (evalSubE s e2 r)
...
evalSubE s (EAbs t e) r = EAbs t
                (evalSubE (SLft s) e (REnvCons t r))
evalSubE (SSlsh e) (EVar VZ t) (REnvCons t r) = e
evalSubE (SSlsh e) (EVar (VS v) t) (REnvCons tp r)
                                        = EVar v t
evalSubE (SLft s) (EVar VZ t) (REnvCons t r)
                                        = EVar VZ t
evalSubE (SLft s) (EVar (VS v) t) (REnvCons tp r)
                    = evalSubE (SSft tp)
                        (evalSubE s (EVar v t) r)
                            (evalSubR s r)
evalSubE (SSft tp) (EVar v t) (REnvCons t r)
                                  = EVar (VS v) t
```

Evaluating the substitution over bracketed expressions implies evaluating a code template with a function, similar to the `bd` of section 4.1, which traverses the expression and applies the substitution when the *brackets counter* is zero.

```
evalSubE s (EBr e renv) r
                = EBr (bds (CountSBrZ s r) e) renv
evalSubE s (ESubst e sp) r
                = ESubst (evalSubE s e r) sp
```

To apply a substitution to the Alternation expression we take the following steps. First, we leave the pattern unchanged. Then, similarly as done in the Abstraction expression but in a more general case, we evaluate the effect of the pattern over the substitution and the context representation of the matched expression. Next we apply this new substitution to the matched expression with the new context representation. Finally, we evaluate the original substitution over the alternative expression.

```
evalSubE s  (EAlt p e1 e2) r
            = case (getType e2) of (Arr t1 t2)
                -> EAlt p (evalSubE (evalPatS p s)
                              e1 (evalPatR p r t1))
                          (evalSubE s e2 r)
```

Observe that the decision taken in section 2.3 of only including the extensions of type contexts in pattern judgments simplifies the definition of this operation. For example, the signature of the auxiliar function `evalPatS` is simply:

```
evalPatS :: (Pat t eout) -> Subst g gp
    -> Subst {eapp eout g} {eapp eout gp}
```

That is, taking a pattern p, with judgment $\vdash p : \tau_1 \Rightarrow \Gamma'$, and a substitution Θ, with judgment $\Gamma_1 \vdash \Theta \Rightarrow \Gamma_2$, evaluating the effect of p over Θ results in a substitution Θ' with judgment $\Gamma_1, \Gamma' \vdash \Theta' \Rightarrow \Gamma_2, \Gamma'$. That way, we isolate the effect of p from Γ_1 and Γ_2.

The function that applies the substitution to the context representation is separated in three cases. In the Slash case the first type is removed, in the Shift case the new type is appended at the beginning, and, in the Lift case, the first type is left unchanged and the substitution is applied recursively to the rest of the context.

```
evalSubR :: Subst g gp -> RepEnv g -> RepEnv gp

evalSubR (SSlsh e) (REnvCons t r) = r
evalSubR (SSft t) r = REnvCons t r
evalSubR (SLft s) (REnvCons t r)
                    = REnvCons t (evalSubR s r)
```

4.3 Pattern Matching

The evaluation of alternation `(EAlt p e1 e2)` is done by evaluating the pattern `p`, and `e1` or `e2` depending on the result of pattern matching. The pattern matching evaluation function, `evalPat`, has three arguments: a pattern judgment of type `(Pat t eout)`, a value of type `t`, to match with the pattern, and an input variable of type `Env ein`. If pattern matching succeeds, the function returns a

value (Just env), being env the extended environment (with type Env eapp eout ein), and e1 is evaluated in this environment. If matching fails, a Nothing value is returned, and e2 is evaluated in the current context.

For example, if i is passed when evaluating the pattern (PLInt i), the same environment passed as argument is returned. On the other hand, evaluating (PVar t) never fails, just returning the current environment extended with the value passed.

```
evalPat :: (Pat t eout) -> t -> Env ein
        -> Maybe (Env {eapp eout ein})

evalPat (PLInt i) v env = if (i==v)
                            then (Just env)
                            else Nothing
...
evalPat (PPair p1 p2) (v1,v2) env
            = case (evalPat p1 v1 env) of
                Just env1 ->  evalPat p2 v2 env1
                Nothing     -> Nothing
evalPat (PVar t) v env  = Just (EnvCons v env)
evalPat PAny v env  = Just env
evalPat (PCod p) v env  = evalCPat p v env
```

In the case of (PPair p1 p2), the pattern p1 is evaluated extending the current environment and then p2 is evaluated extending the environment returned by p1.

The code patterns are evaluated by the function evalCPat.

```
evalCPat    :: (PatCod f eout) -> (Cod f) -> Env ein
            -> Maybe (Env {eapp eout ein})

evalCPat (PCPVar t) e env
    = case e of
        Q v renv ->
            case (eqType (getType v) t) of
                Just Eq -> Just (EnvCons e env)
                Nothing -> Nothing
        _ -> Nothing
evalCPat PCPAny e env = Just env
evalCPat PCPFail e env
    = case e of
        F renv -> Just env
        Q eq renv -> Nothing
evalCPat (PCLit t)  e env
    = case e of
        Q (ELInt i) renv ->
            case (eqType Int t) of
                Just Eq -> Just (EnvCons e env)
                Nothing -> Nothing
        Q (ELBool b) renv ->
            case (eqType Bool t) of
                Just Eq -> Just (EnvCons e env)
                Nothing -> Nothing
        _ -> Nothing
evalCPat (PCLInt i)  e env
    = case e of
        Q (ELInt v) renv -> if (i==v)
                              then (Just env)
                              else Nothing
        _ -> Nothing
...
evalCPat (PCPair p1 p2) e env
    = case e of
        Q (EPair v1 v2) renv ->
            case (evalCPat p1
                 (eval (EBr v1 renv) env) env) of
                Just env1 -> evalCPat p2
                    (eval (EBr v2 renv) env) env1
                Nothing -> Nothing
        _ -> Nothing
...
evalCPat (PCAbs tx pb) e env
    = case e of
        Q (EAbs tvx vb) renv ->
            case (eqType tvx tx) of
                Just Eq -> evalCPat pb
                    (eval (EBr vb
                    (REnvCons tvx renv)) env) env
                Nothing -> Nothing
        _ -> Nothing
...
evalCPat (PCVar v t)  e env
    = case e of
        Q (EVar vv vt) renv ->
            case (eqType t vt) of
                Just Eq ->  if (eqVar v vv)
                               then Just env
                               else Nothing
                Nothing -> Nothing
        _ -> Nothing
...
evalCPat (PCBr r) e env
    = case e of
        Q (EBr ve renv2) renv ->
            case (eqType (Cod r) (Cod renv2)) of
                Just Eq -> Just env
                Nothing -> Nothing
        _ -> Nothing
...
evalCPat (PCSubst p s)  e env
    = case e of
        Q (ESubst ve vs) renv ->
            if (eqSubst s vs)
                then (evalCPat p
                    (eval (EBr ve renv) env) env)
                else Nothing
        _ -> Nothing
```

Consider the case of (PCAbs tx pb). If the value passed is a code (Q (EAbs tvx vb) renv) and tx represents the same type than tvx, the code pattern pb is evaluated to match with a quotation of vb with context renv extended by tvx ((REnvCons tvx renv)).

4.4 Soundness

The soundness of a type system with respect to the semantics means that, if a term is well-typed, then its evaluation either returns a value of same type or gives rise to an infinite reduction sequence. In other words, well-typed terms never go wrong. To prove soundness, subject reduction and progress must be proved. The former property means that reduction preserves typing while the latter means that programs which are well-typed are either values or can be further reduced (evaluation never gets stuck).

According to the type of the evaluation function, Exp p n t -> Env n -> t, the evaluation of any expression e satisfying the type judgment $P; \Gamma \vdash e : \tau$ yields, if it terminates, a value of type τ. This means that subject reduction is automatically ensured by Ωmega's type system.

Concerning progress, observe that every well-typed term of the language always matches one of the clauses of `eval`. Therefore, if the term is not a value, there is a reduction rule that is applicable to it.

5. Related Work

Our language is based on multi-stage languages like MetaML [23, 24, 27, 10, 26] and MetaOCaml [25, 4], with the incorporation of features presented in languages like Template Haskell [20], $reFL^{ect}$ [8, 9] and $\nu^{\Box}$ [11, 12, 13] with the aim of supporting intensional analysis in a flexible way.

Typing MetaML and MetaOCaml have static type checking, associating a type *code of* τ to the quotation of an expression of type τ. On the other hand, languages like Template Haskell, $reFL^{ect}$ and the one proposed in [22] associate a universal type *code* to all quotations. As a consequence, these languages need to perform a dynamic type-checking for generated code, excepting for Template Haskell which performs compile-time code generation. Our language follows the approach of [22]. We perform dynamic type-checking for generated code, avoiding run-time errors by the inclusion of an exception expression in the `run` construct.

In our language quoted open expressions are represented by annotating the type *code* with a type context, containing the types of the free variables. These variables can be captured by an explicit substitution mechanism provided by the language. This approach is similar to that of $\nu^{\Box}$, which uses names to represent free variables in quoted expressions.

Intensional Analysis Neither MetaML nor MetaOCaml are proposed as code analyzers, they focus on code generation and its optimization. Taha [23, 24] argued that by introducing β reduction at higher levels and code inspection the property of coherence is violated. Therefore there exists many optimizations that can only be applied to code at stage 0. Moreover, cross-stage persistence, one of the most distinguishing features of these languages, can not be present as well.

In Template Haskell code is represented by an algebraic data type, allowing its inspection. In contrast, our language uses a high-level pattern matching interface to intensional analysis, in the line of $\nu^{\Box}$ and $reFL^{ect}$. In $\nu^{\Box}$ pattern matching is only defined over the simply typed λ-calculus fragment of the language. Our pattern matching mechanism is similar to the one proposed in $reFL^{ect}$.

Ωmega for language design The use of Ωmega for developing the semantics of our language is inspired in the encoding of MetaML done by Sheard in [19].

6. Conclusions

In this paper we presented an homogeneous functional multi-stage language with support for intensional analysis. A pattern matching mechanism was defined as a high-level interface to perform code inspection. The type of quoted expressions reflects the free variables of the expression but not its type, which is inferred at run-time. Although ill-typed quoted expressions can be generated at run-time only well-typed generated code can be evaluated by `run`. An explicit substitution operator over quoted expressions was included too.

The proposed language may seem impractical due to its type annotations. However, like in [8] and [22], a type annotation algorithm from implicitly typed terms to annotated terms could be defined to avoid this. This algorithm would be essentially an extension of the Hindley-Milner type inference algorithm.

Static and Dynamic Semantics were represented in Ωmega by encoding the typing judgments as GADTs and defining a big-step semantics written as an evaluation function, respectively. Since the evaluation function has a case defined for any well-typed term, the Ωmega implementation of the semantics showed that the evaluation of well-typed terms doesn't go wrong.

References

[1] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, Vol. 2, chapter 2, pages 118-309. Oxford University Press, Oxford, 1992.

[2] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002.

[3] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, 1996.

[4] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-like inference for classifiers. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, volume 2986 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2004.

[5] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press, 2002.

[6] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[7] Daniel P. Friedman and Mitchell Wand. Reification: Reflection Without Metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348–355, 1984.

[8] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. Technical Report PRG-RR-03-16, Programming Research Group, Oxford University Computing Laboratory, October 2003.

[9] Sava Krstic and John Matthews. Semantics of the *refl*^{ect} language. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–42, 2004.

[10] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming*, pages 193–207, 1999.

[11] Aleksandar Nanevski. Meta-programming with names and necessity. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional programming*, pages 206–217, New York, NY, USA, 2002. ACM Press.

[12] Aleksandar Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, School of Computer Science, Carnegie Mellon University, June 2004.

[13] Aleksandar Nanevski and Frank Pfenning. Staged computation with names and necessity. *Journal of Functional Programming*, 15(6):893–939, 2005.

[14] Emir Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health and Sciences University, The OGI School of Science and Engineering, 2004.

[15] Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, pages 136–167. Springer, 2004.

[16] T. Sheard, Z. Benaissa, and M. Martel. *Introduction to Multistage Programming Using MetaML*. Pacific Software Research Center,

Oregon Graduate Institute, 2 edition, 2000.

[17] Tim Sheard. Accomplishments and Research Challenges in Meta-programming. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44. Springer-Verlag, 2001.

[18] Tim Sheard. Playing with type systems. Presented at GPCE 05 MetaOCaml Workshop, 2005.

[19] Tim Sheard. Putting curry-howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85. ACM Press, 2005.

[20] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop 02*, pages 1–16, oct 2002.

[21] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, pages 106–124, jul 2004.

[22] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In *Principles of Programming Languages (POPL 98)*, pages 289–302, January 1998.

[23] Walid Taha. *Multi-stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, November 1999.

[24] Walid Taha. A sound reduction semantics for untyped CBN mutli-stage computation. or, the theory of metaML is non-trival (extended abstract). In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00)*, pages 34–43, 2000.

[25] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

[26] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–37, New York, NY, USA, 2003. ACM Press.

[27] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.