TYPE THEORY AND FUNCTIONAL PROGRAMMING
A WORK PROPOSAL

Gustavo Betarte*	*gustun@cs.chalmers.se*
Ana Bove*	*bove@fing.edu.uy*
Juan José Cabezas[†]	*jcabezas@fing.edu.uy*
Guillermo Calderón[†]	*calderon@fing.edu.uy*
Cristina Cornes[‡]	*Cristina.Cornes@inria.fr*
Sylvia da Rosa[†]	*darosa@fing.edu.uy*
Daniel Fridlender*	*frito@cs.chalmers.se*
Verónica Gaspes*	*vero@cs.chalmers.se*
Eduardo Giménez[§]	*Eduardo.Gimenez@lip.ens-lyon.fr*
Alberto Pardo[✠]	*pardo@informatik.th-darmstadt.de*
Paula Severi	*severi@win.tue.nl*
Nora Szasz*	*nora@cs.chalmers.se*
Alvaro Tasistro*	*tato@cs.chalmers.se*
Federico Yemurenko[†]	*yemurenk@fing.edu.uy*

(†) Instituto de Computación, Facultad de Ingeniería, Universidad de la República. J. Herrera y Reissig 565 Piso 5, Montevideo, Uruguay.
(*) Dept. of Computing Science, Chalmers University of Technology, S-412 96, Gothenburg, Sweden. (Tel.: +46 31 7721000)
(‡) INRIA - Roquencourt, B.P. 105, 78153 Le Chesnay Cedex, France.
(§) LIP, URA CNRS 1398, ENS Lyon, France.
(✠) FG Praktische Informatik, Technische Hochschule Darmstadt, Magdalenenstr. 11c, 64289 Darmstadt, Germany.

Date: May 24, 1996.

Abstract

We propose a series of work areas related to type theory and functional programming. By type theory we mean the formulation of Martin-Löf's set theory using the theory of types as logical framework, extended with record types and subtyping. The areas presented are: the implementation of an environment for carrying out constructions in type theory, the systematic development of programming in type theory as a formal discipline and the formalization in type theory of the theory of functional programming languages. We also give a succint introduction to a theory of specifications and programs in which programs and their correctness proofs are separated but still can be built up simultaneously; we propose the implementation of a programming system for this theory too.

The group of people presenting this work proposal is defined by their common interest to work together in at least one of the areas proposed. There will be people working in each of these areas in the Plata region.

1. *Introduction*

We consider the formulation of Martin-Löf's set theory using the theory of types as logical framework [ML87, NPS90, CNSvS94]. This has been extended in [BT96] with dependent record types and subtyping. It is this extended theory to which we refer hereafter as type theory.

Type theory is originally introduced as a formal language for constructive mathematics and can also be seen as a functional programming language. Viewed in the latter way, it possesses a number of features that are not present in most other functional languages. One of them is that it is possible to form *families of types* on a given type. Thus types can be formed by applying such families to individuals. Further, one then has *dependent function types*, i.e. types $\alpha \rightarrow \beta$ where α is a type and β is a family of types on α. If f is a function of this type, then the type of its output fa depends on the input a —it is βa. In Martin-Löf's original theory, there are, in addition to the functional types, only ground types. There is first the type of sets, called *Set*: sets are inductively defined, i.e. they correspond to concrete data types of ordinary functional languages. Also as in functional languages, one can introduce new sets by giving their constructors. Further, for each set we have a type whose objects are the elements of the set in question. That is, we have a predefined family of types on *Set*. We could call this family *Elem* and then $Elem(A)$ would be the type of the elements of A. We usually prefer to omit the family in these cases and write just A for both the object of type *Set* and the type that it determines, letting the context decide on the appropriate reading.

In the theory just described one can also represent propositions. Propositions are understood as inductively defined by their introduction rules. This is a refinement of the constructive explanation of the notion of proposition: to know a proposition is to know what a proof of it is. Further, that a proposition holds means that a proof of it can be given.

Then one can read propositions as sets and their proofs as elements of those sets. Now it is the presence of families of types that allows to carry this interpretation over to propositions about generic individuals and, hence, makes it likely that type theory constitutes a language in which mathematics can be formalized. For instance, given a set A, $A \rightarrow A \rightarrow Set$ is the type of binary relations on A. Then, if R is such a relation, for each element x of A we have a set Rxx. Since each set determines a type, we can form here a family of types over A, namely $[x]Rxx$, where we use brackets for abstraction. Then $A \rightarrow [x]Rxx$ is the type of proofs that R is reflexive. This function type is usually written more perspicuously as $(x : A)Rxx$, that can be read: "for any x in A, Rxx".

As another example, consider the type $(x, y : A)Rxy \rightarrow Ryx$. A function of this type will produce a proof of Ryx given any two elements x, y of A and a proof of Rxy. In virtue of the given explanations, this is the same as proving that if Rxy holds then so does Ryx, for arbitrary x, y in A, i.e. the symmetry of R.

The extension with dependent record types makes it possible to represent also types of structures such as algebraic structures or abstract data types of which arbitrary sets may be specified as components. Dependent record types are, just like record types of ordinary programming languages, sequences of *fields* in which *labels* are declared as of certain types:

$$\langle \mathbf{L}_1 : \alpha_1, ..., \mathbf{L}_n : \alpha_n \rangle$$

But in dependent record types, the type α_i may depend on the preceding labels $\mathbf{L}_1$,...,$\mathbf{L}_{i-1}$. An example is the type of binary relations on a set:

$$\langle \mathbf{S} : Set, \mathbf{R} : \mathbf{S} \rightarrow \mathbf{S} \rightarrow Set \rangle$$

Subtyping is introduced by the record types: a record type ρ is a subtype of another record type σ, which we write $\rho \sqsubseteq \sigma$, if ρ contains every label in σ (and possibly more) and the types of the common labels are, respectively, in the inclusion relation.

Then, for the case in which record types are interpreted as types of algebraic structures, we have that every type of structure obtained by enrichment of a previously given type with further structure becomes a subtype of the original type. Thus, for instance, group is a subtype of monoid and, as a consequence, every function defined on monoids can be applied to any group. In particular, properties of monoids are also properties of groups, which allows to carry over to type theory a form of reasoning that is used straightforwardly in informal mathematics.

On the programming side, the possibility of expressing propositions about generic individuals allows to have specifications of programs as types. Then type checking serves as both proof checking and program verification. In type theory, type checking is decidable for a class of expressions sufficiently large in practice.

In this work proposal, we present three areas of work connected to both type theory and functional programming. The first one consists in the problem of implementing on machine an environment for carrying out constructions in type theory. When we stress the view of type theory as a logical theory we call such an environment a *proof editor* or *assistant*. When we view type theory as a programming language or theory of programs, we call the environment a *programming system*. This work is connected to functional programming not just because type theory can be seen as a functional programming language, but also because the system to be developed is a large application to be programmed in a functional language.

The second area of work consists in the systematic development of programming in type theory, as a formal discipline. The natural reference for such a development is the series of techniques studied in functional programming, which we would like to reconstruct using type theory.

The third area uses type theory as a formal language for constructive mathematics and consists in the formalization of the theory of functional languages. This includes the formalizations of type systems and dynamic semantics and of the relations thereof. The main motivation behind this work is that the formalization in type theory of the relevant parts of the theory of functional languages gives implementations of these languages that are correct by construction, due to the constructive nature of type theory and, in particular, to the correspondence between proofs and programs.

These three areas of work are described with more detail in the first three sections that follow.

In the last section we introduce a fourth area of work, connected to a theory of specifications and programs, introduced in [Sza95]. This theory is motivated by the fact that the identification of proofs and programs in type theory makes programs contain large parts which are useful only for establishing the correctness of the programs, but not to perform computations. Opposite to that, we can consider programs to be just as those of ordinary functional languages while proofs are completely separated from programs. With this view, we can formulate a theory in which program construction and program verification are to be carried out as separate activities. But this is unsatisfactory too, since usually the same principles are used to guide the construction of both program and proof. We would like to have a theory with rules allowing the simultaneous construction of programs and proofs, while keeping them separate. The theory of specifications and programs that we consider here is precisely one such theory.

The main work we would like to carry out in connection to this theory is the implementation of a programming system for it.

The group of people presenting this work proposal is defined by their common interest to work together in at least one of the areas described. Also, there will be members of the group working in each of these areas in the Plata region, either because they already reside there or because they plan to do so in the near future.

2. *Implementation of a Proof Editor for Type Theory*

A proof editor, roughly, consists of a type checking algorithm embedded in an interactive environment which helps the user in the process of proof (program) construction. This assistance ranges from providing tactics which help the user in the refinement and completion of proofs up to a window interface. In the last years several proof editors have been developed for theories related to type theory. Among others, there is ALF [AGNvS94] based on a variant of type theory and Coq [DFH$^+$91] and LEGO [LP92] based on the Calculus of Constructions [CH86].

We are interested in the construction of a proof editor for the framework described in the introduction. The major features that distinguish type theory in our sense from the theory underlying ALF are:

- The only functions that are recursively defined are operators associated to each set that express the general principle of definition of functions on that set by primitive recursion. Any other function is to be introduced by an abbreviatory definition. This enforces termination of every program.
- The presence of (dependent) record types and subtyping.
- Facilities like definitions of types and type families and local definitions

Actually the first step in this work is to complete a precise formulation of the theory. Once this is done we can face the design and implementation of a type-checker. It is crucial for the whole system to be of any use that the actual implementation of the type-checker is correct. We would then like to get as close as possible to a full proof of its correctness. For this, it is important to have a fully understood formulation of the theory and a careful design of the type-checker.

The next step after that would be the design and implementation of editing facilities for the construction and manipulation of proofs including a command for the automatic derivation of the recursion operator for each set that is introduced.

Related to the latter point is the study of how to adapt the algorithms already known for compiling pattern matching in functional languages to the context of type theory. The aim is to provide the proof editor with a syntax for terms closer to that of functional languages. In the context of type theory, compilation of pattern matching must check that patterns are complete, as programs should be total. This task may be difficult in the context of dependent types. Some work has already been done in this direction for the Coq system [MS94] and other related systems. We propose to study how to extend these works to a larger class of patterns (in particular to patterns of dependent types) in order to develop heuristics to compile definitions in type theory using recursion and pattern matching into primitive recursion operators.

Summarizing, the interesting problems that we now have detected in connection to the design of proof editors for type theory are: the necessity to deal with extensions of the theory and the necessity of obtaining a design of the type-checker that makes its verification as simple as possible.

Experimental implementations of variants of the system we are planning are currently being developed both at the Department of Computing Science at Chalmers University, Gothenburg, Sweden and at the Laboratorio de Ciencia de la Computación at the Universidad de la República in Montevideo, Uruguay. The latter includes so far a type-checker for Martin-Löf's framework and a mechanism for introducing definitions. The present version at Gothenburg is already checking the extended framework with record types and subtyping, but a more careful design of the checking algorithm is under process.

The two systems have been completely implemented in Haskell [Hud92], and programming techniques such as parsing combinators, monads and continuations-based I/O have been extensively exploited. The implementors of the system at Gothenburg greatly profited from the interaction with the Functional Programming Group working at the same department. It has been this experience which has strongly convinced the members of this project of the convenience of having a solid research group in functional programming to work close to.

The study, design and implementation of a proof editor for the framework under consideration is a research topic by itself, nevertheless it is also fundamental for the experimentation with programming logics described in Section 3 and Section 4. Further, that the proof editor is supposed to be extensively used enforces the whole system to be eventually thought of as a user-friendly product. This entails that the design and implementation task should consider not only the construction of the proof engine but also questions such as syntax-directed edition of proofs, modularization of theories and interfaces, among others. Therefore, people coming from different working groups – theory of languages, functional programming, type theory, computer graphics – would naturally cooperate in the development of this project.

3. *Programming in Type Theory*

Type theory provides a framework within which it is in principle possible to develop programming as a formal discipline. However, no attempt has been made to carry out this development in a systematic way.

Particular examples of program construction can be found in a number of works, mostly using early versions of the theory or extensions thereof. Thus, for instance, [Bje85] studies the construction of sorting algorithms. In [Tho91] a number of examples are presented, which illustrate the most significant features of type theory as a functional programming language. Similar thing is done in [NPS90]. Classical examples, like the majority voting algorithm are considered in [BCMS89] and a parsing algorithm is given in [Chi87]. Finally, [Tas95] has studied the formal derivation of algorithms of sorting by insertion.

We propose to undertake the systematic and formal development of programming in type theory. A natural initial reference to such a programme is given by the series of techniques studied in functional programming, which we would then like to reconstruct using type theory. This defines a first big area of work, including

- the study of recursion schemes,
- polymorphism and overloading,
- programming with lists and other concrete data types,
- abstract data types,
- algorithms on classical data structures,
- continuations and monads,
- laziness and infinite objects,
- interactive programs.

This list includes well known techniques as well as problems that have only recently started to be tackled in functional programming as, for instance, the development of efficient algorithms as those studied in classical textbooks on data structures. Even for the better known techniques, their reconstruction in type theory constitutes an interesting problem, due to the various distinctive features of type theory as a functional programming language. The most important of these features are:

1. The possibility of writing full specifications as types.
2. Programs are constructed together with their correctness proofs.
3. All programs terminate and, in principle, only primitive recursion is available.

It is precisely because of 1) and 2) that type theory is to be counted among the theories of formal programming. Then the general question of the development of derivation techniques becomes interesting. Included in this question are, for instance, the intertwinning of program verification and program derivation and the analysis of techniques studied in other systems of program derivation, such as those developed for imperative programs by Dijkstra and others or those of program transformation.

Also, 1) makes it possible in principle to give more perspicuous specifications of general (higher order) functions, as for instance the various recursion operators on concrete data types as well as of data types themselves. See for instance the following example:
We are used to the typing of the recursion operator on lists $reduce : b {\rightarrow} (a {\rightarrow} b {\rightarrow} b) {\rightarrow} [a] {\rightarrow} b$
In type theory we can make its type show that the type of the result depends on the third argument (the list) and that we do some form of recursion on it:

$$
\begin{aligned}
listrec : & (A : Set) \\
& (B : (List(A))Set) \\
& (x : B([\,])) \\
& (f : (y : A)(ys : List(A))(h : B(ys))B(y{::}ys)) \\
& (l : List(A)) \\
& B(l)
\end{aligned}
$$

As to the specification of data types, one could for instance define the type of lists of a given length. Then operations like for instance "zipWith" can be specified as respecting the length of their arguments. This can be done as follows:

$$
\begin{aligned}
& \mathit{Vec} : Set \rightarrow N \rightarrow Set \\
& \mathit{emptyVec} : (A : Set)\, \mathit{Vec}\, A\, 0 \\
& \mathit{inhabVec} : (A : Set)(n : N) A \rightarrow \mathit{Vec}\, A\, n \rightarrow \mathit{Vec}\, A\, (n+1)
\end{aligned}
$$

Vec is a new datatype (a set) parameterized over another datatype (for the elements in the vectors) and a natural number (for the length of the vectors); it has two constructors *emptyVec* and *inhabVec*. Then we can type

$$
zipWith : (A, B, C : Set)(n : N)(A \rightarrow B \rightarrow C) \rightarrow \mathit{Vec}\, A\, n \rightarrow \mathit{Vec}\, B\, n \rightarrow \mathit{Vec}\, C\, n
$$

It is then in the type of *zipWith* that it takes lists of equal length and produces a list of this same given length. This idea can be extended to other notions of invariants in datastructures (height in trees for example or, as in [vS96], more elaborate properties in binary queues).

Finally, abstract data types can be specified together with the properties required on their components. Examples of this can be seen in [NPS90] and [Tas95].

The full exploitation of these possibilities is then a part of the question when studying programming in type theory as different from programming in ordinary functional languages.

Also feature 3) above introduces novel questions: foremost, is how to represent more general recursion schemas, i.e. course-of-values recursion and, since all programs must be terminating, the corresponding termination proofs. Different techniques can be studied in this connection.

Finally, lazy functional languages provide for programming techniques to which, in principle, we do not have access in type theory, namely the introduction of potentially infinite objects.

How to deal with this notion of infinite object in type theory is matter of study.

In undertaking this programme we expect to learn about formal programming techniques at the same time that we put type theory to test as a programming theory. The programme includes the actual implementation on machine of the programs that are to be developed, using some of the editors available or to be constructed. So, among the expected results of this work are ideas of convenient extensions of the theory as well as of desirable features of programming systems for it.

Also, we should obtain material and experience useful for educational purposes, particularly for putting together a gentle introduction to the subject.

4. *Formalized Theory of Functional Languages*

We are interested in the application of type theory to the theory of programming languages. Viewing type theory as a formal language for constructive mathematics we can then be said to be interested in investigating constructive formalizations of the mathematics of programs.

Now, constructive type theory can also be viewed as a programming language. Then the proofs of certain properties of the languages studied can be read off as algorithms playing a role in implementations of those languages.

In other words, the formalization of the relevant parts of the theory of programming languages gives implementations of these languages that are correct by construction. A main motivation of our work is to actually carry out this idea in practice, that is, to investigate the production of verified implementations of programming languages. This includes the actual implementation on machine of the whole development, using some of the editors available or to be constructed.

In general the procedure will involve describing in type theory syntax, static semantics and dynamic semantics of the language to be studied and proving properties of these.

The formalization of the semantics of programming languages has actually another application. It can be used to prove properties of particular programs written in the language —for instance that a given program terminates for all possible inputs or that two particular programs have the same semantics.

This can be useful not only for the results per se, but also to provide a formal setting where to carry out program transformation. In program transformation, rules are used to transform programs into other programs, preserving certain properties, most notably input/output behavior.

We are especially interested in the theory of functional programming languages.

A first experiment along the lines described has been conducted in [Bov95]. There, a small polymorphic functional language is considered. The type system of the language as well as its evaluation semantics are formalized and then a proof of the *subject reduction* property is constructed. Subject reduction is the main property relating the evaluation of expressions to their types. In this work, the proof assistant ALF [AGNvS94] has been used.

A related work is [Coq96] in which a formal proof is given of the correctness of an operational semantics of shared graph reduction with respect to a certain model. The proof is in an extension of type theory with co-inductive types and was constructed using the proof assistant ALF. Co-inductive types are types whose elements may be non well-founded, like, for instance, infinite streams.

Other works have considered the formalization of parts of the theory of lambda calculus, including typed lambda calculi. For instance [Coq96] presents a formalization in ALF of a proof of normalization of typed lambda calculus. Using a different theory, namely the calculus of constructions, [Hue93] has formalized a substantial part of the theory of untyped lambda calculus and [Pol94] has done the same for pure type systems.

Although closely related to each other, all the works cited present significant differences as to the choices made in the formalization. One such case is the treatment of bound variables. Text-books, i.e. informal presentations of the theories in question, typically make conventions as to the use of bound and free variables that simplify matters considerably. In a completely formal presentation, these conventions cannot be made or they have to be formalized as well. This introduces a range of different possible formalizations of at least the syntax of the language and of the type systems. Questions like this are not yet completely settled. They are of course important not just in this context, but also in that of the implementation of type checkers or proof assistants for type theory or similar languages.

After the first experiment in [Bov95], our next step in this work is naturally given by the formal development of the proof of existence of most general type schemes, alias the algorithm of type inference.

We would like then to progress towards covering the essential processes involved in the compilation of functional languages. At that stage, it would correspond to study abstract machines, be it environment machines or of graph reduction, and the corresponding processes of translation. The study of the theory of these abstract machines may raise questions as to the necessity or convenience of certain extensions to type theory. One example has already been mentioned, i.e. in [Coq96] co-inductive types have been used.

The latter extension is necessary for applying type theory to the verification of interactive and concurrent programs, where non-ending computations and other circular phenomena commonly arise.

An example of such an application is [Gim95]. There, proofs of correctness of a communicating protocol are constructed using the system Coq. Also, as a by-side product, a functional interpreter of processes written in *Haskell* is extracted from a proof of its specification.

5. *A Theory of Programs and Specifications*

Using type theory as a programming logic, one identifies specifications with sets and propositions and, correspondingly, programs with elements of sets and with proofs. Then, programs happen to contain something else than just what is needed to perform computations, namely information that is useful only for showing their correctness.

As an example, consider the following algorithm to divide a natural number into 2: for each n in N we compute two natural numbers x and y, such that $n = x{+}y$ and x and y are either equal or consecutive (this last property can be described by saying that $x \leq y \leq x{+}1$). Note that then, x is the result of dividing n by 2. We compute these numbers recursively on n: For $n{=}0$ it will be $x{=}y{=}0$. And, if x and y are the two numbers we obtain for n, then for $n{+}1$ they will be y and $x{+}1$, which still satisfy the property given above (here one can see why two numbers are needed: the first one is the result of the division, and the second one is used in the recursive step).

We can specify this task in Martin-Löf's set theory as the one of finding an element in the following set:

$$\forall n \epsilon N.\, \exists p \epsilon N \times N.\, (n{=}p_1{+}p_2) \;\&\; (p_1{\leq}p_2) \;\&\; (p_2{\leq}s(p_1))$$

The term we obtain is the following:

$$
\begin{array}{l}
\boldsymbol{\lambda}\mathbf{n.natrec(n} \\
\qquad\qquad (<\mathbf{0},\mathbf{0}>, <id(0), leqid(0), leqs(leqid(0))>) \\
\qquad\qquad \mathbf{x.i.}(<fst(\mathbf{i})_{\mathbf{2}}, \mathbf{s}(fst(\mathbf{i})_{\mathbf{1}})>, <idtr(s(fst(i)_1)+fst(i)_2, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad idsuc(snd(i)_1), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad addc(s(fst(i)_1), fst(i)_2)), \\
\qquad\qquad\qquad\qquad\qquad\qquad snd(i)_3, \\
\qquad\qquad\qquad\qquad\qquad\qquad sleq(snd(i)_2)>))
\end{array}
$$

We have marked the algorithm itself with bold characters. The rest corresponds to the proof of its correctness. The operator *natrec* corresponds to the schema of primitive recursion on the natural numbers. The rest of the constants stand for lemmas required to prove the program correct. Note that this term is very inefficient as a program for calculating the integral part of $n/2$ for a number n because each time it calculates the desired result, it also computes a proof that this result is correct. So, on the one hand we are interested in correct programs, that is, we want a formal proof of the correctness of the algorithm we have written. But we do not want to have to run the correctness proof each time we execute our program.

Now, if programs are to look like programs written in an ordinary functional programming language, then proofs must be defined separately. This motivates a calculus for program verification, in which the notion of set is split into two: the datatypes and the propositions. Programs are then elements of datatypes and do not contain any proof information. The correctness proofs are constructed separately from the programs.

In our example, the program performing the division task will be a function taking a natural number and yielding a pair of natural numbers. This function will be of the form:

$$f \equiv \lambda n.natrec(n, <0,0>, x.q.<q_2, s(q_1)>) \;\epsilon\; N \rightarrow N \times N$$

Then, to prove that this function does what we want, we will have to provide a proof of the following proposition:

$$\forall n \epsilon N.\ (n=(f\,n)_1+(f\,n)_2)\ \&\ ((f\,n)_1 \leq (f\,n)_2)\ \&\ ((f\,n)_2 \leq s((f\,n)_1))$$

and one such proof is the following term:

$$
\begin{array}{l}
\Lambda n.natind(n, \\
\qquad\qquad <id(0), leqid(0), leqs(leqid(0))>, \\
\qquad\qquad x.h.<idtr(s((f\,x)_1)+(f\,x)_2, idsuc(h_1), addc(s((f\,x)_1), (f\,x)_2)), \\
\qquad\qquad\qquad h_3, \\
\qquad\qquad\qquad sleq(h_2)>)
\end{array}
$$

where *natind* expresses the principle of natural induction.

If we now look at the program and the proof, we will see that they have the same structure: both are abstractions over a natural number and then recursion (respectively, induction) is done over this number. This is a very simple case and the similarities stop there. But in general, and with larger examples, the similarities continue, and we can see that there is some kind of *template* that the program and the correctness proof share. This template reflects the way we have reasoned to solve our problem. This points to a drawback of this methodology: we are making the same reasoning twice. That is, although we usually obtain the same skeleton for both the program and the correctness proof, we have to make the skeleton explicit twice.

Now compare this solution to the one we presented using Martin-Löf's set theory. In both cases we are interested in writing a program that is proved correct with respect to a given specification. If we look at terms as trees, we can see that in the first approach we build a tree, inside which there are hidden a program and a correctness proof. These, in turn, cannot easily be extracted out. In the second approach we build two trees, one for the program and one for the correctness proof, which have the same form. This tree structure represents what we called in the discussion above the template, or skeleton, reflecting the reasoning followed. We would like to have a framework featuring what is good about each of the former approaches. On the one hand, we want to be able to build programs and their correctness proofs at the same time, in the way it is done in type theory. We want, on the other hand, to keep programs and proofs separate.

With this motivation, we introduce the theory of specifications. A specification is a pair formed by a datatype and a predicate on this datatype. An implementation of such a specification is also a pair, formed by a program of the given datatype and a proof that the predicate holds for that program. Now, the theory is a calculus for program derivation, i.e. a calculus with rules that allow the simultaneous construction of the components of implementations, rather than the separate construction of programs and proofs. In the theory of specifications, operations are available that allow the construction of implementations using other implementations as building blocks. Let us have a closer look at the mechanisms of construction of implementations.

The only primitive way of forming specifications is by means of the specification forming operator Φ: given a datatype A and a predicate P over elements of A, a specification $\Phi x \epsilon A.\ P(x)$ is formed. It must be read as: find an x of type A and a proof that $P(x)$ holds. Correspondingly, implementations are constructed using a pairing operator $\langle\ ,\ \rangle$, in such a way that, given a program a of type A and a proof p of $P(a)$, the pair $\langle a\,,p\rangle$ is an implementation of $\Phi x \epsilon A.\ P(x)$. Then the rest of the constructs are defined (within the theory) in terms of the primitive ones. As a consequence, every implementation is identical to a pair of program and proof. To extract a program out of an implementation we need then only to compute the value of the latter and take the first projection of the result.

Let us for instance see how to introduce the rule allowing to construct implementations by primitive recursion on natural numbers: Let $S(n)$ be a specification depending on natural number n. We can construct an implementation of $S(n)$ provided we have an implementation b of $S(0)$ and a function i taking us from implementations of $S(x)$ to implementations of $S(s(x))$ for arbitrary natural number x. Let us introduce the operator nattac for expressing this construction. We would then say that $\mathsf{nattac}(n, b, i)$ is an implementation of $S(n)$. Now, as has been said, every implementation is identical to a pair of program and proof. Then, if we let s be an implementation of $S(x)$ then $i(s)$ is a such a pair. But the program component of $i(s)$ does not depend on the proof component of s, simply because programs do not contain proof information. Actually, the function i is itself a pair of functions that work, roughly speaking, component-wise between $S(x)$ and $S(s(x))$. This observation is essential for giving the definition of the construct that we are introducing. The implementation b corresponding to the base case is split into a program b_{p} and corresponding correctness proof b_{c}, just by taking projections. The function i corresponding to the step in the recursion can also be split into a function i_{p} mapping programs to programs and function i_c correspondingly mapping correctness proofs to correctness proofs. Then the program b_p and the function i_p can be put together using *natrec*, so as to get a program working by primitive recursion on natural numbers. Similarly, b_c and i_c can be combined using *natind* to build a proof that the latter program satisfies the predicate given in the specification $S(n)$. In other words, $\mathsf{nattac}(n, b, i)$ can be defined as a pair formed by the program and the proof formed as we have just explained. Actually, the precise formulation of this definition is a delicate point. It relies on a precise stipulation of the rules governing the functions between specifications, which is in fact the central question in the formulation of the theory.

Now, going back to our example, the specification corresponding to the division problem will be written:

$$\Pi n{:}N.\,(\Phi p \epsilon N \times N.\ (n{=}p_1{+}p_2)\ \&\ (p_1{\leq}p_2)\ \&\ (p_2{\leq}s(p_1)))$$

where $\Pi n{:}N.\,S(n)$ is the specification of functions mapping $n : N$ to implementations of the specification $S(n)$.

An implementation of the specification above can be constructed in the same way as it is done when working in set theory. The term we obtain is of the form:

$$\lambda n.\mathsf{nattac}(n, \langle <0, 0>, bv\rangle, u.w.\langle <(w_p)_2, s((w_p)_1)>, iv\rangle)$$

But we have that, because of the definition of λ and nattac, this is in fact a pair, whose program part is exactly the function we wrote for the second solution:

$$\lambda n.natrec(n, <0, 0>, x.w'.<w'_2, s(w'_1)>)\ \epsilon\ N{\rightarrow}N{\times}N$$

So, in this way we achieve our goal. We have a theory where we are building all the time pairs, corresponding to an algorithm and its correctness proof. Going back to the tree-picture of terms we discussed before, what we are doing now is building only one tree, whose form reflects the way we reasoned to solve our problem. The tree has pairs as leaves, which are the implementations with which we finally fill in the holes of the template. But in fact, this tree is just an abbreviation which, once unfolded, yields a pair of trees, corresponding to a program and a verification. These two trees have the same form as the original tree, and their leaves are the result of splitting the leaves of the original tree into program part and verification part.

At present, we have formulated a version of the theory [ST96]. This formulation uses a more basic framework, in the same way as Martin-Löf's monomorphic set theory is expressed in the theory of logical types [NPS90]. This makes it possible to add new datatypes and propositions by just introducing appropriate constants. That is, the theory is easy to extend.

We would like now to implement a programming system for this theory. This is a project similar in nature to that described in the first section of this work. Nevertheless, it cannot be reduced to the latter, due to the distinctive concepts present in this theory.

In the first place, we have pair types, which are the types of specifications and implementations. Secondly, functions between these pair types are themselves pairs of functions, as already explained.

References

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A User's Guide to ALF.* Chalmers University of Technology, Sweden, May 1994. Available on the WWW `file://ftp.cs.chalmers.se/pub/users/alti/alf.ps.Z`.

[BCMS89] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[Bje85] Bror Bjerner. Verifying some Efficient Sorting Strategies in Type Theory. PMG Memo 26, Chalmers University of Technology, S–412 96 Göteborg, January 1985.

[Bov95] Ana Bove. A machine assisted proof of the subject reduction property for a small typed functional language. Master's thesis, INCO, Universidad de la Republica, Montevideo, Uruguay, 1995.

[BT96] Gustavo Betarte and Alvaro Tasistro. *Extension of Martin-Löf's theory of types with record types and subtyping: motivation, rules and type checking.* Extended abstract submitted for publication., 1996.

[CH86] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

[Chi87] P. Chisholm. Derivation of a Parsing Algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, 8:1–42, 1987.

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *EATCS*, 52, February 1994.

[Coq96] Catarina Coquand. *Computation in Type Theory.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden, May 1996.

[DFH+91] G. Dowek, A. Felty, H. Herbelin, H. Huet, G. P. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide version 5.6. Technical report, Rapport Technique 134, INRIA, December 1991.

[Gim95] Eduardo Giménez. An application of co-inductive types in coq: Verification of the alternating bit protocol. In *BRA Workshop on Types for Proofs and Programs*, 1995. To appear in the LNCS Series.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Hue93] Gerard Huet. Residual theory in lambda calculus, a complete Gallina developement, 1993.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical report, LFCS Technical Report ECS-LFCS-92-211, 1992.

[ML87] Per Martin-Löf. *Philosophical Implications of Type Theory.* Privately circulated notes., 1987. Lectures given at the Facoltá de Lettere e Filosofia, Universitá degli Studi di Firenze, Florence, March 15th. - May 15th., 1987.

[MS94] P. Manoury and M. Simonot. Automatizing termination proofs of recursively defined functions. In *Theoretical Computer Science.*, volume 135, pages 319–343. 1994.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[Pol94] Robert Pollack. *The Theory of Lego A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[ST96] Nora Szasz and Alvaro Tasistro. A Theory of Specifications. Draft Paper, 1996.

[Sza95] Nora Szasz. Towards a Theory of Specifications, Programs and Proofs. In *Proceedings of the 7th. Nordic Workshop on Programming Theory, PMG report 86*, Göteborg, November 1995.

[Tas95] Alvaro Tasistro. Abstract Insertion Sort in Type Theory. In *Proceedings of the 7th. Nordic Workshop on Programming Theory, PMG report 86*, Göteborg, November 1995.

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[vS96] Björn von Sydow. Dependently typed binomial queues. In *Proceedings of Das Wintermöte '96*, pages 141–146, 1996.