Type Theory and Programming *

Thierry Coquand Bengt Nordström Jan M. Smith
Björn von Sydow

June 3, 1994

This paper gives an introduction to type theory, focusing on its recent use as a logical framework for proofs and programs. The first two sections give a background to type theory intended for the reader who is new to the subject. The following presents Martin-Löf's monomorphic type theory and an implementation, ALF, of this theory. Finally, a few small tutorial examples in ALF are given.

Introduction

Programming is the activity of constructing a program that meets some, often vaguely expressed, user needs. The solution process involves understanding the problem, finding the relevant properties that the program must satisfy and deriving the program itself. The process is typically iterative and design decisions have to be changed as the problem is better understood. However, in successful cases, finally a useful program results. As in any problem-solving situation, the programmer's task is not completed only by exhibiting the solution, i.e., the program. In addition, the final understanding of the problem must be properly expressed and some form of argument must be supplied to make it evident that the problem has actually been solved. In trivial cases, the correctness of the program may be self-evident but, as all programmers know by experience, it is all too easy to make "simple" mistakes with serious consequences. Therefore, current practice is to

*This work has been done within the ESPRIT Basic Research Action "Types for proofs and programs." It has been funded by NUTEK and Chalmers. An earlier version was published in the EATCS bulletin no 52, February 1994.
Authors' address: Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden.
Email: coquand,bengt,smith,sydow@cs.chalmers.se.

increase confidence in programs by testing them on sample input data, with obvious shortcomings. In search of more convincing methodology, the natural alternative is to look to mathematical practice, where proposed solutions must be justified through proofs. Several challenges must be addressed in transferring this practice to the discipline of programming:

- Programs must not only be executable on machines but also be mathematically well-behaved to make reasoning about them tractable.

- Problems (or, to use another word, specifications) must be expressed in a precise language and the meaning of the assertion that program p solves problem P must be made precise.

- A proper mathematical framework (a programming logic) must be developed for reasoning formally with machine assistance about programs.

In addition to these basic requirements, many pragmatic considerations must be taken into account. For example, programs must execute with reasonable efficiency, specifications must be easy to comprehend, and the programming logic must be natural to use for practicing programmers.

It is clear that to achieve these goals the programming language, the specification language and the programming logic should be designed together. To choose a suitable mathematical foundation we may note the specific character of the programming task, that of constructing an object with certain properties. This indicates that it might be useful to consider languages developed to express constructive mathematics. But we should also build on insights gained during forty years of programming language design. Among such insights, we emphasize

- the concept of typing, which has proved its usefulness in practical programming,

- the conceptual convenience of the ability for the user to define new data types adapted to the problem at hand,

- the semantical simplicity and expressive power of programming languages which avoid the notion of state, such as functional languages.

In this paper we give an introduction to the intuitionistic type theory developed by Per Martin-Löf [35, 38]. This theory provides a typed functional programming language with inductive types and a specification language encompassing first order predicate logic together with a simple but powerful logical framework for reasoning about programs. The theory was developed to serve as a foundation for constructive mathematics but, as we hope to show here, this makes it equally suitable as a foundation for programming. We should mention here that other closely related theories with similar aims exist, notably Coquand's and Huet's Calculus of Constructions [11].

The notion of types in programming languages goes back to FORTRAN in the fifties where a rudimentary notion of type was introduced to enable the compiler to distinguish between variables ranging over integers and those ranging over floating point numbers. Subsequent languages, in particular in the Algol/Pascal tradition, introduced richer type systems and put more emphasis on typing. Strong typing turns out in practice to be very helpful in finding program errors at an early stage and user-defined types assist in structuring programs. The seventies saw the development of modular languages where module interfaces provide typings of exported entities of a module. These interfaces can be thought of as a weak form of specification for the module. However, the type systems of these languages are too weak to serve as real specifications. The type of a program expresses only some simple properties, which must be supplemented by comments explaining in natural language the full semantics of the program. The type theory to be explained here provides a much richer type system, where any property in predicate logic can be interpreted as a type. Thus the full specification of a program can be expressed by its type and a type-correct program is totally correct.

To see how a type can be understood as a proposition we consider the type system of the polymorphic functional language Standard ML. More precisely, we consider the fragment of ML's type system with only type variables and the binary type formers `->` and `*`, for function types and cartesian products, respectively. Examples of type expressions in this language are `'a -> ('b -> 'a)`, `'a * 'b -> 'a` and `'a -> 'b`. Now it is instructive to read these expressions as propositions by interpreting `->` as implication, `*` as conjunction and type variables as propositional variables. Then the first two examples are tautologies, but not the third. Interestingly, it is also easy to see that only two first types are inhabited by total functions in ML. A function of the first type is the `K` combinator, or in ML notation `fn a => fn b => a`, and one of the second is the function `fst`, defined as `fn (a,b) => a`. Obviously, no total function can be assigned the third type, since there is no way to use an argument of an arbitrary type `'a` to produce an output of another arbitrary type `'b`. This is not just a formal coincidence; it turns out that one can explain the constructive meaning of the logical operations in propositional logic so that provability of an expression read as a proposition is the same as inhabitance of the expression read as a type. This is Heyting's explanation, which we will give in detail in the next section. As we will see, this explanation can be extended to full predicate logic, thus providing types that can be read as propositions involving quantifiers. The crucial extension of ML's type system that will make this possible is the provision of *dependent types*.

A pioneering work on proof checking is de Bruijn's AUTOMATH [13, 14]. This is a computer system for checking ordinary mathematical proofs and, for this purpose, de Bruijn was not satisfied with a traditional formalization of mathematics in set theory expressed in predicate logic; instead he was led to a type theoretic way of expressing proofs, that is, as objects of types. So AUTOMATH can be seen as an early formulation of the kind of type theory we will discuss here.

The first implementation of Martin-Löf's type theory was made in Göteborg [41]; this system was based on the Edinburgh LCF proof assistant [21, 40]. Soon after this a more advanced system, NuPRL, was developed at Cornell [7].

Recently several interactive proof systems based on type theory have been implemented which have the important property that they can serve as logical frameworks: rules and axiom of various theories can easily be expressed in them. The systems Coq [16] and LEGO [31] are both based on the Calculus of Constructions; in this paper we will describe ALF [2, 32], which is based on Martin-Löf's type theory. These computer systems have now been in use for about three years and among the examples carried out in them we just mention the following. The correctness of a data link protocol developed and used by Philips, modeled using I/O automata theory, has been checked with the Coq system [22]. A proof of strong normalization for Girard system F has been developed in LEGO [1]. A nontrivial example of extracting an algorithm from a constructive proof, using ALF, is given in [18] where a type theoretic analysis of Ramsey's theorem is presented. Most of the examples done in these systems have been developed in a truly interactive way, that is, the user has not started out with a detailed proof on paper which then has been transferred to the computer; on the contrary the systems are often of real help when making the proofs. In fact, there are even results which probably would be very difficult to obtain without computer assistance: in [8] a complete set of computation rules for simply typed λ-calculus with explicit substitution is derived from the semantics of the language; this proof involves a detailed syntactical analysis for which computer assistance is essential.

There is, of course, no possibility for us to present any bigger examples here, but at the end of this paper we will illustrate ALF by some smaller examples.

Propositions as sets

In Martin-Löf's type theory there are two basic levels: types and sets. We will later explain what we mean by a type and by a set, but we should point out already now that a set is inductively defined and would, in the usual programming terminology, be called a type.

The basic idea behind using type theory for developing proofs and programs is the Curry-Howard isomorphism between propositions and sets; but before explaining this, a few words about constructive mathematics.

Constructive mathematics arose as an independent branch of mathematics out of the foundational crisis in the beginning of this century, mainly developed by Brouwer under the name intuitionism. It did not get much support because of the general belief that important parts of mathematics were impossible to develop constructively. During the last decades, however, this belief has been shown to be wrong, in particular by the work of Bishop. In his book *Foundations of Constructive Analysis* [4], Bishop rebuilds constructively central parts of classical analysis; and he does it in a way that demonstrates that constructive mathematics can be as elegant as classical mathematics. Of more recent work, we could mention the use of point free topology [27, 34, 44], which often makes it possible to replace highly non-constructive reasoning involving the axiom of choice by constructive proofs [37, 9]. For a presentation of the fundamental ideas of constructive mathematics we refer to Bishop's book, in particular the first chapter "A constructivist manifesto," Dummet [17], Heyting [23], and Troelstra and van Dalen [47, 48].

The debate whether mathematics should be built up constructively or not need not concern us here. It is sufficient to notice that constructive mathematics has some fundamental notions in common with computer science, above all the notion of computation. So, constructive mathematics could be an important source of inspiration for computer science; this was realized already by Bishop [5]. In principle, an implementation of type theory like the system ALF can also be used to express proofs by contradiction; in fact AUTOMATH was used to check classical mathematics. For applications in programming, however, we don't know of any example where non-constructive reasoning is essential.

In order to explain how a proposition can be expressed as a set we will explain the intuitionistic meaning of the logical constants, specifically in the way of Heyting [23]. In classical mathematics, a proposition is thought of as being true or false independently of whether we can prove or disprove it. On the other hand, a proposition is constructively true only if we have a method of proving it. For example, classically the law of excluded middle, $A \vee \neg A$, is true since a proposition is either true or false. Constructively, however, a disjunction is true only if we can prove one of the disjuncts. Since we have no method of proving or disproving an arbitrary proposition A, we have no proof of $A \vee \neg A$ and therefore the law of excluded middle is not intuitionistically valid.

So, the constructive explanations of propositions are spelled out in terms of proofs and not in terms of a world of mathematical objects existing independently of us. Let us first consider implication and conjunction.

> A proof of $A \supset B$ is a function (method, program) which to each proof of A gives a proof of B.

The notion of function or method is primitive in constructive mathematics and a function from a set A to a set B can be viewed as a program which when applied to an element in A gives an element in B as output. For example, in order to prove $A \supset A$ we have to give a method which to each proof of A gives a proof of A; the obvious choice is the method which returns its input as result. This is the identity function $\lambda x.x$, using the λ-notation.

> A proof of $A \,\&\, B$ is a pair whose first component is a proof of A and whose second component is a proof of B.

If we denote the left projection by *fst*, that is $\mathit{fst}(\langle a, b\rangle) = a$ where $\langle a, b\rangle$ is the pair of a and b, then $\lambda x.\mathit{fst}(x)$ is a proof of $(A \,\&\, B) \supset A$, since $\lambda x.\mathit{fst}(x)$ is a function which to each proof of $A \,\&\, B$ gives a proof of A.

The idea behind propositions as sets is to identify a proposition with the set of its proofs. That a proposition is true then means that the corresponding set is nonempty. For implication and conjunction we get, in view of the explanations above,

> $A \supset B$ is identified with $A \rightarrow B$, the set of functions from A to B

and

> $A \,\&\, B$ is identified with $A \times B$, the cartesian product of A and B.

The elements in $A \to B$ are of the form $\lambda x.b$, where $b \in B$ and b may depend on $x \in A$, and the elements in set $A \times B$ are of the form $\langle a, b \rangle$ where $a \in A$ and $b \in B$.

Let us now see what set forming operations are needed for the remaining logical constants.

A disjunction is constructively true if and only if we can prove one of the disjuncts. So a proof of $A \vee B$ is either a proof of A or a proof of B together with the information of which of A or B we have a proof. Hence,

> $A \vee B$ is identified with $A + B$, the disjoint union of A and B.

The elements in $A + B$ are of the form $\mathsf{inl}(a)$ and $\mathsf{inr}(b)$, where $a \in A$ and $b \in B$.

The negation of a proposition A can be defined by:

$$\neg A \quad \equiv \quad A \supset \bot$$

where $\bot$ stands for absurdity, that is a proposition which has no proof. If we let $\emptyset$ denote the empty set, we have

> $\neg A$ is identified with the set $A \to \emptyset$

using the interpretation of implication.

For expressing propositional logic, we have only used sets (types) that are available in many programming languages. In order to deal with the quantifiers, we also need operations defined on families of sets, i.e. sets B depending on elements x in some set A. Heyting's explanation of the existential quantifier is the following.

> A proof of $(\exists x \in A)B$ consists of a construction of an element a in the set A together with a proof of $B[x := a]$.

$B[x := a]$ denotes the result of substituting a for all free occurrences of the variable x. So, a proof of $(\exists x \in A)B$ is a pair whose first component a is an element in the set A and whose second component is a proof of $B[x := a]$. The set corresponding to this is the disjoint union of a family of sets, denoted by $(\Sigma x \in A)B$. The elements in this set are pairs $\langle a, b \rangle$ where $a \in A$ and $b \in B[x := a]$. We get the following interpretation of the existential quantifier.

$(\exists x \in A)B$ is identified with the set $(\Sigma x \in A)B$.

Finally, we have the universal quantifier.

> A proof of $(\forall x \in A)B$ is a function (method, program) which to each element a in the set A gives a proof of $B[x := a]$.

The set corresponding to the universal quantifier is the cartesian product of a family of sets, denoted by $(\Pi x \in A)B$. The elements in this set are functions which, when applied to an element a in the set A gives an element in the set $B[x := a]$. Hence,

> $(\forall x \in A)B$ is identified with the set $(\Pi x \in A)B$.

The elements in $(\Pi x \in A)B$ are of the form $\lambda x.b$ where $b \in B$ and both b and B may depend on $x \in A$. Note that if B does not depend on x then $(\Pi x \in A)B$ is the same as $A \to B$, so $\to$ is not needed as a primitive when we have cartesian products over families of sets. In the same way, $(\Sigma x \in A)B$ is nothing but $A \times B$ when B does not depend on x.

Except the empty set, we have not yet introduced any sets that correspond to atomic propositions. One such set is the equality set $a =_A b$, which expresses that a and b are equal elements in the set A. Recalling that a proposition is identified with the set of its proofs, we see that this set is nonempty if and only if a and b are equal. If a and b are equal elements in the set A, we postulate that the constant $\mathsf{id}(a)$ is an element in the set $a =_A b$. This is similar to recursive realizability interpretations of arithmetic where one usually lets the natural number 0 realize a true atomic formula.

If we express the rules for the logical connectives in Gentzen's natural deduction [19] then the introduction rules correspond to the introduction of the constructors of the set-former interpreting the connective. For instance, the introduction rule for conjunction

$$\frac{A \quad B}{A \& B}$$

corresponds to the introduction of a pair in the cartesian product

$$\frac{a \in A \quad b \in B}{\langle a, b \rangle \in A \times B}$$

The elimination rules correspond to the rules for the selectors. So, in case of conjunction, the elimination rules

$$\frac{A\&B}{A} \qquad \frac{A\&B}{B}$$

correspond to the rules for the projections

$$\frac{c \in A \times B}{\mathit{fst}(c) \in A} \qquad \frac{c \in A \times B}{\mathit{snd}(c) \in B}$$

Also the computation rules for the selectors, that is in the case of the projections,

$$\begin{array}{rcl} \mathit{fst}(\langle a, b\rangle) & = & a \\ \mathit{snd}(\langle a, b\rangle) & = & b \end{array}$$

where $a \in A$ and $a \in B$, have a correspondence in the proofs, namely Prawitz' contraction rules [43], which for conjunction are

$$\cfrac{\cfrac{\begin{matrix}\vdots & \vdots \\ A & B\end{matrix}}{A\&B}}{A} \qquad = \qquad \begin{matrix}\vdots \\ A\end{matrix}$$

and

$$\cfrac{\cfrac{\begin{matrix}\vdots & \vdots \\ A & B\end{matrix}}{A\&B}}{B} \qquad = \qquad \begin{matrix}\vdots \\ B\end{matrix}$$

Similar rules holds for the other logical constants. From this we can see that normalizing a proof in natural deduction is closely related to computing the corresponding proof object to normal form.

Martin-Löf's type theory has enough sets to express all the logical constants: cartesian products and disjoint unions over family of sets as well as the empty set. These sets are only examples of the general method of introducing sets in type theory, which is that sets are inductively formed by introduction rules. So we also have sets like the natural numbers and lists over a given set.

The notation we have used above for elements of sets is polymorphic in the sense that an expression may be element in several sets, e.g. the identity function $\lambda x.x$ is a member in all sets of the form $A \to A$. The set theory we will describe is monomorphic, so e.g. a λ-term of the set $A \to B$ will be of the form $\lambda(A, B, b)$; this is important if we want type checking to be decidable.

The Calculus of Constructions also uses the interpretation of propositions as sets but not by interpreting predicate logic but rather higher order logic. This means that only implication and the universal quantifier need to be represented since, in higher order logic, the other logical constants can be interpreted by these two. However, both Coq and LEGO are based on an extension of the Calculus of Constructions which allows the introduction of inductively defined sets.

Historically, the interpretation of propositions as sets starts with Curry [12] who noticed that the axioms for positive implicational calculus, formulated in the Hilbert style,

$$A \supset B \supset A$$

$$(A \supset B \supset C) \supset (A \supset B) \supset A \supset C$$

correspond to the types of the basic combinators K and S

$$K \in \alpha \to \beta \to \alpha$$

$$S \in (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

Modus ponens then corresponds to application of a function. Curry did not give any explanation of this similarity. Tait [46] noticed, when analyzing Gödel's [20] theory of computable functionals of finite types, the further analogy that if the combinators are expressed as λ-terms, then normalizing a proof of a proposition by removing cuts corresponds to computing the associated λ-term to normal form. Curry and Tait's observations were extended to first order intuitionistic arithmetic by Howard [25]. Independently, similar ideas also occur in de Bruijn's AUTOMATH. Scott [45], inspired by AUTOMATH, was the first one to suggest a theory of constructions in which propositions are introduced by types. The idea of using constructions to represent proofs is also related to recursive realizability interpretations, first developed by Kleene [29] for intuitionistic arithmetic and extensively used in metamathematical investigations of constructive mathematics.

Overview of Martin-Löf's Type Theory

There are basically two ways of introducing types in Martin-Löf's type theory: function types and inductively defined sets. The function types make it possible to express rules in a natural deduction style and logic can then be introduced by the idea of propositions as sets. Because of the possibility of introducing sets by induction, type theory is an open theory; it is in this sense that the theory may serve as a logical framework.

There are four judgement forms in type theory:

- A *type*, A is a type,
- $A = B$, A and B are equal types,
- $a \in A$, a is an object in the type A,
- $a = b \in A$, a and b are equal objects in the type A.

In general, a judgement may depend on a context, i.e. a list of assumptions. For instance, the general form of the third judgement is therefore:

$$a \in A \quad [x_1 \in A_1, \ldots, x_n \in A_n]$$

where a and A may depend on $x_1, \ldots, x_n$ and for $j \leq n$, A_j may depend on $x_1, \ldots, x_{j-1}$. Notice that the order of the assumptions is in general important, since the type of one assumption may depend on earlier assumptions.

We will not go into the details of the meaning of these judgements, but the general character of the semantics can be seen from the explanation of what it means to be a type: to know that A is a type is to know what it means to be an object in A and to know when two objects in A are identical. An important requirement is that the judgements must be decidable; in particular, the equality is understood as definitional equality.

How to form types

The type structure is very simple: there are two ways of forming ground types, the type Set and the type of elements of A if $A \in \mathsf{Set}$, and one way of forming function types. Formally, we declare that Set is a type:

$$\mathsf{Set}\ \mathit{type}$$

If $A \in \mathsf{Set}$, i.e. if A is a set, then $\mathit{El}(A)$ is a type; the objects of this type are the elements of the set A.

$$\frac{A \in \mathsf{Set}}{\mathit{El}(A)\ \mathit{type}}$$

We will often write A instead of $El(A)$, since it will always be clear from the context whether we mean A as a set (i.e. as an object in Set) or as a type.

If A is a type and B is a family of types for $x \in A$, then $(x \in A)B$ is the type which contains functions from A to B as objects. All free occurrences of x in B become bound in $(x \in A)B$.

$$\frac{A\ type \qquad B\ type\ \ [x \in A]}{(x \in A)B\ \ type}$$

To know that an object c is in the type $(x \in A)B$ means that we know that when we apply it to an object a in A we get an object $c(a)$ in $B[x := a]$ and that we get identical objects in $B[x := a_1]$ when we apply it to identical objects a_1 and a_2 in A.

How to form objects in a type

Objects in a type are formed from constants and variables using application and abstraction. We already mentioned how to apply a function to an object:

$$\frac{c \in (x \in A)B \qquad a \in A}{c(a) \in B[x := a]}$$

Functions can be formed by abstraction, if $b \in B$ under the assumption that $x \in A$ then $[x]b$ is an object in $(x \in A)B$. All free occurrences of x in b become bound in $[x]b$.

$$\frac{b \in B\ \ [x \in A]}{[x]b \in (x \in A)B}$$

The abstraction is explained by the ordinary β-rule which defines what it means to apply an abstraction to an object in A.

$$\frac{a \in A \qquad b \in B\ \ [x \in A]}{([x]b)(a) = b[x := a] \in B[x := a]}$$

The η-, α- and ξ-rules from the λ-calculus can be justified. We will sometimes use the notation $(A)B$ when B does not contain any free occurrences of x. We will write $(x_1 \in A_1; \ldots; x_n \in A_n)B$ instead of $(x_1 \in A_1) \ldots (x_n \in A_n)B$ and $b(a_1, \ldots, a_n)$ instead of $b(a_1) \ldots (a_n)$ in order to increase the readability. Similarly, we will write $[x_1] \ldots [x_n]e$ as $[x_1, \ldots, x_n]e$.

An object is *saturated* if it is not a function, i.e. if its type is Set or $El(A)$, for $A \in \mathsf{Set}$. The *arity* of an object is the number of arguments it can be applied to in order for the result to be saturated. It is an important property that a well-typed object has a unique arity.

Most of the generality and usefulness of the language comes from the possibilities of introducing new constants. It is in this way that we can introduce the usual mathematical objects like natural numbers, integers, functions, tuples etc. It is also possible to introduce more complicated inductive sets like sets for proof objects.

An object in the type Set is an inductively defined set. A set is defined by its introduction rules, i.e. by giving a collection of *primitive constants* with appropriate types. For example, the set of natural numbers is defined by declaring the constants

$$\begin{array}{rcl} \mathsf{N} & \in & \mathsf{Set} \\ \mathsf{succ} & \in & (\mathsf{N})\mathsf{N} \\ \mathsf{0} & \in & \mathsf{N} \end{array}$$

Since a set is built up inductively, we know exactly in what ways we can build up the elements in the set. For any expression of type N, we know that it computes either to $\mathsf{0}$ or $\mathsf{succ}(n)$ for a natural number n. It is not permitted to extend N with new constructors.

In the examples below, more sets will be introduced.

Definitions

A distinction is made between primitive and defined constants. The value of a primitive constant is the constant itself. So the constant has only a type, it doesn't have a definition. A *defined constant*, on the other hand, is defined in terms of other objects. When we apply a defined constant to all its arguments in an empty context, e.g. $c(e_1, \ldots, e_n)$, then we get an expression which is a definiendum, i.e. an expression which computes in one step to its definiens (which is a well-typed object).

A defined constant can either be explicitly or implicitly defined. We declare an *explicitly defined constant* c by giving a definition of it:

$$c = a \in A$$

For instance we can make the following explicit definitions:

$$\begin{array}{rcl} 1 & = & \mathsf{succ}(\mathsf{0}) \in \mathsf{N} \\ I_{\mathsf{N}} & = & [x]x \in (\mathsf{N})\mathsf{N} \\ I & = & [A, x]x \in (A \in \mathsf{Set}; A)A \end{array}$$

The last example is the monomorphic identity function which when applied to an arbitrary set A yields the identity function on A. It is easy to check if an explicit definition is correct: you just check that the definiens is an object in the correct type.

We declare an *implicitly defined constant* by showing what definiens it has when we apply it to its arguments. An implicit definition may be recursive and there is no syntactical restriction that guarantees that it is correct. Whether this kind of definition is meaningful can in general only be checked outside the theory. We must be sure that every well-typed expression of the form $c(e_1, \ldots, e_n)$ is a definiendum with a unique well-typed definiens. Here are some examples:

$$\begin{array}{rcl} + & \in & (\mathsf{N}; \mathsf{N})\mathsf{N} \\ +(\mathsf{0}, y) & = & y \\ +(\mathsf{succ}(x), y) & = & \mathsf{succ}(+(x, y)) \\ \mathsf{natrec} & \in & (\mathsf{N}; (\mathsf{N}; \mathsf{N})\mathsf{N}; \mathsf{N})\mathsf{N} \\ \mathsf{natrec}(d, e, \mathsf{0}) & = & d \\ \mathsf{natrec}(d, e, \mathsf{succ}(a)) & = & e(a, \mathsf{natrec}(d, e, a)) \end{array}$$

The last example is a specialized version of the operator for primitive recursion.

An example: definition of conjunction

We are representing proofs as mathematical objects. The type of a proof object represents the proposition which is the conclusion of the proof. Variables are used as names of assumptions and constants are used as rules. To apply a rule to a number of subproofs is done by applying a constant to the corresponding subproof objects.

A theory is presented by a list of typings and definitions of constants. When we read the constant as a name of a rule, then a primitive constant is usually a formation or introduction rule, an implicitly defined constant is an elimination rule (with the contraction rule expressed as the step from the definiendum to the definiens) and finally, an explicitly defined constant is a lemma or derived rule. As an example of this, consider the definition of conjunction.

The formation rule for conjunction expresses that $A \,\&\, B$ is a proposition if A and B are propositions:

$$\frac{A\ prop \qquad B\ prop}{A \,\&\, B\ prop}$$

We express this by introducing the primitive constant $\&$ by the following typing:

$$\& \in (\mathsf{Set};\ \mathsf{Set})\mathsf{Set}$$

We use the type of sets to represent the type of propositions. A canonical proof of the proposition $A \,\&\, B$ consists of a proof of A and a proof of B. This is expressed in natural deduction as

$$\frac{A \qquad B}{A \,\&\, B}$$

This is the introduction rule for conjunction. In type theory we express it by introducing a primitive constant $\&\mathbf{I}$:

$$\&\mathbf{I} \in (A \in \mathsf{Set};\ B \in \mathsf{Set};\ A;\ B)A \,\&\, B$$

Notice that $\&\mathbf{I}$ has an arity which is greater than the number of premises of the corresponding rule. This is because we have to declare explicitly that A and B are ranging over sets, a fact which is not made explicit in the natural deduction rule.

The elimination rules for conjunction

$$\frac{A \,\&\, B}{A} \qquad \frac{A \,\&\, B}{B}$$

are expressed by introducing the implicitly defined constants $\&\mathbf{E}_l$ and $\&\mathbf{E}_r$ by the following declarations and definitions:

$$\begin{array}{rcl} \&\mathbf{E}_l & \in & (A \in \mathsf{Set}; B \in \mathsf{Set}; A \,\&\, B)A \\ \&\mathbf{E}_r & \in & (A \in \mathsf{Set}; B \in \mathsf{Set}; A \,\&\, B)B \\ \&\mathbf{E}_l(A, B, \&\mathbf{I}(A, B, a, b)) & = & a \\ \&\mathbf{E}_r(A, B, \&\mathbf{I}(A, B, a, b)) & = & b \end{array}$$

The two last rules are the contraction rules for $\&$ and these are essential for the correctness of the elimination rule. Since all proofs of $A \,\&\, B$ are equal to a proof of the form $\&\mathbf{I}(A, B, a, b)$, where a is a proof of A and b a proof of B, we know from the contraction rule that we get a proof of B if we apply $\&\mathbf{E}_r$ to an arbitrary proof of $A \,\&\, B$, and similarly for $\&\mathbf{E}_l$.

ALF, an interactive editor for type theory

A basic idea in ALF is to use a proof object as a true representative of a proof. The process of proving the proposition A is represented by the process of building a proof object of A. There is a close connection between the individual steps in proving A and the steps to build a proof object of A. For instance the act of applying a rule is done by building an application of a constant, to assume that a proposition A holds is to make an abstraction of a variable of the type A and to refer to an assumption is to use the corresponding variable.

We want to have an impression that the objects which are built (and changed) are directly manipulated on the screen using the keyboard and the mouse. It is as if we have a hand (represented by the cursor) on the screen to select parts and to grasp for different tools which can manipulate the object. A change of the object is immediately shown on the screen.

When we are proving a proposition A in a theory then we are building a proof object of type A in an environment consisting of a list of declaration of constants. This is presented on the screen by having two windows: a theory window containing declarations of constants and a scratch area containing objects being edited. The scratch area contains different kinds of buffers to build types and objects. A type buffer is used to build a type. The objects which are being built in the scratch area are always correct relative to the current theory.

When we build a top-down proof of a proposition A, then we try to reduce the problem A to some subproblems $B_1, \ldots, B_n$ by using a rule c which takes proofs of $B_1, \ldots, B_n$ to a proof of A. Then we continue by proving $B_1, \ldots, B_n$. For instance, we can reduce the problem A to the two problems $C \supset A$ and C by using modus ponens. In this way we can continue until we have only axioms and assumptions left. This process corresponds exactly to how we can build a mathematical object from the outside and in. Then we are constructing an object of type A by using a function c which takes objects in $B_1, \ldots, B_n$ to an object in A. Then we continue by finding objects in $B_1, \ldots, B_n$.

If we represent the proof process by the process of building a proof object it must be possible to deal with *incomplete* proof objects, i.e. proof objects which represents incomplete proofs. We are using placeholders $?_1, \ldots, ?_n$ for parts of the objects which are to be filled in. The expression

$$? \in A$$

expresses a state of an ongoing process of finding an object in the type A. We say that the expected type of ? is A. Objects are built up from variables and constants using application and abstraction. Therefore there are four ways of refining a placeholder:

- The placeholder is replaced by a constant c. This is correct if the type of c is equal to A.
- The placeholder is replaced by a variable x. The type of x must be equal to A. But we cannot replace a placeholder with any variable of the correct type, the variable must be in the local scope of the placeholder.
- The placeholder is replaced by an abstraction $[x]?_1$. We must have that
$$[x]?_1 \in A$$
which holds if A is equal to a functional type $(y \in B)C$. The type of the variable x must be B and we must remember that $?_1$ may be substituted by an expression which may depend on the variable x. This corresponds to making a new assumption, when we are constructing a proof. We reduce the general problem $(y \in B)C$ to the problem $C[y := x]$ under the assumption that $x \in B$. The assumed object x can be used to construct a solution to C, i.e. we may use the knowledge that we have a solution to the problem B when we are constructing a solution to the problem C.
- Finally, the placeholder can be replaced by an application $c(?_1, \ldots ?_n)$ where c is a constant, or $x(?_1, \ldots, ?_n)$, where x is a variable.

In the case of application, we consider a simple case. We start with $? \in A$ where A is not a function type. If we apply the constant c of type $(x \in B)C$, then the new term will be
$$c(?_1) \in A$$
where the new placeholder $?_1$ must have the type B (since all arguments to c must have that type) and furthermore the type of $c(?_1)$ must be equal to A, i.e. the following equality must hold:
$$C[x := ?_1] = A.$$
So, the editing step from $? \in A$ to $c(?_1) \in A$ is correct if $?_1 \in B$ and $C[x := ?_1] = A$. This operation corresponds to applying a rule when we are constructing a proof. The rule c reduces the problem A to the problem B.

An example in ALF

As a first small example of a proof we prove one half of the distributivity of conjunction over disjunction in propositional logic. A variant of this example was used already by Gentzen [19] to motivate natural deduction. We have already defined the constants for conjunction; we repeat them here together with the corresponding definitions for disjunction as given to ALF. At present, ALF does not support infix operators, so we will write propositions with the logical operations in prefix form as in $\mathsf{And}(A, \mathsf{Or}(B, C))$.

$\mathsf{And}..\in..(A,B \in \mathbf{Set})\ \mathbf{Set}$
$\quad \mathsf{AndI}..\in..(\downarrow A, \downarrow B \in \mathbf{Set};\ A;\ B)\ \mathsf{And}(A,B)$
$\mathsf{AndE}_l..\in..(\downarrow A, \downarrow B \in \mathbf{Set};\ \mathsf{And}(A,B))\ A$
$\quad \mathsf{AndE}_l(\mathsf{AndI}(h_1,h_2))..\equiv..h_1$
$\mathsf{AndE}_r..\in..(\downarrow A, \downarrow B \in \mathbf{Set};\ \mathsf{And}(A,B))\ B$
$\quad \mathsf{AndE}_r(\mathsf{AndI}(h_1,h_2))..\equiv..h_2$
$\mathsf{Or}..\in..(A,B \in \mathbf{Set})\ \mathbf{Set}$
$\quad \mathsf{OrI}_l..\in..(\downarrow A, \downarrow B \in \mathbf{Set};\ A)\ \mathsf{Or}(A,B)$
$\quad \mathsf{OrI}_r..\in..(\downarrow A, \downarrow B \in \mathbf{Set};\ B)\ \mathsf{Or}(A,B)$
$\mathsf{OrE}..\in..(\downarrow A, \downarrow B, \downarrow C \in \mathbf{Set};\ (A)\,C;\ (B)\,C;\ \mathsf{Or}(A,B))\ C$
$\quad \mathsf{OrE}(h,h_1,\mathsf{OrI}_l(h_3))..\equiv..h(h_3)$
$\quad \mathsf{OrE}(h,h_1,\mathsf{OrI}_r(h_3))..\equiv..h_1(h_3)$

This picture and the ones to follow in this section are screen dumps from a session with ALF. The vertical arrows in front of some arguments indicate that these arguments should be suppressed, as is done in the contraction rules. These arguments can typically be inferred by the system, so the user may completely ignore them.

The theorem we want to prove is

$$(A, B, C \in \mathsf{Set}; \mathsf{And}(A, \mathsf{Or}(B, C)))\mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$$

This is a function type. An object of this type expects four arguments: three sets A, B and C and a proof of $\mathsf{And}(A, \mathsf{Or}(B, C))$. Given this input, the function must produce an element of (a proof of) the set (the proposition) $\mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$. We shall do this in two different ways:

1. By defining a function *distr1* as an abbreviation, or in ALF terminology, an explicit constant. This will be done by using the elimination

rules to analyze the last argument and then the introduction rules to build up the result.

2. By defining a function *distr2* as an implicit constant, where the argument is analyzed directly by pattern matching and the elimination rules are not needed.

Using the first method, we instruct the system that we intend to define an explicit constant with the name *distr1*. The system responds with a template for the definition:

$$\boxed{\mathrm{distr1} \equiv ?_{\mathrm{distr1}} \in ?_{\mathrm{distr1.T}}}$$

The definition has two placeholders, one for the definiens and one for its type. We proceed by filling in the type:

$$\boxed{\mathrm{distr1} \equiv ?_{\mathrm{distr1}} \in (A, B, C \in \mathbf{Set};\ \mathrm{And}(A, \mathrm{Or}(B, C)))\ \mathrm{Or}(\mathrm{And}(A, B), \mathrm{And}(A, C))}$$

The definiens is built top-down. The first step is to form an abstraction

$$\boxed{\mathrm{distr1} \equiv [A, B, C, h]?_{e} \in (A, B, C \in \mathbf{Set};\ \mathrm{And}(A, \mathrm{Or}(B, C)))\ \mathrm{Or}(\mathrm{And}(A, B), \mathrm{And}(A, C))}$$

where the $?_e$ should have type $\mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$. To construct an element in this type we must analyze the second projection of h using the *OrE* rule:

$$\boxed{\mathrm{distr1} \equiv [A, B, C, h]\mathrm{OrE}(?_{\mathrm{h1}}, ?_{\mathrm{h2}}, \mathrm{AndE_r}(h))}$$

The three first arguments are inferred by the system and not displayed. We have to fill in the remaining two arguments, where the first has type $(B)\mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$. An object in this type is again a function, which gets a proof of B as input. Using this input, it is easy to construct the result using the left introduction rule for Or. Similarly, the second argument can be completed using the right Or-introduction rule, giving the complete proof

$$\begin{aligned}\mathrm{distr1} \equiv [A,B,C,h]\mathrm{OrE}(&[h_1]\mathrm{OrI_l}(\mathrm{AndI}(\mathrm{AndE_l}(h),h_1)),\\ &[h_1]\mathrm{OrI_r}(\mathrm{AndI}(\mathrm{AndE_l}(h),h_1)),\\ &\mathrm{AndE_r}(h))\end{aligned}$$

The omitted steps in this proof all consist of simple direct manipulations with the mouse; nothing is typed on the keyboard (except the name *distr1*).

In the second method, using pattern matching instead of elimination rules, the initial situation in building the proof is the following:

$$\begin{aligned}&\mathrm{distr2} \in (A,B,C \in \mathbf{Set};\ \mathrm{And}(A,\mathrm{Or}(B,C)))\ \mathrm{Or}(\mathrm{And}(A,B),\mathrm{And}(A,C))\\ &\quad \mathrm{distr2}(A,B,C,h) \equiv ?_{\mathrm{distr2.0.E}}\end{aligned}$$

Here we can now ask the system to analyze the possible forms of an argument. We perform this pattern matching on the last argument, giving

$$\begin{aligned}&\mathrm{distr2} \in (A,B,C \in \mathbf{Set};\ \mathrm{And}(A,\mathrm{Or}(B,C)))\ \mathrm{Or}(\mathrm{And}(A,B),\mathrm{And}(A,C))\\ &\quad \mathrm{distr2}(A,B,C,\mathrm{AndI}(h_1,h_2)) \equiv ?_{\mathrm{distr2.0.0.E}}\end{aligned}$$

There is only one constructor for And, so h must evaluate to $\mathsf{AndI}(h_1, h_2)$, where $h_1 \in A$ and $h_2 \in \mathsf{Or}(B, C)$. We proceed by analyzing h_2, which must reduce to one of two possible forms:

$$\begin{aligned}&\mathrm{distr2} \in (A,B,C \in \mathbf{Set};\ \mathrm{And}(A,\mathrm{Or}(B,C)))\ \mathrm{Or}(\mathrm{And}(A,B),\mathrm{And}(A,C))\\ &\quad \mathrm{distr2}(A,B,C,\mathrm{AndI}(h_1,\mathrm{OrI_l}(h))) \equiv ?_{\mathrm{distr2.0.0.0.E}}\\ &\quad \mathrm{distr2}(A,B,C,\mathrm{AndI}(h_1,\mathrm{OrI_r}(h))) \equiv ?_{\mathrm{distr2.0.0.1.E}}\end{aligned}$$

So, the pattern matching separates the task of defining *distr2* into two mutually exhaustive cases. In both cases it is immediate to construct the right hand side using constructors and the variables on the left hand side:

$$\begin{aligned}&\mathrm{distr2} \in (A,B,C \in \mathbf{Set};\ \mathrm{And}(A,\mathrm{Or}(B,C)))\ \mathrm{Or}(\mathrm{And}(A,B),\mathrm{And}(A,C))\\ &\quad \mathrm{distr2}(A,B,C,\mathrm{AndI}(h_1,\mathrm{OrI_l}(h))) \equiv \mathrm{OrI_l}(\mathrm{AndI}(h_1,h))\\ &\quad \mathrm{distr2}(A,B,C,\mathrm{AndI}(h_1,\mathrm{OrI_r}(h))) \equiv \mathrm{OrI_r}(\mathrm{AndI}(h_1,h))\end{aligned}$$

The proof is complete.

The pattern matching approach is the outcome of experiments with the

ALF system [10] and is not present in type theory proper as presented by Martin-Löf. In fact, the two proofs presented here are just instances of two different disciplines. In the first, one defines for each set once and for all an elimination rule, capturing proof by structural induction over elements of the set.[1] These elimination rules are defined as implicit constants and are justified by reflection on the definition of the set. Having done this, all proofs involving elements of this set are defined as explicit constants. In the second discipline, only the set with its introduction rules is defined at the outset and later proofs are done by pattern-matching, involving a reflection on the arguments specially adapted to the particular proposition one wants to prove. Interestingly, the two disciplines are not proof-theoretically equivalent. One can exhibit propositions that can be proved by pattern-matching but are false in certain models of type theory with the standard elimination rules [24]. It is presently a research topic to gain a better understanding of this phenomenon.

A programming example

To indicate how type theory can be used as a programming logic, we give a simple example of the derivation of a program. The problem we consider is that of finding the minimal element in a non-empty sequence of natural numbers. We have already defined the set of natural numbers; the set of sequences is defined as

```
NSeq..∈..Set
   single..∈..(n ∈ N) NSeq
   cons..∈..(n ∈ N; ns ∈ NSeq) NSeq
```

To define the notion of minimality we need an order relation on the natural numbers. This can be inductively defined as

```
Leq..∈..(m,n ∈ N) Set
   leq_0..∈..(n ∈ N) Leq(0,n)
   leq_succ..∈..(m,n ∈ N; Leq(m,n)) Leq(succ(m),succ(n))
```

[1] The elimination rules for And and Or given here can be slightly generalized, using dependent sets. In this generalization, the two And-elimination rules are replaced by just one rule.

Of course, these definitions are standard and can be expected to exist in a library of useful sets in the environment of a system like ALF. The same goes for simple properties of these sets, such as the fact that Leq is total, i.e. $\mathsf{Or}(\mathsf{Leq}(m, n), \mathsf{Leq}(n, m))$ for all m and n, which is easily proved by pattern matching. Below we shall use a proof object $\mathit{leq\_total}(m, n)$ of this without giving the proof here. We note also that this is the only property of Leq that we need.

We can now define the relation $\mathsf{Minimal}$ between a natural number and a sequence. The idea is that $\mathsf{Minimal}(n, ns)$ should express that n is the minimal element in ns. This relation is inductively defined by the following rules:

$$\frac{n \in \mathsf{N}}{\mathsf{Minimal}(n, \mathsf{single}(n))} \qquad \frac{\mathsf{Minimal}(n, ns) \qquad m \leq n}{\mathsf{Minimal}(m, \mathsf{cons}(m, ns))} \qquad \frac{\mathsf{Minimal}(n, ns) \qquad n \leq m}{\mathsf{Minimal}(n, \mathsf{cons}(m, ns))}$$

We express this in type theory as the definitions

```
Minimal ∈ (n ∈ N; ns ∈ NSeq) Set
   m0 ∈ (n ∈ N) Minimal(n, single(n))
   m1 ∈ (m, n ∈ N; ns ∈ NSeq; Minimal(n, ns); Leq(m, n)) Minimal(m, cons(m, ns))
   m2 ∈ (m, n ∈ N; ns ∈ NSeq; Minimal(n, ns); Leq(n, m)) Minimal(n, cons(m, ns))
```

We now have enough notions to specify a program that given a sequence produces its minimal element. Using the existential quantifier the type of this program is $(ns \in \mathsf{NSeq})\exists(\mathsf{N}, [n]\mathsf{Minimal}(n, ns))$. We prove this by pattern matching, defining an implicit constant min:

```
min ∈ (ns ∈ NSeq) ∃(N, [n]Minimal(n, ns))
   min(single(n)) ≡ ∃_intro(n, m0(n))
   min(cons(n, ns1)) ≡
      case min(ns1) ∈ ∃(N, [n']Minimal(n', ns1)) of
         ∃_intro(a, h) ⇒ case leq_total(a, n) ∈ Or(Leq(a, n), Leq(n, a)) of
                             OrI1(h1) ⇒ ∃_intro(a, m2(n, a, ns1, h, h1))
                             OrI2(h1) ⇒ ∃_intro(n, m1(n, a, ns1, h, h1))
                         end
      end
```

The proof uses case expressions, a further extension of the pattern matching discipline, which should be easy to understand. We note the recursive call in the cons equation; it is necessary for termination that the recursive call is, as here, to a *structurally smaller* argument, i.e. to a variable which is a subpattern of the corresponding pattern in the left hand side. With this proviso, the recursive call is guaranteed to reduce to canonical form $\exists\mathsf{\_intro}(a, h)$ and we can use the lemma *leq_total* to decide which of a and n is the minimum.

If we compute $min(ns)$ for a given ns we get a pair, consisting of the minimal element and a proof of the minimality. The latter component is useful for building the proof smoothly, but from a computational point of view it is redundant. Once we have completed the proof we would like to strip it off before we use the program. We can here only indicate the ideas behind such *program extraction* [39, 3]. If we are only interested in the first component of $min(ns)$, we would like to erase the second component everywhere. In this particular case it easy to see that this is possible, since the second component h of the recursive call is only used to build the second component of the result in the cons case. In general, the analysis can be quite involved, since proof components may play also a computational rôle. Similarly, $leq\_total(a, n)$ returns information on which of $\mathsf{Leq}(a, n)$ and $\mathsf{Leq}(n, a)$ that is true, and also a proof of that fact. It is easy to see that the proof components of $leq\_total(a, n)$ are only used in building the verification part of $min(ns)$ and thus the type $\mathsf{Or}(\mathsf{Leq}(a, n), \mathsf{Leq}(n, a))$ can be simplified to Bool, giving exactly the program one would write in a functional language. It is currently an active field of research to find algorithms for the automatic analysis of proofs to remove such computationally redundant parts.

A simple compiler

We give an example of a proof of correctness of a simple compiler for polynomial expressions and its representation in type theory. The proof we present has been mechanically checked in the implementation of type theory ALF presented above. It illustrates that type theory provides a convenient notation for proofs that proceed by induction and case analysis on the form of a possible derivation. This style of proof is common in natural semantics [28, 15] and structural operational semantics [42].

This example illustrates the following points of a type theoretical representation of a programming problem.

- The problem and its solution is completely formalized. It has been represented on a computer and mechanically checked.

- Identification of types and propositions, and of programs and proofs. The proofs are done by case analysis and structural induction on derivations and they are represented by functional expressions (with case expression and recursive definitions).

- Connected to the previous point, we can represent in this formalism features that are usually considered to be purely semantical notion; for instance, in this example we define internally the notion of value of a polynomial expression.

- Type theory can be seen as a functional programming language, whose main feature compared to other functional languages (like ML [36] and Haskell [26]) is the use of dependent types.

- No use of domain theory. This is to be contrasted with the LCF approach [21]. Either we represent directly a program as a term of type theory, or if we want to consider a possibly non-terminating program, we represent it as an inductively defined relation.

This example was suggested by Colin Runciman, who uses it as a LCF exercise. The example gives a good illustration of the differences between the LCF approach [21, 40] and the present "natural semantics" approach [28]. In the LCF approach, each function is considered a priori to be partial, and one has to prove that a given function is total. In the present type theoretic approach, a function represents always a *total* function (for instance, the compiler itself will be represented by such a function). Functions that are only "partially defined" are represented here by inductively defined relations.

One drawback of this approach is in dealing with the fact that such a relation R is "functional": one element is in relation with at most one other element. For instance, it is more elegant to simply write $\phi(f(x))$, where f is the partial function corresponding to the functional relation R, rather than "for all element y, if $R(x,y)$, then $\phi(y)$". This difficulty does not appear in the present example.

Presentation of the example

We want to compute a polynomial expression built out of multiplication, addition, basic integers, and one variable on a simple stack machine. The instructions for this machine are:

- duplication of the top of the stack: from the stack $n :: S$ we go to the stack $n :: n :: S$,

- reverse the two top items of the stack: from the stack $n_1 :: n_2 :: S$ we go to the stack $n_2 :: n_1 :: S$,

- replace the two top items of the stack by an item that is their sum (resp. product): from the stack $n_1 :: n_2 :: S$ we go to the stack $(n_1 * n_2) :: S$ where $*$ is $+$ or $\times$,

- replace the top of the stack by a given item n_0, the stack $n :: S$ becomes $n_0 :: S$.

The problem is to compile a given polynomial expression $e(x)$ to a list l of instructions such that if we execute l on a stack $n :: S$, then the result of this computation is the stack $v :: S$, where v is the value of the expression $e(n)$.

We shall not analyze in detail the solution of this problem in type theory, but only give some remarks that may help in reading this solution. The set of expressions is inductively defined by the constructors

$$\begin{array}{rcl} \mathsf{Sum} & \in & (\mathsf{Expr})(\mathsf{Expr})\mathsf{Expr} \\ \mathsf{Pro} & \in & (\mathsf{Expr})(\mathsf{Expr})\mathsf{Expr} \\ \mathsf{Num} & \in & (\mathsf{N})\mathsf{Expr} \\ \mathsf{Arg} & \in & \mathsf{Expr} \end{array}$$

The set of stacks is defined by:

$$\begin{array}{rcl} \mathsf{Null} & \in & \mathsf{Stack} \\ \mathsf{Push} & \in & (\mathsf{N})(\mathsf{Stack})\mathsf{Stack} \end{array}$$

and the set of instructions has the constructors

$$\begin{array}{rcl} \mathsf{Dup} & \in & \mathsf{Instr} \\ \mathsf{Rev} & \in & \mathsf{Instr} \\ \mathsf{Add} & \in & \mathsf{Instr} \\ \mathsf{Mul} & \in & \mathsf{Instr} \\ \mathsf{Lit} & \in & (\mathsf{N})\mathsf{Instr} \end{array}$$

As said before, usual "functions" are represented in two ways. The evaluation function is represented as a type theoretic function $Eval \in (\mathsf{Expr})(\mathsf{N})\mathsf{N}$, such that $Eval(e, n)$ gives the value of the expression e when its unique variable is set equal to the value n. This is a recursive function, which is defined by case on the expression e.

The execution "function" which computes a final stack from an initial stack and a list of instructions, however, is not a function in a type theoretic sense. It is only a "partial" function, and it is not clear yet if it is needed to extend type theory with such functions (see however the proposal [6]). It is here represented as an inductively defined relation EXEC between a list of instructions and two stacks (representing the initial and final stack). For defining this relation, we need first to introduce an inductively defined relation Exec between only one instruction and two stacks (representing the initial and final stack). The relation EXEC will be the "transitive closure" of this relation. The originality of the type theoretic approach is to consider these inductively defined relations as inductively defined (dependent) *sets*. The constructors of these sets represent then the defining introduction rules of the corresponding relations. The relation Exec has the constructors:

$$\begin{array}{rcl}
\mathsf{ExecDup} & \in & (n \in \mathsf{N}; s \in \mathsf{Stack})\mathsf{Exec}(\mathsf{Dup}, \mathsf{Push}(n, s), \mathsf{Push}(n, \mathsf{Push}(n, s))) \\
\mathsf{ExecRev} & \in & (n_1, n_2 \in \mathsf{N}; s \in \mathsf{Stack})\mathsf{Exec}(\mathsf{Rev}, \mathsf{Push}(n_1, \mathsf{Push}(n_2, s)), \mathsf{Push}(n_2, \mathsf{Push}(n_1, s))) \\
\mathsf{ExecLit} & \in & (n, m \in \mathsf{N}; s \in \mathsf{Stack})\mathsf{Exec}(\mathsf{Lit}(n), \mathsf{Push}(m, s), \mathsf{Push}(n, s)) \\
\mathsf{ExecAdd} & \in & (n_1, n_2 \in \mathsf{N}; s \in \mathsf{Stack})\mathsf{Exec}(\mathsf{Add}, \mathsf{Push}(n_1, \mathsf{Push}(n_2, s)), \mathsf{Push}(plus(n_1, n_2), s)) \\
\mathsf{ExecMul} & \in & (n_1, n_2 \in \mathsf{N}; s \in \mathsf{Stack})\mathsf{Exec}(\mathsf{Mul}, \mathsf{Push}(n_1, \mathsf{Push}(n_2, s)), \mathsf{Push}(mult(n_1, n_2), s))
\end{array}$$

Let us write $s_1 \stackrel{i}{\rightarrow} s_2$ for $\mathsf{Exec}(i, s_1, s_2)$, and $n :: s$ for $\mathsf{Push}(n, s)$. The usual presentation of Exec [28] will be in term of the introduction rules:

$$\frac{}{n :: s \stackrel{\mathsf{Dup}}{\rightarrow} n :: n :: s} \qquad \frac{}{n_1 :: n_2 :: s \stackrel{\mathsf{Rev}}{\rightarrow} n_2 :: n_1 :: s}$$

$$\frac{}{m :: s \stackrel{\mathsf{Lit}(n)}{\rightarrow} n :: s} \qquad \frac{}{n_1 :: n_2 :: s \stackrel{\mathsf{Add}}{\rightarrow} plus(n_1, n_2) :: s}$$

$$\frac{}{n_1 :: n_2 :: s \stackrel{\mathsf{Mul}}{\rightarrow} mult(n_1, n_2) :: s}$$

We introduce the defined set

$$Prog \quad = \quad \mathsf{List}(\mathsf{Instr})$$

The relation EXEC can then be defined as a transitive closure of Exec by the constructors:

$$\begin{array}{rcl} \mathsf{EXEC} & \in & (Prog; \mathsf{Stack}; \mathsf{Stack})\mathsf{Set} \\ \mathsf{EXECnil} & \in & (s : \mathsf{Stack})\mathsf{EXEC}(\mathsf{nil}, s, s) \\ \mathsf{EXECcons} & \in & (\mathsf{Exec}(i, s_1, s_2); \mathsf{EXEC}(p, s_2, s_3))\mathsf{EXEC}(\mathsf{cons}(i, p), s_1, s_3) \end{array}$$

Let us write $s_1 \overset{p}{\longmapsto} s_2$ for $\mathsf{EXEC}(p, s_1, s_2)$, and $i.p$ for $\mathsf{cons}(i, p)$. The usual presentation of such a relation is given by two introduction rules

$$\frac{}{s \overset{\mathsf{nil}}{\longmapsto} s} \qquad\qquad \frac{s_1 \overset{i}{\to} s_2 \quad s_2 \overset{p}{\longmapsto} s_3}{s_1 \overset{i.p}{\longmapsto} s_3}$$

The compiler itself is represented by a type theoretic function $Comp \in (\mathsf{Expr})Prog$ because it should represent a deterministic total program. The main theorem to be proved can then be formulated as

$$n :: S \overset{Comp(e)}{\longmapsto} Eval(e, n) :: S$$

for all polynomial expressions e.

The next two figures contain the complete proof as represented in ALF. The first figure describes the data types (stack, expression, instruction) and the evaluation and compilation functions. The second figure contains a proof of correctness of the compilation function.

```
STACK : Set
  null : STACK     push : (n:N;s:STACK)STACK

EXPR : Set
  SUM : (e1:EXPR;e2:EXPR)EXPR
  PRO : (f1:EXPR;f2:EXPR)EXPR
  NUM : (n:N)EXPR
  ARG : EXPR

Eval : (e:EXPR;n:N)N
  Eval(SUM(e1,e2),n) = plus(Eval(e1,n),Eval(e2,n))
  Eval(PRO(f1,f2),n) = mult(Eval(f1,n),Eval(f2,n))
  Eval(NUM(n1),n) = n1
  Eval(ARG,n) = n

INSTR : Set
  DUP : INSTR  REV : INSTR   ADD : INSTR   MUL : INSTR  LIT : (n:N)INSTR

Exec : (i:INSTR;s1:STACK;s2:STACK)Set
  Exec_Dup : (n:N;s:STACK)Exec(DUP,push(n,s),push(n,push(n,s)))
  Exec_Rev : (n1,n2:N;s:STACK)Exec(REV,push(n1,push(n2,s)),push(n2,push(n1,s)))
  Exec_Add : (n1,n2:N;s:STACK)Exec(ADD,push(n1,push(n2,s)),push(plus(n1,n2),s))
  Exec_Mul : (n1,n2:N;s:STACK)Exec(MUL,push(n1,push(n2,s)),push(mult(n1,n2),s))
  Exec_Lit : (n,m:N;s:STACK)Exec(LIT(n),push(m,s),push(n,s))

PROGRAM = List(INSTR)

EXEC : (p:PROGRAM;s:STACK;s':STACK)Set
  exec_nil : (s:STACK)EXEC(nil,s,s)
  exec_seq : (Exec(i,s1,s2);EXEC(p,s2,s3))EXEC(cons(i,p),s1,s3)

Comp : (e:EXPR)PROGRAM
  Comp(SUM(e1,e2)) = cons(DUP,append(Comp(e2),append(cons(REV,nil),
                     append(Comp(e1),cons(ADD,nil)))))
  Comp(PRO(f1,f2)) = cons(DUP,append(Comp(f2),append(cons(REV,nil),
                     append(Comp(f1),cons(MUL,nil)))))
  Comp(NUM(n)) = cons(LIT(n),nil)
  Comp(ARG) = nil
```

Figure 1: Description of the compiler

```
lemma : (EXEC(p1,s1,s2);EXEC(p2,s2,s3))EXEC(append(p1,p2),s1,s3)

  lemma(exec_seq(h2,h3),h1) = exec_seq(h2,lemma(h3,h1))

  lemma(exec_nil(_),h1) = h1


thm : (e:EXPR;s:STACK;n:N)EXEC(Comp(e),push(n,s),push(Eval(e,n),s))

  thm(SUM(e1,e2),s1,n) = lemma(
                             exec_seq(Exec_Dup(n,s1),thm(e2,push(n,s1),n)),
                             exec_seq(Exec_Rev(Eval(e2,n),n,s1),
                               lemma(thm(e1,push(Eval(e2,n),s1),n),
                                 exec_seq(Exec_Add(Eval(e1,n),Eval(e2,n),s1),
                                   exec_nil(push(Eval(SUM(e1,e2),n),s1))))))

  thm(PRO(f1,f2),s1,n) = lemma(
                             exec_seq(Exec_Dup(n,s1),thm(e2,push(n,s1),n)),
                             exec_seq(Exec_Rev(Eval(e2,n),n,s1),
                               lemma(thm(e1,push(Eval(e2,n),s1),n),
                                 exec_seq(Exec_Mul(Eval(e1,n),Eval(e2,n),s1),
                                   exec_nil(push(Eval(PRO(e1,e2),n),s1))))))

  thm(NUM(n1),s1,n) = exec_seq(Exec_Lit(n1,n,s1),exec_nil(push(n1,s1)))

  thm(ARG,s1,n) = exec_nil(push(n,s1))
```

Figure 2: Proof of correctness

Acknowledgements

The major implementation work was done by Lena Magnusson who implemented the proof engine. We want to thank her for many insightful comments. Johan Nordlander implemented the window interface, we also want to thank him for many discussions.

References

[1] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, 1993.

[2] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.

[3] S. Berardi. An optimization algorithm for simply typed lambda-calculus. Technical report, Turin University, 1993.

[4] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.

[5] Errett Bishop. Mathematics as a numerical language. In Myhill, Kino, and Vesley, editors, *Intuitionism and Proof Theory*, pages 53–71, Amsterdam, 1970. North Holland.

[6] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121:89–112, 1993.

[7] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[8] Catarina Coquand. From Semantics to Rules, A Machine Assisted Analysis. Technical report, Dept. of Comp. Science, Chalmers Univ. of Technology, September 1993.

[9] Thierry Coquand. Constructive topology and combinatorics. In *proceeding of the conference Constructivity in Computer Science, San Antonio, LNCS 613*, pages 28–32, 1992.

[10] Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[11] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[12] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[13] N. G. de Bruijn. The Mathematical Language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61, Versailles, France, 1968. IRIA, Springer-Verlag.

[14] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, New York, 1980. Academic Press.

[15] J. Despeyroux. Proof of Translation in Natural Semantics. In *Proceedings of the First ACM Conference on Logic in Computer Science*, pages 193–205, 1986.

[16] 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.

[17] M. Dummett. *Elements of intuitionism.* Clarendon Press, Oxford, 1977.

[18] Daniel Fridlender. Ramsey's theorem in type theory. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, October 1993.

[19] Gerhard Gentzen. Investigations into Logical Deduction. In E. Szabo, editor, *The Collected Papers of Gerhard Gentzen.* North-Holland Publishing Company, 1969.

[20] Kurt Gödel. Über eine bisher noch nicht benutze erweitrung des finiten standpunktes. *Dialectica*, 12, 1958.

[21] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[22] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In Herman Geuvers, editor, *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 173–212. Esprit Basic Research Action 6453, May 1993.

[23] Arend Heyting. *Intuitionism: An Introduction.* North-Holland, Amsterdam, 1956.

[24] Martin Hofmann. A model of intensional martin-löf type theory in which unicity of identity proofs does not hold. Technical report, Dept. of Computer Science, University of Edinburgh, June 1993. Draft.

[25] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[26] 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.

[27] P. T. Johnstone. The point of pointless topology. *Bull. Amer. Math. Soc*, (8):41–53, 1983.

[28] G. Kahn. Natural Semantics. Technical Report 123, INRIA, Salt Lake City, 1987.

[29] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

[30] G. Kahn. L. Thery, Y. Bertot. Real Theorem Provers Deserve Real User-Interfaces. Technical report, INRIA, Rocquencourt, 1992.

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

[32] Lena Magnusson. The new Implementation of ALF. In *The informal proceeding from the logical framework workshop at Båstad, June 1992*, 1992.

[33] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *The Formal Proceeding of the 1993 Workshop on Types for Proofs and Programs*, Nijmegen, 1994.

[34] Per Martin-Löf. *Notes on Constructive Mathematics.* Almqvist & Wiksell, 1968.

[35] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[36] R. Milner. Standard ML Proposal. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), January 1984.

[37] Christopher J. Mulvey and Joan Wick Pelletier. A Globalization of the Hahn-Banach Theorem. *Advances in Mathematics*, 89(1), 1991.

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

[39] Christine Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions.* PhD thesis, L'Universite Paris VII, 1989.

[40] Lawrence C. Paulson. *Logic and Computation.* Cambridge University Press, 1987.

[41] Kent Petersson. A Programming System for Type Theory. PMG report 9, Chalmers University of Technology, S–412 96 Göteborg, 1982, 1984.

[42] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, September 1981.

[43] D. Prawitz. *Natural Deduction.* Almquist & Wiksell, Stockholm, 1965.

[44] Giovanni Sambin. Intuitionistic Formal Spaces - A First Communication. In *The Proceedings of Conference on Logic and its Applications, Bulgaria.* Plenum Press, 1986.

[45] Dana Scott. Constructive validity. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, Berlin, 1970.

[46] W. Tait. Infinitely long terms of transfinite type. In *Formal systems and recursive functions*, pages 176–185, Amsterdam, 1965. North-Holland.

[47] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics. An Introduction*, volume I. North-Holland, 1988.

[48] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics. An Introduction*, volume II. North-Holland, 1988.