Martin-Löf's Type Theory

B. Nordström, K. Petersson and J. M. Smith

Contents

1 Introduction

The type theory described in this chapter has been developed by Martin-Löf with the original aim of being a clarification of constructive mathematics. Unlike most other formalizations of mathematics, type theory is not based on predicate logic. Instead, the logical constants are interpreted within type theory through the Curry-Howard correspondence between propositions and sets [10, 22]: a proposition is interpreted as a set whose elements represent the proofs of the proposition.

It is also possible to view a set as a problem description in a way similar to Kolmogorov's explanation of the intuitionistic propositional calculus [25]. In particular, a set can be seen as a specification of a programming problem; the elements of the set are then the programs that satisfy the specification.

An advantage of using type theory for program construction is that it is possible to express both specifications and programs within the same formalism. Furthermore, the proof rules can be used to derive a correct program from a specification as well as to verify that a given program has a certain property. As a programming language, type theory is similar to typed functional languages such as ML [19, 32] and Haskell [23], but a major difference is that the evaluation of a well-typed program always terminates.

The notion of constructive proof is closely related to the notion of computer program. To prove a proposition $(\forall x \in A)(\exists y \in B)P(x, y)$ constructively means to give a function f which when applied to an element a in A gives an element b in B such that $P(a, b)$ holds. So if the proposition $(\forall x \in A)(\exists y \in B)P(x, y)$ expresses a specification, then the function f obtained from the proof is a program satisfying the specification. A constructive proof could therefore itself be seen as a computer program and the process of computing the value of a program corresponds to the process of normalizing a proof. It is by this computational content of a constructive proof that type theory can be used as a programming language; and since the program is obtained from a proof of its specification, type theory can be used as a programming logic. The relevance of constructive mathematics for computer science was pointed out already by Bishop [4].

Recently, several implementations of type theory have been made which can serve as logical frameworks, that is, different theories can be directly expressed in the implementations. The formulation of type theory we will describe in this chapter form the basis for such a framework, which we will briefly present in the last section.

The chapter is structured as follows. First we will give a short overview of different formulations and implementations of type theory. Section 2 will explain the fundamental idea of propositions as sets by Heyting's explanation of the intuitionistic meaning of the logical constants. The following section will give a rather detailed description of the basic rules and their semantics; on a first reading some of this material may just be glanced at, in particular the subsection on hypothetical judgements. In section 4 we illustrate type theory as a logical framework by expressing propositional logic in it. Section 5 introduces a number of different sets and the final section give a short description of ALF, an implementation of the type theory of this chapter.

Although self-contained, this chapter can be seen as complement to our book, *Programming in Type Theory. An Introduction* [33], in that we here give a presentation of Martin-Löf's monomorphic type theory in which there are two basic levels, that of types and that of sets. The book is mainly concerned with a polymorphic formulation where instead of a level of types there is a theory of expressions. One major difference between these two formulations is that in the monomorphic formulation there is more type information in the terms, which makes it possible to implement a type checker [27]; this is important when type theory is used as a logical framework where type checking is the same as proof checking.

1.1 Different formulations of type theory

One of the basic ideas behind Martin-Löf's type theory is the Curry-Howard interpretation of propositions as types, that is, in our terminology, propositions as sets. This view of propositions is closely related to Heyting's explanation of intuitionistic logic [21] and will be explained in detail below.

Another source for type theory is proof theory. Using the identification of propositions and sets, normalizing a derivation corresponds to computing the value of the proof term expressing the derivation. One of Martin-Löf's original aims with type theory was that it could serve as a framework in which other theories could be interpreted. And a normalization proof for type theory would then immediately give normalization for a theory expressed in type theory.

In Martin-Löf's first formulation of type theory from 1971 [28], theories like first order arithmetic, Gödel's T [18], second order logic and simple type theory [5] could easily be interpreted. However, this formulation contained a reflection principle expressed by a universe V and including the axiom $\mathsf{V} \in \mathsf{V}$, which was shown by Girard to be inconsistent. Coquand and Huet's Calculus of Constructions [8] is closely related to the type theory in [28]: instead of having a universe V, they have the two types Prop and Type and the axiom $\mathsf{Prop} \in \mathsf{Type}$, thereby avoiding Girard's paradox.

Martin-Löf's later formulations of type theory have all been predicative; in particular second order logic and simple type theory cannot be interpreted in them. The strength of the theory considered in this chapter instead comes from the possibility of defining sets by induction.

The formulation of type theory from 1979 in *Constructive Mathematics and Computer Programming* [30] is polymorphic and extensional. One important difference with the earlier treatments of type theory is that normalization is not obtained by metamathematical reasoning; instead, a direct semantics is given, based on Tait's computability method. A consequence of the semantics is that a term, which is an element in a set, can be computed to normal form. For the semantics of this theory, lazy evaluation is essential. Because of a strong elimination rule for the set expressing the propositional equality, judgemental equality is not decidable. This theory is also the one in *Intuitionistic Type Theory* [31]. It is also the theory used in the Nuprl system [6] and by the group in Groningen [3].

The type theory presented in this chapter was put forward by Martin-Löf in 1986 with the specific intention that it should serve as a logical framework.

1.2 Implementations

One major application of type theory is to use it as a programming logic in which you derive programs from specifications. Such derivations easily become long and tedious and, hence, error prone; so, it is essential to formalize the proofs and to have computerized tools to check them.

There are several examples of computer implementations of proof checkers for formal logics. An early example is the AUTOMATH system [11, 12] which was designed by de Bruijn to check proofs of mathematical theorems. Quite large proofs were checked by the system, for example the proofs in Landau's book *Grundlagen der Analysis* [24]. Another system, which is more intended as a proof assistant, is the Edinburgh (Cambridge) LCF system [19, 34]. The proofs are constructed in a goal directed fashion, starting from the proposition the user wants to prove and then using tactics to divide it into simpler propositions. The LCF system also introduced the notion of metalanguage (ML) in which the user could implement her own proof strategies. Based on the LCF system, a system for Martin-Löf's type theory was implemented in Göteborg 1982 [35]. Another, more advanced, system for type theory was developed by Constable et al at Cornell University [6].

During the last years, several logical frameworks based on type theory have been implemented: the Edinburgh LF [20], Coq from INRIA [13], LEGO from Edinburgh [26], and ALF from Göteborg [1, 27]. Coq and LEGO are both based on Coquand and Huet's calculus of constructions, while ALF is an implementation of the theory we describe in this chapter. A brief overview of the ALF system is given in section 6.

2 Propositions as sets

The basic idea of type theory to identify propositions with sets goes back to Curry [10], 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

$$K \in A \to B \to A$$

$$S \in (A \to B \to C) \to (A \to B) \to A \to C$$

Modus ponens then corresponds to functional application. Tait [39] noticed the further analogy that removing a cut in a derivation corresponds to a reduction step of the combinator representing the proof. Howard [22] extended these ideas to first-order intuitionistic arithmetic. Another way to see that propositions can be seen as sets is through Heyting's [21] explanations of the logical constants. The constructive explanation of logic is in terms of proofs: a proposition is true if we know how to prove it. For implication we have

> 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. The idea of propositions as sets is now to identify a proposition with the set of its proofs. In case of implication we get

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

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

Heyting's explanation of conjunction is that a proof of $A \wedge B$ is a pair whose first component is a proof of A and whose second component is a proof of B. Hence, we get the following interpretation of a conjunction as a set.

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

The elements in the set $A \times B$ are of the form $\langle a, b \rangle$ where $a \in A$ and $b \in B$.

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 the set $A+B$ 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.

In order to interpret propositions defined using quantifiers, we need operations defined on families of sets, i.e. sets B depending on elements x in some set A. We let $B\,[x \leftarrow a]$ denote the expression obtained by substituting a for all free occurrences of x in B. 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 $B\,[x \leftarrow a]$.

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 $B\,[x \leftarrow a]$. 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 $b \in B\,[x \leftarrow a]$. 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 $B\,[x \leftarrow a]$.

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 $B\,[x \leftarrow a]$. Hence,

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

The elements in the set $(\Pi x \in A)B$ 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 $a =_A b$.

When explaining the sets interpreting propositions we have used an informal notation to express elements of the sets. This notation differs from the one we will use in type theory in that that notation will be monomorphic in the sense that the constructors of a set will depend on the set. For instance, an element of $A \to B$ will be of the form $\lambda(A, B, b)$ and an element of $A \times B$ will be of the form $\langle A, B, a, b\rangle$.

3 Semantics and formal rules

We will in this section first introduce the notion of type and the judgement forms this explanation give rise to. We then explain what a family of types is and introduce the notions of variable, assumption and substitution together with the rules that follow from the semantic explanations. Next, the function types are introduced with their semantic explanation and the formal rules which the explanation justifies. The rules are formulated in the style of natural deduction [36].

3.1 Types

The basic notion in Martin-Löf's type theory is the notion of type. A type is explained by saying what an object of the type is and what it means for two objects of the type to be identical. This means that we can make the judgement

$$A \text{ is a type},$$

which we in the formal system write as

$$A \ \mathit{type},$$

when we know the conditions for asserting that something is an object of type A and when we know the conditions for asserting that two objects of type A are identical. We require that the conditions for identifying two objects must define an equivalence relation.

When we have a type, we know from the semantic explanation of what it means to be a type what the conditions are to be an object of that type.

So, if A is a type and we have an object a that satisfies these conditions then

$$a \text{ is an object of type } A,$$

which we formally write

$$a \in A.$$

Furthermore, from the semantics of what it means to be a type and the knowledge that A is a type we also know the conditions for two objects of type A to be identical. Hence, if A is a type and a and b are objects of type A and these objects satisfies the equality conditions in the semantic explanation of A then

$$a \text{ and } b \text{ are identical objects of type } A,$$

which we write

$$a = b \in A.$$

Two types are equal when an arbitrary object of one type is also an object of the other and when two identical objects of one type are identical objects of the other. If A and B are types we know the conditions for being an object and the conditions for being identical objects of these types. Then we can investigate if all objects of type A are also objects of type B and if all identical objects of type A are also objects of type B and vice versa. If these conditions are satisfied then

$$A \text{ and } B \text{ are identical types,}$$

which we formally write

$$A = B.$$

The requirement that the equality between objects of a type must be an equivalence relation is formalized by the rules:

Reflexivity of objects

$$\frac{a \in A}{a = a \in A}$$

Symmetry of objects

$$\frac{a = b \in A}{b = a \in A}$$

Transitivity of objects

$$\frac{a = b \in A \qquad b = c \in A}{a = c \in A}$$

The corresponding rules for types are easily justified from the meaning of what it means to be a type.

Reflexivity of types

$$\frac{A \; type}{A = A}$$

Symmetry of types

$$\frac{A = B}{B = A}$$

Transitivity of types

$$\frac{A = B \qquad B = C}{A = C}$$

The meaning of the judgement forms $a \in A$, $a = b \in A$ and $A = B$ immediately justifies the rules

Type equality rules

$$\frac{a \in A \qquad A = B}{a \in B} \qquad\qquad \frac{a = b \in A \qquad A = B}{a = b \in B}$$

3.2 Hypothetical judgements

The judgements we have introduced so far do not depend on any assumptions. In general, a hypothetical judgement is made in a *context* of the form

$$x_1 \in A_1, \; x_2 \in A_2, \; \ldots, \; x_n \in A_n$$

where we already know that A_1 is a type, A_2 is a type in the context $x_1 \in A_1$, ..., and A_n is a type in the context $x_1 \in A_1$, $x_2 \in A_2$, ..., $x_{n-1} \in A_n$. The explanations of hypothetical judgements are made by induction on the length of a context. We have already given the meaning of the judgement forms in the empty context; hence we could now directly explain the judgement forms in a context of length n. However, in order not to hide the explanations by heavy notation, we will give them for hypothetical judgements only depending on one assumption and then illustrate the general case with the judgement that A is a type in a context of length n.

Let C be a type which does not depend on any assumptions. That A is a type when $x \in C$, which we write

$$A \; type \; [x \in C],$$

means that, for an arbitrary object c of type C, $A\,[x \leftarrow c]$ is a type, that is, A is a type when c is substituted for x. Furthermore we must also know that if c and d are identical objects of type C then $A\,[x \leftarrow c]$ and $A\,[x \leftarrow d]$ are the same types. When A is a type depending on $x \in C$ we say that A is *a family of types over the type* C.

That A and B are identical families of types over the type C,

$$A = B\ [x \in C],$$

means that $A\,[x \leftarrow c]$ and $A\,[x \leftarrow c]$ are equal types for an arbitrary object c of type C.

That a is an object of type A when $x \in C$,

$$a \in A\ [x \in C],$$

means that we know that $a\,[x \leftarrow c]$ is an object of type $A\,[x \leftarrow c]$ for an arbitrary object c of type C. We must also know that $a\,[x \leftarrow c]$ and $a\,[x \leftarrow d]$ are identical objects of type $A\,[x \leftarrow c]$ whenever c and d are identical objects of type C.

That a and b are identical objects of type A depending on $x \in C$,

$$a = b \in A\ [x \in C],$$

means that $a\,[x \leftarrow c]$ and $b\,[x \leftarrow c]$ are the same objects of type $A\,[x \leftarrow c]$ for an arbitrary object c of type C.

We will illustrate the general case by giving the meaning of the judgement that A is a type in a context of length n; the other hypothetical judgements are explained in a similar way. We assume that we already know the explanations of the judgement forms in a context of length $n-1$. Let

$$x_1 \in A_1,\ x_2 \in A_2,\ \ldots,\ x_n \in A_n$$

be a context of length n. We then know that

$$\begin{array}{l} A_1\ \mathit{type} \\ A_2\ \mathit{type}\ [x_1 \in A_1] \\ \qquad\quad \vdots \\ A_n\ \mathit{type}\ [x_1 \in A_1,\ x_2 \in A_2,\ \ldots,\ x_{n-1} \in A_{n-1}] \end{array}$$

To know the hypothetical judgement

$$A\ \mathit{type}\ [x_1 \in A_1,\ x_2 \in A_2,\ \ldots,\ x_n \in A_n]$$

means that we know that the judgement

$$\begin{array}{l} A\,[x_1 \leftarrow a]\ \mathit{type} \\ \quad [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]] \end{array}$$

holds for an arbitrary object a of type A_1 in the empty context. We must also require that if a and b are arbitrary identical objects of type A_1 then the judgement

$$A\,[x_1 \leftarrow a] = A\,[x_1 \leftarrow b] \\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]$$

holds. This explanation justifies two rules for substitution of objects in types. We can formulate these rules in different ways, simultaneously substituting objects for one or several of the variables in the context. Formulating the rules so they follow the semantical explanation as closely as possible gives us:

Substitution in types

$$\frac{A\ \mathit{type}\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a \in A_1}{A\,[x_1 \leftarrow a]\ \mathit{type}\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

$$\frac{A\ \mathit{type}\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a = b \in A_1}{A\,[x_1 \leftarrow a] = A\,[x_1 \leftarrow b]\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

The explanations of the other hypothetical judgement forms give the following substitution rules. Let A and B be types in the context $x_1 \in A_1$, $x_2 \in A_2,\ \ldots,\ x_n \in A_n$.

Substitution in equal types

$$\frac{A = B\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a \in A_1}{A\,[x_j \leftarrow a] = B\,[x_j \leftarrow a]\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

Let A be a type in the context $x_1 \in A_1,\ x_2 \in A_2,\ \ldots,\ x_n \in A_n$.

Substitution in objects

$$\frac{a \in A\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a \in A_1}{a\,[x_1 \leftarrow a] \in A\,[x_1 \leftarrow a]\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

Let A be a type and c and d be objects of type A in the context $x_1 \in A_1$, $x_2 \in A_2,\ \ldots,\ x_n \in A_n$.

Substitution in equal objects

$$\frac{c = d \in A\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a \in A_1}{c\,[x_1 \leftarrow a] = d\,[x_1 \leftarrow a] \in A\,[x_1 \leftarrow a]\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

The explanations of the hypothetical judgement forms justifies the following rule for introducing assumptions.

Assumption

$$\frac{\begin{array}{ll} A_1 \ \mathit{type} & \\ A_2 \ \mathit{type} & [x_1 \in A_1] \\ \quad\vdots & \\ A_n \ \mathit{type} & [x_1 \in A_1, \ldots, x_{n-1} \in A_{n-1}] \\ A \ \mathit{type} & [x_1 \in A_1, \ldots, x_{n-1} \in A_{n-1}, x_n \in A_n] \end{array}}{x \in A \ [x_1 \in A_1, \ldots x_n \in A_n, x \in A]}$$

In this rule all premises are explicit. In order to make the rules shorter and more comprehensible we will often leave out that part of the context which is the same in the conclusion and each premise.

The rules given in the previous section without assumptions could be justified also for hypothetical judgements.

3.3 Function types

One of the basic ways to form a new type from old ones is to form a function type. So, if we we have a type A and a family B of types over A, we want to form the dependent function type $(x \in A)B$ of functions from A to B. In order to do this, we must explain what it means to be an object of type $(x \in A)B$ and what it means for two objects of type $(x \in A)B$ to be identical. The function type is explained in terms of application.

To know that an object c is of type $(x \in A)B$ means that we know that when we apply it to an arbitrary object a of type A we get an object $c(a)$ in $B\,[x \leftarrow a]$ and that we get identical objects in $B\,[x \leftarrow a]$ when we apply it to identical objects a and b of A.

That two objects c and d of $(x \in A)B$ are identical means that when we apply them on an arbitrary object a of type A we get identical objects of type $B\,[x \leftarrow a]$.

Since we now have explained what it means to be an object of a function type and the conditions for two objects of a function type to be equal, we can justify the rule for forming the function type.

Function type

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

We also obtain the rule for forming equal function types.

Equal function types

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

We will use the abbreviation $(A)B$ for $(x \in A)B$ when B does not depend on x. We will also write $(x \in A; y \in B)C$ instead of $(x \in A)(y \in B)C$ and $(x, y \in A)B$ instead of $(x \in A; y \in A)C$.

We can also justify the following two rules for application

Application

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

We also have the following rules for showing that two functions are equal.

Application

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

Extensionality

$$\frac{c \in (x \in A)B \quad d \in (x \in A)B \quad c(x) = d(x) \in B\ [x \in A]}{c = d \in (x \in A)B}$$

x must occur free neither in c nor in d

Instead of writing repeated applications as $c(a_1)(a_2)\cdots(a_n)$ we will use the simpler form $c(a_1, a_2, \ldots, a_n)$.

One fundamental way to introduce a function is to abstract a variable from an expression:

Abstraction

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

We will write repeated abstractions as $[x_1, x_2, \ldots, x_n]b$ and also exclude the outermost parentheses when there is no risk of confusion.

How do we know that this rule is correct, i.e. how do we know that $[x]b$ is a function of the type $(x \in A)B$? By the semantics of function types, we must know that when we apply $[x]b$ of type $(x \in A)B$ on an object a of type A, then we get an object of type $B[x \leftarrow a]$; the explanation is by β-conversion:

β-conversion

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

We must also know that when we apply an abstraction $[x]b$, where $b \in B\ [x \in A]$, on two identical objects a_1 and a_2 of type A, then we get identical results of type $B\,[x \leftarrow a]$ as results. We can see this in the following way. By β-conversion we know that $([x]b)(a_1) = b\,[x \leftarrow a_1] \in B\,[x \leftarrow a_1]$ and $([x]b)(a_2) = b\,[x \leftarrow a_2] \in B\,[x \leftarrow a_2]$. By the meaning of the judgements B *type* $[x \in A]$ and $b \in B\ [x \in A]$ we know that $B[x \leftarrow a_1] = B[x \leftarrow a_2]$ and that $b\,[x \leftarrow a_1] = b\,[x \leftarrow a_2] \in B\,[x \leftarrow a_1]$. Hence, by symmetry and transitivity, we get $([x]b)(a_1) = ([x]b(a_2) \in B\,[x \leftarrow a_1]$ from $a_1 = a_2 \in A$.

To summarize: to be an object f in a functional type $(x \in A)B$ means that it is possible to make an application $f(a)$ if $a \in A$. Then by looking at β-conversion as the definition of what it means to apply an abstracted expression to an object it is possible to give a meaning to an abstracted expression. Hence, application is more primitive then abstraction on this type level. Later we will see that for the *set* of functions, the situation is different.

By the rules we have introduced, we can derive the rules

η-conversion

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

x must not occur free in c

ξ-rule

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

3.4 The type Set

The objects in the type Set consist of inductively defined sets. In order to explain a type we have to explain what it means to be an object in it and what it means for two such objects to be the same. So, to know that Set is a type we must explain what a set is and what it means for two sets to be the same: to know that A is an object in Set (or equivalently that A is a set) is to know how to form canonical elements in A and when two canonical elements are equal. A canonical element is an element on constructor form; examples are zero and the successor function for natural numbers.

Two sets are the same if an element of one of the sets is also an element of the other and if two equal elements of one of the sets are also equal elements of the other.

This explanation justifies the following rule

Set-formation

$$\mathsf{Set}\ \textit{type}$$

If we have a set A we may form a type $El(A)$ whose objects are the elements of the set A:

El-formation

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

Notice that it is required that the sets are built up inductively: we know exactly in what ways we can build up the elements in the set and different ways corresponding to different constructors. As an example, for the ordinary definition of natural numbers, there are precisely two ways of building up elements, one using zero and the other one using the successor function. This is in contrast with types which in general are not built up inductively. For instance, the type Set can obviously not be defined inductively. It is always possible to introduce new sets (i.e. objects in Set). The concept of type is open; it is possible to add more types to the language, for instance by adding a new object A to Set gives the new type $\mathsf{El}(A)$).

In the sequel, we will often write A instead of $\mathit{El}(A)$ since it will always be clear from the context if A stands for the set A or the type of elements of A.

3.5 Definitions

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: it is in this way rules and axioms of a theory is represented in the framework.

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 and not a definition; instead it gets its meaning by the semantics of the theory. Such a constant is also called a constructor. Examples of primitive constants are N, succ and $\mathsf{0}$; they can be introduced by the following declarations:

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

A defined constant is defined in terms of other objects. When we apply a defined constant to all its arguments in an empty context, for instance, $c(e_1, \ldots, e_n)$, then we get an expression which is a definiendum, that is, 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 it as an abbreviation of an object a in a type A:

$$c = a \in A$$

For instance, we can make the following explicit definitions:

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

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 see 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. This is done by pattern-matching and the definition may be recursive. Since it is not decidable if an expression defined by pattern-matching on a set really defines a value for each element of the set, the correctness of an implicit definition is in general a semantical issue. We must be sure that all well-typed expressions of the form $c(e_1, \ldots, e_n)$ is a definiendum with a unique well-typed definiens. Here are two examples, addition and the operator for primitive recursion in arithmetic:

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

4 Propositional logic

Type theory can be used as a logical framework, that is, it can be used to represent different theories. In general, a theory is presented by a list of typings

$$c_1 \in A_1, \ldots, \; c_n \in A_n$$

where $c_1, \ldots, c_n$ are new primitive constants, and a list of definitions

$$d_1 = e_1 \in B_1, \ldots, \; d_m = e_m \in A_m$$

where $d_1, \ldots, d_m$ are new defined constants.

The basic types of Martin-Löf's type theory are Set and the types of elements of the particular sets we introduce. In the next section we will give a number of examples of sets, but first we use the idea of propositions as sets to express that propositional logic with conjunction and implication; the other connectives can be introduced in the same way. When viewed as the type of propositions, the semantics of Set can be seen as the constructive explanation of propositions: a proposition is defined by laying down what counts as a direct (or canonical) proof of it; or differently expressed: a proposition is defined by its introduction rules. Given a proposition A, that is, an object of the type Set, then $El(A)$ is the type of proofs of A. From the semantics of sets we get that two proofs are the same if they have the same form and identical parts; we also get that two propositions are the same if a proof of one of the propositions is also a proof of the other and if identical proofs of one of the propositions are also identical proofs of the other.

The primitive constant & for conjunction is introduced by the following declaration

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

From this declaration we obtain, by repeated function application, the clause for conjunction in the usual inductive definition of formulas in the propositional calculus:

& -formation

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set}}{A \& B \in \mathsf{Set}}$$

where we have used infix notation, that is, we have written $A\&B$ instead of $\&(A, B)$.

We must now define what counts as a proof of a conjunction, and that is done by the following declaration of the primitive constant $\&_I$.

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

This declaration is the inductive definition of the set $\&(A, B)$, that is, all elements in the set is equal to an element of the form $\&_I(A, B, a, b)$, where A and B are sets and $a \in A$ and $b \in B$. A proof of the syntactical form $\&_I(A, B, a, b)$ is called a *canonical proof* of $A\&B$.

By function application, we obtain the introduction rule for conjunction from the declaration of $\&_I$.

& -introduction

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad a \in A \qquad b \in B}{\&_I(A, B, a, b) \in A\&B}$$

To obtain the two elimination rules for conjunction, we introduce the two defined constants

$$\&_{E1} \in (A, B \in \mathsf{Set}; A\&B)A$$

and

$$\&_{E2} \in (A, B \in \mathsf{Set}; A\&B)B$$

by the defining equations

$$\&_{E1}(A, B, \&_I(A, B, a, b)) = a \in A$$

and

$$\&_{E2}(A, B, \&_I(A, B, a, b)) = b \in B$$

respectively. Notice that it is the definition of the constants which justifies their typings. To see that the typing of $\&_{E1}$ is correct, assume that A and B are sets, and that $p \in A\&B$. We must then show that $\&_{E1}(A, B, p)$ is an element in A. But since $p \in A\&B$, we know that p is equal to an element of the form $\&_I(A, B, a, b)$, where $a \in A$ and $b \in B$. But then we have that $\&_{E1}(A, B, p) = \&_{E1}(A, B, \&_I(A, B, a, b))$ which is equal to a by the defining equation of $\&_{E1}$.

From the typings of $\&_{E1}$ and $\&_{E2}$ we obtain, by function application, the elimination rules for conjunction:

& -elimination 1

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad c \in A\&B}{\&_{E1}(A, B, c) \in A}$$

and

& -elimination 2

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad c \in A\&B}{\&_{E2}(A, B, c) \in B}$$

The defining equations for $\&_{E1}$ and $\&_{E2}$ correspond to Prawitz' reduction rules in natural deduction:

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

and

$$\frac{\dfrac{\begin{array}{c}\vdots\\ B\end{array}}{A\&B}}{B} \quad = \quad \begin{array}{c}\vdots\\ B\end{array}$$

respectively. Notice the role which these rules play here. They are used to justify the correctness, that is, the well typings of the elimination rules. The elimination rules are looked upon as methods which can be executed, and it is the reduction rules which defines the execution of the elimination rules.

The primitive constant $\supset$ for implication is introduced by the declaration

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

As for conjunction, we obtain from this declaration the clause for implication in the inductive definition of formulas in the propositional calculus:

$\supset$ -formation

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set}}{A \supset B \in \mathsf{Set}}$$

A canonical proof of an implication is formed by the primitive constant $\supset_I$, declared by

$$\supset_I \in (A, B \in \mathsf{Set}; (A)B)A \supset B$$

By function application, the introduction rule for implication is obtained from the declaration of $\supset_I$:

$\supset$ -introduction

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad b(x) \in B\ [x \in A]}{\supset_I (A, B, b) \in A \supset B}$$

So, to get a canonical proof of $A \supset B$ we must have a function b which when applied on a proof of A gives a proof of B, and the proof then obtained is $\supset_I (A, B, b)$.

To obtain modus ponens, the elimination rule for $\supset$, we introduce the defined constant

$$\supset_E \in (A, B \in \mathsf{Set}; A \supset B; A)B$$

which is defined by the equation

$$\supset_E (A, B, \supset_I (A, B, b, a)) = b(a) \in B$$

In the same way as for conjunction, we can use this definition to show that $\supset_E$ is well-typed.

The defining equation corresponds to the reduction rule

$$\dfrac{\dfrac{\begin{array}{c}A\\ \vdots\\ B\end{array}}{A \supset B} \qquad \begin{array}{c}\vdots\\ A\end{array}}{B} \quad = \quad \begin{array}{c}\vdots\\ A\\ \vdots\\ B\end{array}$$

By function application, we obtain from the typing of $\supset_E$

$\supset$ -elimination

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad b \in A \supset B \qquad a \in A}{\supset_E (A, B, b, a) \in B}$$

5 Set theory

We will in this section introduce a theory of sets with natural numbers, lists, functions, etc. which could be used when specifying and implementing computer programs. We will also show how this theory is represented in the type theory framework.

When defining a set, we first introduce a primitive constant for the set and then give the primitive constants for the constructors, which express the different ways elements of the set can be constructed. The typing rule for constant denoting the set is called the *formation rule* of the set and the typing rules for the constructors are called the *introduction rules*. Finally, we introduce a selector as an implicitly defined constant to express the induction principle of the set; the selector is defined by pattern-matching and may be recursive. The type rule for the selector is called the *elimination rule* and the defining equations are called *equality rules*.

Given the introduction rules, it is possible to mechanically derive the elimination rule and the equality rules for a set; how this can be done have been investigated by Martin-Löf [29], Backhouse [2], Coquand and Paulin [9], and Dybjer[14].

5.1 The set of Boolean values

The set of Boolean values is an example of an enumeration set. The values of an enumeration set are exactly the same as the constructors of the set and all constructors yield different elements. For the Booleans this means that there are two ways of forming an element and therefore also two constructors; true and false. Since we have given the elements of the set and their equality, we can introduce a constant for the set and make the type declaration

$$\mathsf{Bool} \in \mathsf{Set}$$

We can also declare the types of the constructor constants

$$\mathsf{true} \in \mathsf{Bool}$$
$$\mathsf{false} \in \mathsf{Bool}$$

The principal selector constant of an enumeration set is a function that performs case analysis on Boolean values. For the Booleans we introduce the if constant with the type

$$\mathsf{if} \in (C \in (\mathsf{Bool})\mathsf{Set}; b \in \mathsf{Bool}; C(\mathsf{true}); C(\mathsf{false}))\, C(b)$$

and the defining equations

$$\mathsf{if}(C, \mathsf{true}, a, b) = a$$
$$\mathsf{if}(C, \mathsf{false}, a, b) = b$$

In these two definitional equalities we have omitted the types since they can be obtained immediately from the typing of if. In the sequel, we will often write just $a = b$ instead of $a = b \in A$ when the type A is clear from the context.

5.2 The empty set

To introduce the empty set, $\{\}$, we just define a set with no constructors at all. First we make a type declaration for the set

$$\{\} \in \mathsf{Set}$$

Since there are no constructors we immediately define the selector case and its type by the declaration

$$\mathsf{case} \in (C \in (\{\})\mathsf{Set}; a \in \{\})\, C(a)$$

The empty set corresponds to the absurd proposition and the selector corresponds to the natural deduction rule for absurdity

$$\frac{\bot\ \mathit{true} \qquad C\ \mathit{prop}}{C\ \mathit{true}}$$

5.3 The set of natural numbers

In order to introduce the set of natural numbers, N, we must give the rules for forming all the natural numbers as well as all the rules for forming two equal natural numbers. These are the introduction rules for natural numbers.

There are two ways of forming natural numbers, $\mathsf{0}$ is a natural number and if n is a natural number then $\mathsf{succ}(n)$ is a natural number. There are also two corresponding ways of forming equal natural numbers, the natural number $\mathsf{0}$ is equal to $\mathsf{0}$ and if the natural number n is equal to m, then $\mathsf{succ}(n)$ is equal to $\mathsf{succ}(m)$. So we have explained the meaning of the natural numbers as a set, and can therefore make the type declaration

$$\mathsf{N} \in \mathsf{Set}$$

and form the introduction rules for the natural numbers, by declaring the types of the constructor constants $\mathsf{0}$ and succ

$$\mathsf{0} \in \mathsf{N}$$

$$\mathsf{succ} \in (n \in \mathsf{N})\,\mathsf{N}$$

The general rules in the framework makes it possible to give the introduction rules in this simple form.

We will introduce a very general form of selector for natural numbers, natrec, as a defined constant. It could be used both for expressing elements by primitive recursion and proving properties by induction. The functional constant natrec takes four arguments; the first is a family of sets that determines the set which the result belongs to, the second and third are the results for the zero and successor case, respectively, and the fourth argument, finally, is the natural number which is the principal argument of the selector. Formally, the type of natrec is

$$\begin{array}{ll}\mathsf{natrec} \in & (C \in (\mathsf{N})\,\mathsf{Set}; \\ & \;d \in C(\mathsf{0}); \\ & \;e \in (x \in \mathsf{N}; y \in C(x))\,C(\mathsf{succ}(x)); \\ & \;n \in \mathsf{N}) \\ & C(n)\end{array}$$

The defining equations for the natrec constant are

$$\mathsf{natrec}(C, d, e, \mathsf{0}) = d$$

$$\mathsf{natrec}(C, d, e, \mathsf{succ}(m)) = e(n, \mathsf{natrec}(C, d, e, m))$$

The selector for natural numbers could, as we already mentioned, be used for introducing ordinary primitive recursive functions. Addition and multiplication could, for example, be introduced as two defined constants

$$\mathsf{plus} \in (\mathsf{N};\mathsf{N})\,\mathsf{N}$$

$$\mathsf{mult} \in (\mathsf{N};\mathsf{N})\,\mathsf{N}$$

by the defining equations

$$\mathsf{plus}(m,n) = \mathsf{natrec}([x]\mathsf{N}, n, [x,y]\mathsf{succ}(y), m)$$

$$\mathsf{mult}(m,n) = \mathsf{natrec}([x]\mathsf{N}, \mathsf{0}, [x,y]\mathsf{plus}(y,n), m)$$

Using the rules for application together with the type and the definitional equalities for the constant natrec it is easy to derive the type of the right hand side of the equalities above as well as the following equalities for addition and multiplication:

$$\begin{array}{l} \mathsf{plus}(\mathsf{0},n) = n \in \mathsf{N} \quad [n \in \mathsf{N}] \\ \mathsf{plus}(\mathsf{succ}(m),n) = \mathsf{succ}(\mathsf{plus}(m,n)) \in \mathsf{N} \quad [m \in \mathsf{N}, n \in \mathsf{N}] \end{array}$$

$$\begin{array}{l} \mathsf{mult}(\mathsf{0},n) = \mathsf{0} \in \mathsf{N} \quad [n \in \mathsf{N}] \\ \mathsf{mult}(\mathsf{succ}(m),n) = \mathsf{plus}(\mathsf{mult}(m,n),n) \in \mathsf{N} \quad [m \in \mathsf{N}, n \in \mathsf{N}] \end{array}$$

In general, if we have a primitive recursive function f from N to A

$$\begin{array}{rcl} f(\mathsf{0}) & = & d \\ f(\mathsf{succ}(n)) & = & e(n, f(n)) \end{array}$$

where $d \in A$ and e is a function in $(\mathsf{N};A)A$, we can introduce it as a defined constant

$$f' \in (\mathsf{N})A$$

using the defining equation

$$f'(n) = \mathsf{natrec}([x]A, d', e', n)$$

where d' and e' are functions in type theory which correspond to d and e in the definition of f.

The type of the constant natrec represents the usual elimination rule for natural numbers

$$\frac{\begin{array}{l} C(x)\ \mathsf{Set}\ [x \in \mathsf{N}] \\ d \in C(\mathsf{0}) \\ e \in C(\mathsf{succ}(x))\ [x \in \mathsf{N}, y \in C(x)] \\ x \in \mathsf{N} \end{array}}{\mathsf{natrec}(C,d,e,x) \in C(x)}$$

which can be obtained by assuming the arguments and then apply the constant natrec on them. Note that, in the conclusion of the rule, the expression $\mathsf{natrec}(C,d,e,x)$ contains the family C. This is a consequence of the explicit declaration of natrec in the framework.

5.4 The set of functions (Cartesian product of a family of sets)

We have already introduced the *type* $(x \in A)B$ of functions from the type A to the type B. We need the corresponding *set* of functions from a set to another set. If we have a set A and a family of sets B over A, we can form the cartesian product of a family of sets, which is denoted $\Pi(A, B)$. The elements of this set are functions which when applied to an element a in A yield an element in $B(a)$. The elements of the set $\Pi(A, B)$ are formed by applying the constructor λ to the sets A and B and an object of the corresponding function type.

The constant Π is introduced by the type declaration

$$\Pi \in (A \in \mathsf{Set}; B \in (x \in A)\mathsf{Set})\, \mathsf{Set}$$

and the constant λ by

$$\lambda \in (A \in \mathsf{Set}; B \in (x \in A)\, \mathsf{Set}; f \in (x \in A)B)\, \Pi(A, B)$$

These constant declarations correspond to the rules

$$\frac{A \in \mathsf{Set} \qquad B(x) \in \mathsf{Set}\ [x \in A]}{\Pi(A, B) \in \mathsf{Set}}$$

$$\frac{A \in \mathsf{Set} \qquad B(x) \in \mathsf{Set}\ [x \in A] \qquad f \in B(x)\ [x \in A]}{\lambda(A, B, f) \in \Pi(A, B)}$$

Notice that the elements of a cartesian product of a family of sets, $\Pi(A, B)$, are more general than ordinary functions from A to B in that the result of applying an element of $\Pi(A, B)$ to an argument can be in a set which may depend on the value of the argument.

The most important defined constant in the Π-set is the constant for application. In type theory this selector takes as arguments not only an element of $\Pi(A, B)$ and an object of type A but also the sets A and B themselves. The constant is introduced by the type declaration

$$\mathsf{apply} \in (A \in \mathsf{Set}; B \in (x \in A)\mathsf{Set}; g \in \Pi(A, B); a \in A)\, B(a)$$

and the definitional equality

$$\mathsf{apply}(A, B, \lambda(f), a) = f(a)$$

The cartesian product of a family of sets is, when viewed as a proposition the same as universal quantification. The type of the constructor corresponds to the introduction rule

$$\frac{B(x)\ \mathit{true}\ [x \in A]}{(\forall\, x \in A)\, B(x)\ \mathit{true}}$$

and the type of the selector corresponds to the elimination rule

$$\frac{(\forall\, x \in A)\, B(x) \ \ true \qquad a \in A}{B(a) \ \ true}$$

The cartesian product of a family of sets is a generalization of the ordinary function set. If the family of sets B over A is the same for all elements of A, then the cartesian product is just the set of ordinary functions. The constant $\rightarrow$ is introduced by the following explicit definition:

$$\rightarrow \in (A, B \in \mathsf{Set})\, \mathsf{Set}$$
$$\rightarrow = [A, B]\Pi(A, [x]B)$$

The set of functions is, when viewed as a proposition, the same as implication since the type of the constructor is the same as the introduction rule for implication

$$\frac{B \ \ true\ [A \ \ true]}{A \supset B \ \ true}$$

and the type of the selector is the same as the elimination rule

$$\frac{A \supset B \ \ true \qquad A \ \ true}{B \ \ true}$$

Given the empty set and the set of function we can define a constant for negation in the following way

$$\neg \in (A \in \mathsf{Set})\, \mathsf{Set}$$
$$\neg(A) = A {\rightarrow} \{\}$$

Example 5.4.1. Let us see how to prove the proposition $A \supset \neg\neg A$. In order to prove the proposition we must find an element in the set

$$A \rightarrow (\neg(\neg A)) \ \equiv \ A \rightarrow ((A \rightarrow \{\}) \rightarrow \{\})$$

We start by making the assumptions $x \in A$ and $y \in A \rightarrow \{\}$ and then obtain an element in $\{\}$

$$\mathsf{apply}(A \rightarrow \{\}, A, y, x) \in \{\}$$

and therefore

$$\begin{array}{l} \lambda(A, (A \rightarrow \{\}) \rightarrow \{\}, \\ \quad [x]\lambda(A \rightarrow \{\}, \{\}, \\ \qquad\quad [y]\mathsf{apply}(A \rightarrow \{\}, A, y, x))) \\ \in A \rightarrow ((A \rightarrow \{\}) \rightarrow \{\}) \ \equiv \ A \rightarrow (\neg(\neg A)) \end{array}$$

Example 5.4.2. Using the rules for natural numbers, booleans and functions we will show how to define a function, $\mathsf{eqN} \in (\mathsf{N}, \mathsf{N})\mathsf{Bool}$, that decides if two natural numbers are equal. We want the following equalities to hold:

$$
\begin{aligned}
\mathsf{eqN}(0,0) &= \mathsf{true}\\
\mathsf{eqN}(0,\mathsf{succ}(n)) &= \mathsf{false}\\
\mathsf{eqN}(\mathsf{succ}(m),0) &= \mathsf{false}\\
\mathsf{eqN}(\mathsf{succ}(m),\mathsf{succ}(n)) &= \mathsf{eqN}(m,n)
\end{aligned}
$$

It is impossible to define eqN directly just using natural numbers and recursion on the arguments. We have to do recursion on the arguments separately and first use recursion on the first argument to compute a *function* which when applied to the second argument gives us the result we want. So we first define a function $f \in (\mathsf{N})\,(\mathsf{N} \to \mathsf{Bool})$ which satisfies the equalities

$$
\begin{aligned}
f(0) &= \lambda([m]\,\mathsf{iszero}(m))\\
f(\mathsf{succ}(n)) &= \lambda([m]\,\mathsf{natrec}(m,\mathsf{false},[x,y]\,\mathsf{apply}(f(n),x)))
\end{aligned}
$$

where

$$
\mathsf{iszero}(m) = \mathsf{natrec}(m,\mathsf{true},[x,y]\,\mathsf{false})
$$

If we use the recursion operator explicitly, we can define f as

$$
\begin{aligned}
f(n) = \mathsf{natrec}(n,\ & \\
& \lambda([m]\mathsf{iszero}(m)),\\
& [u,v]\,\lambda((m)\mathsf{natrec}(m,\mathsf{false},[x,y]\,\mathsf{apply}(v,x))))
\end{aligned}
$$

The function f is such that $f(n)$ is equal to a function which gives true if is applied to n and false otherwise, that is, we can use it to define eqN as follows

$$
\mathsf{eqN}(m,n) = \mathsf{apply}(f(m),n)
$$

It is a simple exercise to show that

$$
\mathsf{eqN} \in (\mathsf{N},\mathsf{N})\,\mathsf{Bool}
$$

and that it satisfies the equalities we want it to satisfy.

5.5 Propositional equality

The equality on the judgement level $a = b \in A$ is a definitional equality and two objects are equal if they have the same normal form. In order to express that, for example, addition of natural numbers is a commutative operation it is necessary to introduce a set for propositional equality.

If a and b are elements in the set A, then $\mathsf{Id}(A, a, b)$ is a set. We express this by introducing the constant Id and its type

$$\mathsf{Id} \in (X \in \mathsf{Set}; a \in X; b \in X)\, \mathsf{Set}$$

The only constructor of elements in equality sets is id and it is introduced by the type declaration

$$\mathsf{id} \in (X \in \mathsf{Set}; x \in X)\, \mathsf{Id}(X, x, x)$$

To say that id is the only constructor for $\mathsf{Id}(A, a, b)$ is the same as to say that $\mathsf{Id}(A, a, b)$ is the least reflexive relation. Transitivity, symmetry and congruence can be proven from this definition. We use the name idpeel for the selector and it is introduced by the type declaration

$$\begin{array}{ll} \mathsf{idpeel} \in & (A \in \mathsf{Set}; \\ & \;C \in (x, y \in A; e \in \mathsf{Id}(A, x, y))\, \mathsf{Set}; \\ & \;a, b \in A; \\ & \;e \in \mathsf{Id}(A, a, b); \\ & \;d \in (x \in A)\, C(x, x, \mathsf{id}(A, x))) \\ & C(a, b, e) \end{array}$$

and the equality

$$\mathsf{idpeel}(A, C, a, b, \mathsf{id}(A, a), d) = d(a)$$

The intuition behind this constant is that it expresses a substitution rule for elements which are propositionally equal.

Example 5.5.3. The type of the constructor in the set $\mathsf{Id}(A, a, b)$ corresponds to the reflexivity rule of equality. The symmetry and transitivity rules can easily be derived.

Let A be a set and a and b two elements of A. Assume that

$$d \in \mathsf{Id}(A, a, b)$$

In order to prove symmetry, we must construct an element in $\mathsf{Id}(A, b, a)$. By applying idpeel on A, $[x, y, e]\mathsf{Id}(A, y, x)$, a, b, d and $[x]\mathsf{id}(A, x)$ we get, by simple typechecking, an element in the set $\mathsf{Id}(A, b, a)$.

$$\mathsf{idpeel}(A, [x, y, e]\mathsf{Id}(A, y, x), a, b, d, [x]\mathsf{id}(A, x)) \in \mathsf{Id}(A, b, a)$$

The derived rule for symmetry can therefore be expressed by the constant symm defined by

$$\mathsf{idsymm} \in (A \in \mathsf{Set}; a, b \in A; d \in \mathsf{Id}(A, a, b))\, \mathsf{Id}(A, b, a)$$

$$\mathsf{idsymm}(A, a, b, d) = \mathsf{idpeel}(A, [x, y, e]\mathsf{Id}(A, y, x), a, b, d, [x]\mathsf{id}(A, x))$$

Transitivity is proved in a similar way. Let A be a set and a, b and c elements of A. Assume that

$$d \in \mathsf{Id}(A, a, b) \text{ and } e \in \mathsf{Id}(A, b, c)$$

By applying idpeel on A, $[x, y, z]\mathsf{Id}(A, y, c) \rightarrow \mathsf{Id}(A, z, c)$, a, b, d and the identity function $[x]\lambda(\mathsf{Id}(A, x, c), \mathsf{Id}(A, x, c), [w]w)$ we get an element in the set $\mathsf{Id}(A, b, c) \rightarrow \mathsf{Id}(A, a, c)$. This element is applied on e in order to get the desired element in $\mathsf{Id}(A, a, c)$.

$$\mathsf{idtrans} \in (A \in \mathsf{Set}; a, b, c \in A; d \in \mathsf{Id}(A, a, b); e \in \mathsf{Id}(A, b, c))\, \mathsf{Id}(A, a, c)$$

$$\begin{aligned}\mathsf{idtrans}(A, a, b, c, d, e) = \mathsf{apply}(&\mathsf{Id}(A, b, c), \mathsf{Id}(A, a, c),\\ &\mathsf{idpeel}(A, [x, y, z]\mathsf{Id}(A, y, c) \rightarrow \mathsf{Id}(A, x, c),\\ &\qquad a, b, d,\\ &\qquad [x]\lambda(\mathsf{Id}(A, x, c), \mathsf{Id}(A, x, c), [w]w)),\\ &e)\end{aligned}$$

Example 5.5.4. Let us see how we can derive a rule for substitution in set expressions. We want to have a rule

$$\frac{P(x) \in \mathsf{set}\ [x \in A] \qquad a \in A \qquad b \in A \qquad c \in \mathsf{Id}(A, a, b) \qquad p \in P(a)}{\mathsf{subst}(P, a, b, c, p) \in P(b)}$$

To derive such a rule, first assume that we have a set A and elements a and b of A. Furthermore assume that $c \in \mathsf{Id}(A, a, b)$, $P(x) \in \mathsf{Set}\ [x \in A]$ and $p \in P(a)$. Type checking gives us that

$$\lambda(P(x), P(x), [w]w) \in P(x) \rightarrow P(x)\ [x \in A]$$

$$\begin{aligned}&\mathsf{idpeel}(A, [x, y, z](P(x) \rightarrow P(y)), a, b, c, [x]\lambda(P(x), P(x), [w]w))\\ &\in P(a) \rightarrow P(b)\end{aligned}$$

We can now apply the function above on p to obtain an element in $P(b)$. So we can define a constant subst that expresses the substitution rule above. The type of subst is

$$\begin{aligned}\mathsf{subst} \in\ &(A \in \mathsf{Set};\\ &\ P \in (A)\mathsf{Set};\\ &\ a, b \in A;\\ &\ c \in \mathsf{Id}(A, a, b);\\ &\ p \in P(a))\\ &P(b)\end{aligned}$$

and the defining equation

$$\begin{aligned}\mathsf{subst}(A, P, a, b, c, p) = \mathsf{apply}(&P(a), P(b),\\ &\mathsf{idpeel}(A, [x, y, z](P(x) \rightarrow P(y)),\\ &\qquad a, b, c,\\ &\qquad [x]\lambda(P(x), P(x), [w]w)),\\ &p)\end{aligned}$$

5.6 The set of lists

The set of lists $\mathsf{List}(A)$ is introduced in a similar way as the natural numbers, except that there is a parameter A that determines which set the elements of a list belongs to. There are two constructors to build a list, nil for the empty list and cons to add an element to a list. The constants we have introduced so far have the following types:

$$\mathsf{List} \in (A \in \mathsf{set})\,\mathsf{Set}$$

$$\begin{aligned}&\mathsf{nil} \in (A \in \mathsf{set})\,\mathsf{List}(A)\\ &\mathsf{cons} \in (A \in \mathsf{set}; a \in A; l \in \mathsf{List}(A))\,\mathsf{List}(A)\end{aligned}$$

The selector $\mathsf{listrec}$ for types is a constant that expresses primitive recursion for lists. The selector is introduced by the type declaration

$$\begin{aligned}\mathsf{listrec} \in \;&(A \in \mathsf{Set};\\ &\;C \in (\mathsf{List}(A))\,\mathsf{Set};\\ &\;c \in C(\mathsf{nil}(A));\\ &\;e \in (x \in A; y \in \mathsf{List}(A); z \in C(y))\,C(\mathsf{cons}(A, x, y));\\ &\;l \in \mathsf{List}(A))\\ &C(l)\end{aligned}$$

The defining equations for the $\mathsf{listrec}$ constant are

$$\mathsf{listrec}(A, C, c, e, \mathsf{nil}(A)) = c$$

$$\mathsf{listrec}(A, C, c, e, \mathsf{cons}(A, a, l)) = e(l, a, \mathsf{listrec}(A, C, c, e, l))$$

5.7 Disjoint union of two sets

If we have two sets A and B we can form the disjoint union $A+B$. The elements of this set are either of the form $\mathsf{inl}(A, B, a)$ or of the form $\mathsf{inr}(A, B, b)$ where $a \in A$ and $b \in B$. In order to express this in the framework we introduce the constants

$$+ \in (A, B \in \mathsf{Set})\,\mathsf{Set}$$

$$\mathsf{inl} \in (A, B \in \mathsf{Set}; A)\,A + B$$

$$\mathsf{inr} \in (A, B \in \mathsf{Set}; B)\,A + B$$

The selector when is introduced by the type declaration

$$\begin{array}{ll}\mathsf{when} \in & (A, B \in \mathsf{Set}; \\ & \;C \in (A+B)\,\mathsf{Set}; \\ & \;e \in (x \in A)\,C(\mathsf{inl}(A, B, x)); \\ & \;f \in (y \in B)\,C(\mathsf{inr}(A, B, y)); \\ & \;p \in A+B) \\ & C(p)\end{array}$$

and defined by the equations

$$\mathsf{when}(A, B, C, e, f, \mathsf{inl}(A, B, a)) = e(a)$$

$$\mathsf{when}(A, B, C, e, f, \mathsf{inr}(A, B, b)) = f(b)$$

Seen as a proposition the disjoint union of two sets expresses disjunction. The constructors correspond to the introduction rules

$$\frac{A \; true}{A \vee B \; true} \qquad \frac{B \; true}{A \vee B \; true}$$

and the selector when corresponds to the elimination rule.

$$\frac{A \vee B \; true \qquad C \; prop \qquad C \; true \; [A \; true] \qquad C \; true \; [B \; true]}{C \; true}$$

5.8 Disjoint union of a family of sets

In order to be able to deal with the existential quantifier and to have a set of ordinary pairs, we will introduce the disjoint union of a family of sets. The set is introduced by the type declaration

$$\Sigma \in (A \in \mathsf{Set}; B \in (A)\mathsf{Set})\,\mathsf{Set}$$

There is one constructor in this set, pair, which is introduced by the type declaration

$$\mathsf{pair} \in (A \in \mathsf{Set}; B \in (A)\mathsf{Set}; a \in A; B(a))\,\Sigma(A, B)$$

The selector of a set $\Sigma(A, B)$ splits a pair into its parts. It is defined by the type declaration

$$\begin{array}{ll}\mathsf{split} \in & (A \in \mathsf{Set}; B \in (A)\,\mathsf{Set}; \\ & \;C \in (\Sigma(A, B))\,\mathsf{Set}; \\ & \;d \in (a \in A; b \in B(a))\,C(\mathsf{pair}(A, B, a, b)); \\ & \;p \in \Sigma(A, B)) \\ & C(p)\end{array}$$

and the defining equation

$$\mathsf{split}(A, B, C, d, \mathsf{pair}(A, B, a, b)) = d(a, b)$$

Given the selector split it is easy to define the two projection functions that give the first and second component of a pair.

$$\begin{array}{l} \mathsf{fst} \in (A \in \mathsf{Set}; B \in (A)\,\mathsf{Set}; p \in \Sigma(A, B))\; A \\ \mathsf{fst}(A, B, p) = \mathsf{split}(A, B, [x]A, [x, y]x, p) \end{array}$$

$$\begin{array}{l} \mathsf{snd} \in (A \in \mathsf{Set}; B \in (A)\,\mathsf{Set}; p \in \Sigma(A, B))\; B(\mathsf{fst}(A, B, p)) \\ \mathsf{snd}(A, B, p) = \mathsf{split}(A, B, [x]B(\mathsf{fst}(A, B, p)), [x, y]y, p) \end{array}$$

When viewed as a proposition the disjoint union of a family of sets Σ(A,B) corresponds to the existential quantifier $(\exists\, x \in A)B(x)$. The types of constructor pair and when correspond to the natural deduction rules for the existential quantifier

$$\frac{a \in A \qquad B(a)\ \mathit{true}}{(\exists\, x \in A)\, B(x)\ \mathit{true}}$$

$$\frac{(\exists\, x \in A)\, B(x)\ \mathit{true} \qquad C\ \mathit{prop} \qquad C\ \mathit{true}\ [x \in A, B(x)\ \mathit{true}]}{C\ \mathit{true}}$$

5.9 The set of small sets

A set of small sets U, or a universe, is a set that reflects some part of the set structure on the object level. It is of course necessary to introduce this set if one wants to do some computation using sets, for example to specify and prove a type checking algorithm correct, but it is also necessary in order to prove inequalities such as $0 \neq \mathsf{succ}(0)$. Furthermore, the universe can be used for defining families of sets using recursion, for example non-empty lists and sets such as N^n.

We will introduce the universe simultanously with a function S that maps an element of U to the set the element encodes. The universe we will introduce has one constructor for each set we have defined. The constants for sets are introduced by the type declaration

$$\begin{array}{l} \mathsf{U} \in \mathsf{Set} \\ \mathsf{S} \in (\mathsf{U})\mathsf{Set} \end{array}$$

Then we introduce the constructors in U and the defining equations for S

$$\begin{array}{l} \mathsf{Bool}_U \in \mathsf{U} \\ \mathsf{S}(\mathsf{Bool}_U) = \mathsf{Bool} \end{array}$$

$$\begin{array}{l} \{\}_U \in \mathsf{U} \\ \mathsf{S}(\{\}_U) = \{\} \end{array}$$

$$\begin{array}{l} \mathsf{N}_U \in \mathsf{U} \\ \mathsf{S}(\mathsf{N}_U) = \mathsf{N} \end{array}$$

$$\begin{array}{l} \Pi_U \in (A \in \mathsf{U}; B \in (\mathsf{S}(A))\mathsf{U})\,\mathsf{U} \\ \mathsf{S}(\Pi_U(A, B)) = \Pi(\mathsf{S}(A), [h]\mathsf{S}(B(h))) \end{array}$$

$$\begin{array}{l} \mathsf{Id}_U \in (A \in \mathsf{U}; a \in \mathsf{S}(A); b \in \mathsf{S}(A))\,\mathsf{U} \\ \mathsf{S}(\mathsf{Id}_U(A, a, b)) = \mathsf{Id}(\mathsf{S}(A), a, b) \end{array}$$

$$\begin{array}{l} \mathsf{List}_U \in (A \in \mathsf{U})\,\mathsf{U} \\ \mathsf{S}(\mathsf{List}_U(A)) = \mathsf{List}(\mathsf{S}(A)) \end{array}$$

$$\begin{array}{l} +_U \in (A \in \mathsf{U}; B \in \mathsf{U})\,\mathsf{U} \\ \mathsf{S}(+_U(A, B)) = +(\mathsf{S}(A), \mathsf{S}(B)) \end{array}$$

$$\begin{array}{l} \Sigma_U \in (A \in \mathsf{U}; B \in (\mathsf{S}(A))\mathsf{U})\,\mathsf{U} \\ \mathsf{S}(\Sigma_U(A, B)) = \Sigma(\mathsf{S}(A), [h]\mathsf{S}(B(h))) \end{array}$$

Example 5.9.5. Let us see how we can derive an element in the set

$$\neg\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0))$$

or, in other words, how we can find an expression in the set

$$\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)) \rightarrow \{\}$$

We start by assuming that

$$x \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0))$$

Then we construct a function, Iszero, that maps a natural number to an element in the universe.

$$\begin{array}{l} \mathsf{Iszero} \in (\mathsf{N})\,\mathsf{U} \\ \mathsf{Iszero}(m) = \mathsf{natrec}(m, \mathsf{Bool}_U, [y, z]\{\}_U) \end{array}$$

It is easy to see that

$$\begin{array}{l} \mathsf{Iszero}(0) = \mathsf{S}(\mathsf{Bool}_U) = \mathsf{Bool} \\ \mathsf{Iszero}(\mathsf{succ}(0)) = \mathsf{S}(\{\}_U) = \{\} \end{array}$$

and therefore

$$\mathsf{true} \in \mathsf{Bool} = \mathsf{Iszero}(0)$$

$$\mathsf{subst}(x, \mathsf{true}) \in \mathsf{Iszero}(\mathsf{succ}(0)) = \{\}$$

Finally, we have the element we are looking for

$$\lambda(\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)), \{\}, [x]\mathsf{subst}(x, \mathsf{true})) \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)) \rightarrow \{\}$$

It is shown in Smith [37] that without a universe no negated equalities can be proved.

6 ALF, an interactive editor for type theory

At the department of Computing Science in Göteborg, we have developed an interactive editor for objects and types in type theory. The editor is based on direct manipulation, that is, the things which are being built are shown on the screen, and editing is done by pointing and clicking on the screen.

The proof object is used 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. The language of type theory is extended with place holders (written as indexed question marks). The notation $? \in A$ stands for the problem of finding an object in A. An object is edited by replacing the placeholders by expressions which may contain placeholders. It is also possible to delete a subpart of an object by replacing it with a placeholder.

There is a close connection between the individual steps in proving A and the steps to build a proof object of A. When we are making 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. If we have a problem

$$? \in A$$

then it is possible to refine the place holder in the following ways:

- 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 that we have a constant, we must have that $c(?_1, \ldots ?_n) \in A$, which holds if the type of the constant c is equal to $(x_1 \in A_1; \ldots; x_n \in A_n)B$ and $?_1 \in A_1, ?_2 \in A_2\,[x_1 \leftarrow ?_1], \ldots, x_n \in A_n[x_1 \leftarrow ?_1, \ldots, x_{n-1} \leftarrow ?_{n-1}]$ and

$$B[x_1 \leftarrow ?_1, \ldots, x_{n-1} \leftarrow ?_{n-1}] = A$$

So, we have reduced the problem A to the subproblems A_1, $A_2\,[x_1 \leftarrow ?_1]$, $\ldots$, $A_n[x_1 \leftarrow ?_1, \ldots, x_{n-1} \leftarrow ?_{n-1}]$ and further refinements must satisfy the constraint $B[x_1 \leftarrow ?_1, \ldots, x_{n-1} \leftarrow ?_{n-1}] = A$. The number n of new placeholders can be computed from the arity of the constant c and the expected arity of the placeholder. As an example, if we start with $? \in A$ and A is not a function type and 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, that is, the following equality must hold:

$$C\,[x \leftarrow ?_1] = A.$$

These kind of constraints will in general be simplified by the system. So, the editing step from $? \in A$ to $c(?_1) \in A$ is correct if $?_1 \in B$ and $C\,[x \leftarrow ?_1] = A$. This operation corresponds to applying a rule when we are constructing a proof. The rule c reduces the problem A to the problem B.

- 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 function type $(y \in B)C$. The type of the variable x must be B and we must keep track of the fact 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 $C\,[y \leftarrow x]$ under the assumption that $x \in B$. The assumed object x can be used to construct a solution to C, that is, we may use the knowledge that we have a solution to the problem B when we are constructing a solution to the problem C.

- 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 have been abstracted earlier.

To delete a part of a proof object corresponds to regretting some earlier steps in the proof. Notice that the deleted steps do not have to be the last steps in the derivation, by moving the pointer around in the proof object it is possible to undo any of the preceeding steps without altering the effect of following steps. However, the deletion of a sub-object is a non-trivial operation; it may cause the deletion of other parts which are depending on this.

The proof engine, which is the abstract machine representing an ongoing proof process (or an ongoing construction of a mathematical object) has two parts: the theory (which is a list of constant declarations) and the scratch area. Objects are built up in the scratch area and moved to the theory part when they are completed. There are two basic operations which are used to manipulate the scratch area. The *insertion* command replaces a placeholder by a new (possible incomplete) object and the *deletion* command replaces a sub-object by a placeholder.

When implementing type theory we have to decide what kind of inductive definitions and definitional equalities to allow. The situation is similar for both, we could give syntactic restrictions which guarantees that only meaningful definitions and equalities are allowed. We could for instance impose that an inductive definition has to be strictly positive and a definional equality has to be primitive recursive. We know, however, that any such restriction would disallow meaningful definitions. We have therefore – for the moment – no restrictions at all. This means that the correctness of a definition is the user's responsibility.

Among the examples developed in ALF, we can mention a proof that Ackermann's function is not primitive recursive [38], functional completeness of combinatorial logic [16], Tait's normalization proof for Gödel's T [17], the fundamental theorem of arithmetic [40], a constructive version of Ramsey's theorem [15], and a semantical analysis of simply typed lambda calculus with explicit substitution [7].

References

[1] 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.

[2] Roland Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In *Proceedings of the Workshop on General Logic, Edinburgh*. Laboratory for the Foundations of Computer Science, University of Edinburgh, February 1987.

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

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

[5] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

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

[7] Catarina Coquand. From Semantics to Rules: a Machine Assisted Analysis. In Börger, Gurevich, and Meinke, editors, *CSL'93*, pages 91–105. Springer-Verlag, LNCS 832, 1994.

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

[9] Thierry Coquand and Christine Paulin. Inductively defined types. In *Proceedings of the Workshop on Programming Logic, Båstad*, 1989.

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

[11] 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.

[12] 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.

[13] 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.

[14] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 1994. To appear.

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

[16] Veronica Gaspes. Formal Proofs of Combinatorial Completeness. In *To appear in the informal proceedings from the logical framework workshop at Båstad*, June 1992.

[17] Veronica Gaspes and Jan M. Smith. Machine Checked Normalization Proofs for Typed Combinator Calculi. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

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

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

[20] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *JACM*, 40(1):143–184, 1993.

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

[22] 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.

[23] 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.

[24] L. S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system*, volume 83 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1979.

[25] A. N. Kolmogorov. Zur Deutung der intuitionistischen Logik. *Matematische Zeitschrift*, 35:58–65, 1932.

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

[27] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, Nijmegen, 1994. Springer-Verlag.

[28] Per Martin-Löf. A Theory of Types. Technical Report 71–3, University of Stockholm, 1971.

[29] Per Martin-Löf. Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland Publishing Company, 1971.

[30] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

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

[32] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

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

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

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

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

[37] Jan M. Smith. The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes. *Journal of Symbolic Logic*, 53(3), 1988.

[38] Nora Szasz. A Machine Checked Proof that Ackermann's Function is not Primitive Recursive. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, June 1991. Also in G. Huet and G. Plotkin, editors, Logical Frameworks, Cambridge University Press.

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

[40] Björn von Sydow. A machine-assisted proof of the fundamental theorem of arithmetic. PMG Report 68, Chalmers University of Technology, June 1992.