Propositional Functions and Families of Types[*]

Jan M. Smith
Department of Computer Science
University of Göteborg/Chalmers
S-412 96 Göteborg
Sweden

March 1989

1 Introduction

In order to capture some of the programmers errors, several computer languages, like Pascal and ML, are equipped with a type system. Using the Curry-Howard interpretation of propositions as types [3, 8], or as we shall say here, propositions as sets, a type system can be made strong enough to be used to specify the task a program is supposed to do. This is one of the basis for Martin-Löf's suggestion in [11] to use his formulation of type theory for programming; his ideas are exploited in [14] and there are several computer implementations of type theory [4, 16]. Similar ideas are also behind Coquand and Huet's calculus of constructions [2].

The idea of propositions as sets is closely related to the intuitionistic explanations of the logical constants given by Heyting [7]. In Martin-Löf's type theory, the interpretation of propositions as sets is fundamental since the notions of proposition and set are identical. So a logical constant is definitionally equal to the corresponding set constant. Conversely, every set forming operation can be viewed as a logical constant, although some sets are more natural to think of as data types.

When using Martin-Löf's type theory for programming one often has to use strong principles, like a universe or well-orderings, when writing specifications or defining data types. For instance, a universe must be used when defining a proposition by induction on natural numbers or lists. There are disadvantages of using a universe and I will instead introduce an extension of type theory by which the use of a universe often can be avoided.

[*]To be published in *Notre Dame Journal of Formal Logic.*

The main reason why the rules given here have not been formulated before is that they require the distinction between sets and types. This is a basic idea of Martin-Löf's framework for type theory, which he first presented in a lecture in Göteborg in March 1986. The extension put forward here is that the elimination rules for the various set forming operations should be generalized so that the conclusion of such a rule is not restricted to be of the form "c is an element in the set C" but will be of the form "c is an object of the type γ". This means that it will be possible to define type valued functions by recursion on a set and, in particular, to define propositional functions by recursion without using a universe. It is then important that the elimination rules are formulated in the general way suggested by Schroeder-Heister [19, 20].

I will first briefly describe, following [15], how sets in Martin-Löf's type theory can be viewed as specifications and then why a universe sometimes must be used when expressing propositions. A presentation of the separation of sets and types will be given before the extension is formulated. Finally, I will give an interpretation of the extended type theory into type theory with one universe.

Acknowledgements. The convenience of a possibility to define propositions by induction on sets has been clear for a long time by people using type theory as a programming logic. For the extension proposed here, I am particularily indebted to Per Martin-Löf, Bengt Nordström and Kent Petersson for many discussions.

2 Specifications as sets

The idea of of viewing a specification of computer programs as a set in Martin-Löf's type theory has its origin both in understanding propositions as sets and in Kolmogorov's explanation in [9] of propositions as problems. Kolmogorov explains the sentential constants in the following way:

$A \wedge B$ is the problem of solving both of the problems A and the problem B.

$A \vee B$ is the problem of solving at least one of the problems A and B.

$A \supset B$ is the problem of solving the problem B provided that a solution to the problem A is given.

$\perp$ is a problem which has no solution.

Using the interpretation of propositions as sets and viewing a specification as a problem, which a program satisfying the specification solves, we can read these explanations as:

$A \times B$ is a specification of programs which, when executed, give a pair $\langle a, b\rangle$ where the program a satisfies the specification A and the program b satisfies the specification B.

$A + B$ is a specification of programs which, when executed, either give $\mathsf{inl}(a)$ where the program a satisfies the specification A or $\mathsf{inr}(b)$ where the program b satisfies the specification B.

$A \to B$ is a specification of programs which, when executed, give $\lambda x.b(x)$ where the program $b(a)$ satisfies the specification B if a is a program satisfying the specification A.

$\emptyset$ is a specification which is not satisfied by any program.

Type constructors corresponding to $\times$, $+$ and $\to$ occurs in many typed programming languages. However, in order to obtain a type system in which any interesting specifications can be expressed, we need cartesian products and disjoint unions on families of sets so that the quantifiers can be interpreted:

$(\Pi x \in A)B$ is a specification of programs which, when executed, give $\lambda x.b(x)$ where the program $b(a)$ satisfies the specification $B(a)$ if a is a program satisfying the specification A.

$(\Sigma x \in A)B$ is a specification of programs which, when executed, give a pair $\langle a, b\rangle$ where the program a satisfies the specification A and the program b satisfies the specification $B(a)$.

Beside these set forming operations corresponding to the the logical constants, we need a set $\mathsf{Id}(A, a, b)$ expressing that the elements a and b of the set A are identical. We also need a number of basic data types like the set N of natural numbers and the set $\mathsf{List}(A)$ of lists of elements in a set A. However, many specifications can still not be expressed with these sets but require a universe.

3 The need of a universe

Martin-Löf's first formulation of type theory [10] contained a universe V in which all sets were elements, including V itself. Such a universe would have been very practical to use but, by Girard's paradox, $\mathsf{V} \in \mathsf{V}$ implies that all sets are non-empty; hence, it is impossible to interpret propositions as sets. In Martin-Löf [12], the universe V is replaced by a series of universes U_0, U_1, ... where U_0 is the set of small sets and $\mathsf{U}_n \in \mathsf{U}_{n+1}$. Following the semantics in Martin-Löf [11], where a set is defined by prescribing how the canonical elements are formed, it is natural to view an element in a universe as a code for the corresponding set; this is the approach in [14, 21] and will be used here.

I will in this section give two examples how one is forced to use a universe in two basic applications; the first is when defining a simple proposition by induction and the second is when proving negated equalities.

If we, informally, want to define a predicate $member(a, l)$ which expresses that $a \in A$ is a member of the list $l \in \mathsf{List}(A)$ where A is a set, we can do that by structural induction on the list l:

$$\begin{aligned} member(a, \mathsf{nil}) &= \bot \\ member(a, b.s) &= (a =_A b) \vee member(a, s) \end{aligned}$$

where nil is the empty list and $b.s$ is the list obtained by adding the element b to the left of the list s. Structural induction on a list is in Martin-Löf's type theory expressed by the list-elimination rule

$$\frac{\begin{array}{l} l \in \mathsf{List}(A) \\ C(v)\ set\ [v \in \mathsf{List}(A)] \\ c \in C(\mathsf{nil}) \\ e(x, y, z) \in C(x.y)\ \ [x \in A,\ y \in \mathsf{List}(A),\ z \in C(y)] \end{array}}{\mathsf{listrec}(l, c, e) \in C(l)}$$

where $\mathsf{listrec}(l, c, e)$ is computed according to the rules

$$\begin{aligned} \mathsf{listrec}(\mathsf{nil}, c, e) &\rightarrow c \\ \mathsf{listrec}(a.s, c, e) &\rightarrow e(a, s, \mathsf{listrec}(s, d, e)) \end{aligned}$$

Using $\mathsf{listrec}$ and the interpretation of propositions as sets, we could write the definition above of $member$ as

$$member(a, l) \quad = \quad \mathsf{listrec}(l, \emptyset, (x, y, z)(\mathsf{Id}(A, a, x) + z))$$

However, member is a set valued function. So, in order to be able to apply list-elimination to show that $member(a, l)$ is a proposition, i.e. a set, we must have a family $C(v)$ of sets on $\mathsf{List}(A)$ so that $\emptyset \in C(\mathsf{nil})$ and $\mathsf{Id}(A, a, x) + z \in C(x.y)\ [x \in A,\ y \in \mathsf{List}(A),\ z \in C(y)]$. Hence, we must have a universe in which the sets we are using are elements.

To express $member$, we need the following codes, writing U for the first universe U_0,

$$\begin{gathered} \widehat{\bot} \in \mathsf{U} \\ A \widehat{+} B \in \mathsf{U}\ [A \in \mathsf{U},\ B \in \mathsf{U}] \\ \widehat{\mathsf{Id}}(A, a, b) \in \mathsf{U}\ [A \in \mathsf{U},\ a \in \mathsf{Set}(A),\ b \in \mathsf{Set}(A)] \end{gathered}$$

where Set is the decoding function for which we have the set-formation rule

$$\mathsf{Set}(A)\ set\ [A \in \mathsf{U}]$$

and the set-equalities

$$\begin{aligned}
\mathsf{Set}(\widehat{\emptyset}) &= \emptyset \\
\mathsf{Set}(A \mathbin{\widehat{+}} B) &= \mathsf{Set}(A) + \mathsf{Set}(B) \quad [A \in \mathsf{U},\ B \in \mathsf{U}] \\
\mathsf{Set}(\widehat{\mathsf{Id}}(A, a, b)) &= \mathsf{Id}(\mathsf{Set}(A), a, b) \quad [A \in \mathsf{U},\ a \in \mathsf{Set}(A),\ b \in \mathsf{Set}(A)]
\end{aligned}$$

Now we can define *member* in type theory by

$$\mathit{member}(a, l) \quad = \quad \mathsf{Set}(\mathsf{listrec}(l, \widehat{\emptyset}, (x, y, z)(\widehat{\mathsf{Id}}(A, a, x) \mathbin{\widehat{+}} z)))$$

and we can show, using list-elimination, that

$$\mathit{member}(a, l)\ \mathit{set}\ [a \in A,\ l \in \mathsf{List}(A)]$$

There are two disadvantages with this definition of *member*. First, the definition involves some coding compared with the informal definition of *member*. This is not so serious, since we could introduce some syntactical sugaring to avoid the coding. The second objection is more severe: the judgement $\mathit{member}(a, l)\ \mathit{set}\ [a \in A,\ l \in \mathsf{List}(A)]$ holds only when A is a small set, i.e. when $A \in \mathsf{U}$. So *member* is actually not defined for all sets A; in particular, we cannot use the above definition if U was used when defining the set A.

I will here just hint how a universe can be used to show that $\mathsf{0}$ is different from $\mathsf{1}$, for the details see [14]. By recursion on the natural numbers, we can define a function F such that

$$\begin{aligned}
F(\mathsf{0}) &= \emptyset \\
F(\mathsf{1}) &= \mathsf{T}
\end{aligned}$$

where T is the singleton set $\{\mathsf{tt}\}$. Since F is set valued, the formal definition of F in type theory requires a universe:

$$F(n) \quad = \quad \mathsf{Set}(\mathsf{natrec}(n, \widehat{\emptyset}, (x, y)\widehat{\mathsf{T}}))$$

where natrec is the recursion operator on the set of natural numbers. Assuming $\mathsf{Id}(\mathsf{N}, \mathsf{0}, \mathsf{1})$ it is easy, using $\mathsf{tt} \in F(1)$, to show that $F(0)$ is nonempty. Since $F(\mathsf{0}) = \emptyset$ we then obtain $\mathsf{Id}(\mathsf{N}, \mathsf{0}, \mathsf{1}) \to \emptyset$, i.e., by definition, $\neg\mathsf{Id}(\mathsf{N}, \mathsf{0}, \mathsf{1})$.

In Smith [22] it is shown that in type theory without a universe, no negated equalities at all can be proved.

4 The logical framework

The main reason to introduce a type level, more basic than the level of sets, is to have a framework in which sets can be introduced by simple declarations. This is important when building a computer system since you then do not want to make major changes of the implementation when introducing a new set forming operation. The Edinburgh LF [6] is based on similar ideas.

The type level introduced by Martin-Löf has judgements of the forms

α is a type,
α and β are equal types,
a is an object of the type α, and
a and b are equal objects of the type α,

which we formally write

$$\alpha{:}\mathit{type}$$
$$\alpha = \beta{:}\mathit{type}$$
$$a{:}\alpha$$
$$a = b{:}\alpha$$

respectively. In a series of lectures in Florence in spring 1987, Martin-Löf presented a detailed semantics of the type level in which judgemental equality is intensional. When building up Martin-Löf's set theory using the framework, we need function types, the type of sets and to each object in the type of sets, the type of elements of that set. The rules are formulated in a natural deduction style, but we will here not give the general rules concerned with substitution, equality and handling of contexts; the semantics and rules are given in detail in [17].

If we have a type α and a family β of types on α, then we can form the dependent function type from α to β:

Fun-formation

$$\frac{\alpha \,:\, \mathit{type} \qquad \beta \,:\, \mathit{type}\ [x \,:\, \alpha]}{(x : \alpha)\beta \,:\, \mathit{type}}$$

Functions are introduced by abstraction and we have the rules:

Abstraction

$$\frac{b \,:\, \beta\ [x \,:\, \alpha]}{(x)b \,:\, (x : \alpha)\beta}$$

Application

$$\frac{a \,:\, \alpha \qquad c \,:\, (x : \alpha)\beta}{c(a) \,:\, \beta(a/x)}$$

where $\beta(a/x)$ denotes the result of substituting a for the free variable x, assuming the usual restrictions on the free variables of a. We also have the following definitional equalities for objects in a function type:

β-conversion

$$\frac{a \,:\, \alpha \qquad b \,:\, \beta\ [x \,:\, \alpha]}{((x)b)(a) = b(a/x) \,:\, \beta(a/x)}$$

η-conversion

$$\frac{c \;:\; (x:\alpha)\beta}{(x)(c(x)) = c \;:\; (x:\alpha)\beta}$$

x must not occur free in c

ξ-conversion

$$\frac{b = d \;:\; \beta \;[x \;:\; \alpha]}{(x)b = (x)d \;:\; (x:\alpha)\beta}$$

We will use the abbreviation $(\alpha)\beta$ for $(x:\alpha)\beta$ when β does not depend on x and we will often display $f \;:\; (x_1:\alpha_1)\cdots(x_n:\alpha_n)\,\beta$ as

$$\begin{aligned} f \;:\; &(x_1 : \alpha_1)\\ &\quad\vdots\\ &(x_n : \alpha_n)\\ &\quad\beta \end{aligned}$$

The notation $f(x_1,\ldots,x_n)$ will be used for the repeated applications $f(x_1)\cdots(x_n)$ and, similarly, we will write $(x_1,\ldots,x_n)e$ for the repeated abstractions $(x_1)\cdots(x_n)e$.

That there is a type of sets is expressed by the rule

set-formation

$$\mathsf{set} \;:\; \mathit{type}$$

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

El-formation

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

The notation $a \in A$, which we used in the beginning of the paper, can now be seen as an abbreviation of $a \;:\; \mathit{El}(A)$.

We illustrate how sets can be introduced in the framework by declaring the constants for the set of natural numbers and cartesian products.

That N is a set is expressed by

$$\mathsf{N} \;:\; \mathsf{set}$$

The constructors for elements in N are then declared by

$$0 \;:\; \mathit{El}(\mathsf{N}) \qquad\qquad \mathsf{succ} \;:\; (\mathit{El}(\mathsf{N}))\mathit{El}(\mathsf{N})$$

When declaring the recursion operator natrec for natural numbers, we need dependent function types:

$$\begin{aligned}\mathsf{natrec}\ :\ & (C : (\mathit{El}(\mathsf{N}))\mathsf{set})\\ & (n : \mathit{El}(\mathsf{N}))\\ & (d : \mathit{El}(C(\mathsf{0})))\\ & (e : (x : \mathit{El}(\mathsf{N}))(y : (\mathit{El}(C(x)))\mathit{El}(C(\mathsf{succ}(x)))))\\ & \quad \mathit{El}(C(n))\end{aligned}$$

The computation rules for the recursion operator are expressed by the definitional equalities

$$\begin{aligned}&\mathsf{natrec}(C, 0, d, e) = d : \mathit{El}(C(\mathsf{0}))\\ &\qquad [C : (\mathit{El}(\mathsf{N}))\mathsf{set},\ d : \mathit{El}(C(\mathsf{0})),\ e : (x : \mathit{El}(\mathsf{N}))(\mathit{El}(C(x)))\mathit{El}(C(\mathsf{succ}(x)))]\end{aligned}$$

$$\begin{aligned}&\mathsf{natrec}(C, \mathsf{succ}(n), d, e) = e(n, \mathsf{natrec}(C, n, d, e)) : \mathit{El}(C(\mathsf{succ}(n)))\\ &\qquad [C : \mathit{El}(\mathsf{N})\mathsf{set},\ d : \mathit{El}(C(\mathsf{0})),\ e : (x : \mathit{El}(\mathsf{N}))(\mathit{El}(C(x)))\mathit{El}(C(\mathsf{succ}(x))),\ n : \mathit{El}(\mathsf{N})]\end{aligned}$$

Using the rules of the framework, we may derive the natural deduction rules for the natural numbers in Martin-Löf's set theory. For instance, from the declaration of natrec, we may obtain the elimination rule for natural numbers

$$\frac{\begin{array}{l} C(v) : \mathsf{set}\ [v : \mathit{El}(\mathsf{N})]\\ n : \mathit{El}(\mathsf{N})\\ d : \mathit{El}(C(\mathsf{0}))\\ e(x, y) : \mathit{El}(C(\mathsf{succ}(x)))\ [x : \mathit{El}(\mathsf{N}),\ y : \mathit{El}(C(x))]\end{array}}{\mathsf{natrec}(C, n, d, e) : \mathit{El}(C(n))}$$

Note that, in the conclusion of the rule, the expression $\mathsf{natrec}(C, n, d, e)$ contains the family $C(v)$ on N. This is a consequence of the explicit declaration of natrec in the framework, but C is also needed in the expression if we want mechanical type checking. So, when expressing set theory in the framework, we obtain a monomorphic theory. We may define a stripping function on the expressions which takes away the set information and we would then obtain expressions of the polymorphic theory in [11, 13]. However, the polymorphic theory is fundamentally different from the monomorphic theory; in Salvesen [18] it is shown that there are derivable judgements in the polymorphic theory which cannot come from any derivable judgement in the monomorphic theory by stripping.

The cartesian product on a family of sets is formed by the declaration

$$\Pi : (A : \mathsf{set})((B : \mathit{El}(A))\mathsf{set})\,\mathsf{set}$$

The elements in a cartesian product are obtained by λ-abstraction:

$$\begin{array}{lll} \lambda & : & (A : \mathsf{set}) \\ & & (B : (El(A))\mathsf{set}) \\ & & (b : (x : El(A))El(B(x))) \\ & & \qquad El(\Pi(A,B)) \end{array}$$

From the declaration of λ we get the introduction rule for the cartesian product:

$$\frac{\begin{array}{l} A : \mathsf{set} \\ B(x) : \mathsf{set}\ [x : El(A)] \\ b(x) : El(B(x))\ [x : El(A)] \end{array}}{\lambda(A,B,b) : El(\Pi(A,B))}$$

If we declare the constant apply by

$$\begin{array}{lll} \mathsf{apply} & : & (A : \mathsf{set}) \\ & & (B : (El(A))\mathsf{set}) \\ & & (El(\Pi(A,B))) \\ & & (u : El(A)) \\ & & \qquad El(B(u)) \end{array}$$

we obtain the elimination rule

$$\frac{\begin{array}{l} A : \mathsf{set} \\ B(x) : \mathsf{set}\ [x : El(A)] \\ c : El(\Pi(A,B)) \\ a : El(A) \end{array}}{\mathsf{apply}(A,B,c,a) : El(B(a))}$$

This rule corresponds $\forall$-elimination when interpreting propositions as sets. However, this elimination rule does not follow the pattern of the other elimination rules of Martin-Löf's set theory in that it does not express a recursion principle. In the preface of Martin-Löf [13], higher order assumptions were introduced by which it is possible to formulate recursion on a cartesian product. The selector apply is then replaced by $\mathsf{funsplit}$, which is declared by

$$\begin{aligned}\mathsf{funsplit}\ :\ & (A\!:\!\mathsf{set})\\ & (B\!:\!(\mathit{El}(A))\mathsf{set})\\ & (C\!:\!(\mathit{El}(\Pi(A,B)))\mathsf{set})\\ & (d : (y : (x\!:\!\mathit{El}(A))\mathit{El}(B(x)))\ \mathit{El}(C(\lambda(A,B,y))))\\ & (c\!:\!\mathit{El}(\Pi(A,B)))\\ & \qquad \mathit{El}(C(c))\end{aligned}$$

and associated with it is the definitional equality

$$\begin{aligned}&\mathsf{funsplit}(A,B,C,d,\lambda(A,B,b)) = d(b) : \mathit{El}(C(\lambda(A,B,b)))\\ &\quad [A : \mathsf{set},\ B : (\mathit{El}(A))\mathsf{set},\ C : (\mathit{El}((\Pi(A,B)))\mathsf{set},\ b : (x\!:\!\mathit{El}(A))\mathit{El}(B(x)),\\ &\qquad\qquad d : (y : (x\!:\!\mathit{El}(A))\mathit{El}(B(x)))\mathit{El}(C(\lambda(A,B,y)))]\end{aligned}$$

As in [13], apply can now be introduced by the explicit definition

$$\begin{aligned}&\mathsf{apply}(A,B,c,a) = \mathsf{funsplit}(A,B,(x)(B(a)),(y)(y(a)),c)\\ &\quad [A : \mathsf{set},\ B : (\mathit{El}(A))\mathsf{set},\ c : \mathit{El}(\Pi(A,B)),\ a : \mathit{El}(A)]\end{aligned}$$

thereby replacing the above declaration of apply. From the declaration of funsplit, we get the elimination rule for cartesian products

$$\frac{\begin{array}{l}A : \mathsf{set}\\ B(x) : \mathsf{set}\ [x : \mathit{El}(A)]\\ C(z) : \mathsf{set}\ [z\ \mathit{El}(\Pi(A,B))]\\ c : \mathit{El}(\Pi(A,B))\\ d(y) : \mathit{El}(C(\lambda(A,B,y)))\ [y : (x\!:\!\mathit{El}(A))\mathit{El}(B(x))]\end{array}}{\mathsf{funsplit}(A,B,C,d,c) : \mathit{El}(C(c))}$$

where the assumption $y : (x : \mathit{El}(A))\mathit{El}(B(x))$ corresponds to the higher order assumption $y(x) \in B(x)\ [x \in A]$. By the interpretation of propositions as sets, this rule corresponds to the generalized $\forall$-elimination in Schroeder-Heister [20].

5 The extension

In the elimination rule for natural numbers

$$\frac{\begin{array}{l}n : \mathit{El}(\mathsf{N})\\ C(v) : \mathsf{set}\ [v : \mathit{El}(\mathsf{N})]\\ d : \mathit{El}(C(0))\\ e(x,y) : \mathit{El}(C(\mathsf{succ}(x)))\ [x : \mathit{El}(\mathsf{N}),\ y : \mathit{El}(C(x))]\end{array}}{\mathsf{natrec}(C,n,d,e) : \mathit{El}(C(n))}$$

we have as one of the premisses that $C(v)$ is a family of sets on the set of natural numbers. In order to strengthen the rule so that we e.g. can define family of sets by recursion without using a universe, we generalize the rule to an arbitrary family γ of types on the natural numbers:

$$\frac{\begin{array}{l} n : \mathit{El}(\mathsf{N}) \\ \gamma : \mathit{type}\ [v : \mathit{El}(\mathsf{N})] \\ d : \gamma(\mathsf{0}/v) \\ e(x,y) : \gamma(\mathsf{succ}(x)/v)\ [x : \mathit{El}(\mathsf{N}),\ y : \gamma(x/v)] \end{array}}{\mathsf{Natrec}_\gamma(n,d,e) : \gamma(n/v)}$$

Formally, given $\gamma : \mathit{type}\ [v : \mathsf{N}]$, we introduce the constant Natrec_γ by the declaration

$$\begin{aligned} \mathsf{Natrec}_\gamma \ : \ & (n : \mathit{El}(\mathsf{N})) \\ & (\gamma(\mathsf{0}/v)) \\ & (e : (x : \mathit{El}(\mathsf{N}))\,(\gamma(x/v))\gamma(\mathsf{succ}(x)/v)) \\ & \quad \gamma(n/v) \end{aligned}$$

We also have to assert the definitional equalities

$$\begin{aligned} \mathsf{Natrec}_\gamma(0,d,e) = d : \gamma(\mathsf{0}/v) \\ [d : \gamma(\mathsf{0}/v),\ e : (x : \mathit{El}(\mathsf{N}))(\gamma(x/v)))\gamma(\mathsf{succ}(x)/v)] \end{aligned}$$

$$\begin{aligned} \mathsf{Natrec}_\gamma(\mathsf{succ}(n),d,e) = e(n, \mathsf{Natrec}_\gamma(n,d,e)) : \gamma(\mathsf{succ}(n)/v) \\ [d : \gamma(\mathsf{0}/v),\ e : (x : \mathit{El}(\mathsf{N}))(\gamma(x/v)))\gamma(\mathsf{succ}(x)/v),\ n : \mathit{El}(\mathsf{N})] \end{aligned}$$

Note that we cannot introduce a Natrec-operator uniformly over all families γ of types on the set of natural numbers but instead have to, given a family γ, introduce a new constant Natrec_γ. This is in contrast to the declaration of natrec, which is the same constant for all families $C(v)$ of sets on the natural numbers. If we want a uniform operator, we would have to extend the framework with yet another level where we would have *type* as object; such a level would correspond to the level of kinds in the Edinburgh LF [6].

To introduce a type valued recursion operator on a cartesian product $\Pi(A,B)$ where $A : \mathsf{set}$ and $B : (x : \mathit{El}(A))\,\mathsf{set}$, we must first have a family of types on the cartesian product. So let

$$\gamma : \mathit{type}\ [v : \mathit{El}(\Pi(A,B))]$$

be given. The constant $\mathsf{Funsplit}_\gamma$ is then introduced by the declaration

$$\begin{aligned} \mathsf{Funsplit}_\gamma \ : \ & (d : (y : (x : \mathit{El}(A))\mathit{El}(B(x)))\,\gamma(\lambda(A,B,y)/v)) \\ & (c : \mathit{El}(\Pi(A,B))) \\ & \quad \gamma(c/v) \end{aligned}$$

and we assert the definitional equality

$$\begin{aligned}\mathsf{Funsplit}_\gamma(d, \lambda(A,B,b)) \;=\; & d(b) : \gamma(\lambda(A,B,b)) \\ & [d : (y : (x : El(A))El(B(x)))\, \gamma(\lambda(A,B,y)/v)]\end{aligned}$$

Note that it is impossible to generalize apply in this way, since, in the declaration of apply, there is no family $C(v)$ of sets which we can replace by a family of types. This also holds for the selectors *fst* and *snd* for a cartesian product of two sets. Instead we have to use the selector split, by which we have the elimination rule

$$\frac{\begin{array}{l} A : \mathsf{set} \\ B : \mathsf{set} \\ C(v) : \mathsf{set}\ [v : El(A \times B)] \\ p : El(A \times B) \\ e(x,y) : El(C(\langle x, y\rangle))\ [x : El(A), y : El(B)] \end{array}}{\mathsf{split}(A,B,C,p,e) : El(C(p))}$$

This rule corresponds to the generalized elimination rule for conjunction in natural deduction, formulated in [19]:

$$\frac{A \wedge B \qquad C\ [A,\ B]}{C}$$

Given sets A and B and a family of types over a cartesian product of A and B

$$\gamma \;:\; type\ [v \;:\; El(A \times B)]$$

we declare the constant Split_γ by

$$\begin{aligned}\mathsf{Split}_\gamma \;:\; & (c : El(A \times B)) \\ & (d : (x : El(A))(y : El(B))\, \gamma(\langle x, y\rangle/v)) \\ & \quad \gamma(c/v)\end{aligned}$$

and assert the definitional equality

$$\begin{aligned}\mathsf{Split}_\gamma(\langle a, b\rangle, d) \;=\; & d(a,b) : \gamma(\langle a, b\rangle/v) \\ & [a : El(A),\ b : El(B),\ d : (x : El(A))(y : El(B))\, \gamma(\langle x, y\rangle/v)]\end{aligned}$$

In the same way as for N, $\Pi(A,B)$ and $A \times B$, it is now straightforward to introduce type valued recursion operators for the other sets.

6 Applications of the extension

We can now define *member* in type theory so that the definition really captures the informal definition we gave earlier. We first introduce a type valued recursion operator on lists. So, let a set A and a family

$$\gamma \;:\; \mathit{type}\,[v \;:\; \mathit{El}(\mathsf{List}(A))]$$

both be given. The constant $\mathsf{Listrec}_\gamma$ is declared by

$$\begin{aligned}\mathsf{Listrec}_\gamma \;:\;& (l : \mathit{El}(\mathsf{List}(A)))\\ & (\gamma(\mathsf{nil}/v))\\ & (e : (x : \mathit{El}(A))(y : \mathit{El}(\mathsf{List}(A)))(\gamma(y/v))\,\gamma(x.y/v))\\ & \quad\gamma(l/v)\end{aligned}$$

and we also assert the definitional equalities

$$\begin{aligned}\mathsf{Listrec}_\gamma(\mathsf{nil}, d, e) \;=\; d : \gamma(\mathsf{nil}/v)&\\ [d : \gamma(\mathsf{nil}/v),\; e : (x : \mathit{El}(A))(y : \mathit{El}(\mathsf{List}(A)))(\gamma(y/v))\,\gamma(x.y/v)]&\end{aligned}$$

$$\begin{aligned}\mathsf{Listrec}_\gamma(a.l, d, e) \;=\; e(a, l, \mathsf{Listrec}_\gamma(l, d, e)) : \gamma(a.l/v)&\\ [d : \gamma(\mathsf{nil}/v),\; e : (x : \mathit{El}(A))(y : \mathit{El}(\mathsf{List}(A)))(\gamma(y/v))\,\gamma(x.y/v),&\\ a : \mathit{El}(A),\; l : \mathsf{List}(A)]&\end{aligned}$$

To express *member*, the family γ in $\mathsf{Listrec}_\gamma$ is chosen to be the constant family set:

$$\begin{aligned}\mathsf{Listrec}_{\mathsf{set}} \;:\;& (\mathit{El}(\mathsf{List}(A)))\\ & (\mathsf{set})\\ & (e : (\mathit{El}(A))(\mathit{El}(\mathsf{List}(A)))(\mathsf{set})\,\mathsf{set})\\ & \quad\mathsf{set}\end{aligned}$$

We can now introduce *member* by the explicit definition

$$\begin{aligned}\mathit{member}(a, l) \;=\; \mathsf{Listrec}_{\mathsf{set}}(l, \emptyset, (x, y, z)(\mathsf{Id}(A, a, x) + z)) : \mathsf{set}&\\ [l : \mathit{El}(\mathsf{List}(A)),\; a : \mathit{El}(A)]&\end{aligned}$$

Negated equalities can now be derived without a universe. In the proof of $\neg\mathsf{Id}(\mathsf{N}, 0, 1)$, a function F satisfying

$$\begin{aligned}F(0) &= \emptyset\\ F(1) &= \mathsf{T}\end{aligned}$$

was used. F can now be defined by

$$F(n) \;=\; \mathsf{Natrec}_{\mathsf{set}}(n, \emptyset, (x, y)\mathsf{T})$$

and $\neg\mathsf{Id}(\mathsf{N}, 0, 1)$ can be proved.

The two examples above are quite obvious uses of type valued recursion. I have no such basic application of $\mathsf{Funsplit}_\gamma$, but here is a nice example, suggested by Bengt Nordström, of a simplification of the definition of application in a cartesian product on a family of sets. Given $A : \mathsf{set}$ and $B(x) : \mathsf{set}\ [x : A]$ we introduce $\mathsf{apply}_{\Pi(A,B)}$ by the definition

$$\mathsf{apply}_{\Pi(A,B)} = \mathsf{Funsplit}_{(x\,:\,El(A))El(B(x))}((y)y) \ : \ (x : El(A))El(B(x))$$

So $\mathsf{apply}_{\Pi(A,B)}$ is defined by just applying $\mathsf{Funsplit}_{(x\,:\,El(A))El(B(x))}$ on the identity function of $((x : El(A))El(B(x)))\,(x : El(A))El(B(x))$.

In Synek [23] type valued recursion is used when defining a set constructor for mutual recursive sets in terms of well-orderings. In this application, type valued recursion is crucial since otherwise, using a universe instead, the interpretation would only work for recursion involving small sets. A similar application of type valued recursion is also used in [14] when interpreting subsets in type theory.

7 Relation to universes

I will in this section sketch an interpretation of set theory with type valued recursion but without a universe into set theory with a universe but without type valued recursion. Aczel has shown in [1] (see also [5]) that the proof theoretic ordinal of Martin-Löf's type theory with a universe is, in Veblen's notation, $\phi_{\epsilon_0}(0)$. So we will then get an upper limit on the strength of set theory extended with type valued recursion. In particular, we will know that the extension is consistent.

The universe U is defined by an inductive definition so one can justify an elimination rule expressing a recursion principle on U; such an elimination rule is formulated in [14]. Since the concept of set is open, there is no corresponding induction principle for set and this is an important difference between type theory with a universe and type theory extended with type valued recursion. The formulation of type theory investigated in [1] does not include an elimination rule for U; if such a rule is added, one would expect a considerable increase of the proof theoretic strength.

The interpretation is defined in the following way. To each type α we associate a set α' and to each object a of type α an element a' in the set α'. The judgements

$$\begin{gathered} \alpha{:}\mathit{type} \\ \alpha = \beta{:}\mathit{type} \\ a{:}\alpha \\ a = b{:}\alpha \end{gathered}$$

are then interpreted by

$$\begin{gathered} \alpha'{:}\mathsf{set} \\ \alpha' = \beta'{:}\mathsf{set} \\ a'{:}\mathit{El}(\alpha') \\ a' = b'{:}\mathit{El}(\alpha') \end{gathered}$$

respectively. If a judgement depends on a context $x_1 : \alpha_1, \ldots,\ x_n : \alpha_n$, then the interpreted judgement will depend on the context $x_1 : \mathit{El}(\alpha_1'), \ldots,\ x_n : \mathit{El}(\alpha_n')$.

It is easy to see that a type must have the form

$$(x_1 : \alpha_1) \cdots (x_n : \alpha_n)\beta \qquad n = 0, 1, \ldots \ .$$

where β is either set or $\mathit{El}(A)$ for some $A : \mathsf{set}$. We define set' by

$$\mathsf{set}' \ \equiv \ \mathsf{U}$$

For $A : \mathsf{set}$, A' will be an element of U and $\mathit{El}(A)'$ is then defined to be $\mathsf{Set}(A')$.

A function type $(x_1 : \alpha_1) \cdots (x_n : \alpha_n)\beta$ is interpreted as the cartesian product

$$(\Pi x_1 : \alpha_1') \cdots (\Pi x_n : \alpha_n')\beta'$$

where $(\Pi x : A)B(x)$ is a sugared notation for $\Pi(A, B)$. An abstraction introduced by

$$\frac{b : \beta\ [x : \alpha]}{(x)b : (x : \alpha)\beta}$$

is interpreted by $\lambda(\alpha', \beta', (x)b')$ and an application introduced by

$$\frac{a : \alpha \qquad f : (x : \alpha)\beta}{f(a) : \beta(a/x)}$$

is interpreted by $\mathsf{apply}(\alpha', \beta', f', a')$.

Since we have η-conversion for objects of a function type but not for elements in a cartesian product, assumptions cannot be directly interpreted by a corresponding assumption; instead we must interpret an assumption

$$x : (x_1 : \alpha_1) \cdots (x_n : \alpha_n)\beta\ [x : (x_1 : \alpha_1) \cdots (x_n : \alpha_n)\beta]$$

by the derivable judgement

$$\lambda x_1 \ldots x_n \,.\, \mathsf{apply}(\ldots \mathsf{apply}(x, x_1) \ldots, x_n) \,:\, \\ \mathit{El}((\Pi x_1 : \alpha'_1) \cdots (\Pi x_n : \alpha'_n)\beta')\ [x \,:\, \mathit{El}((\Pi x_1 : \alpha'_1) \cdots (\Pi x_n : \alpha'_n)\beta')]$$

where stripping is used on λ and apply in order to avoid heavy notation. I will often use stripping in the sequel, but it will always be clear from the context how to decorate the terms with types.

Given these definitions, together with the interpretation below of the various constants, it is straightforward but tedious to prove, by induction on the length of the derivation, that if a judgement is derivable in type theory with type valued recursion then the interpretation of the judgement is derivable in type theory with a universe.

7.1 Interpretation of the set theoretic constants

The interpretation follows the same pattern for all the set theoretic constants. So we will only give the definitions for cartesian product and natural numbers, including type valued recursion on the natural numbers.

For each constant, we first give the interpretation of its type and then it is quite obvious how the interpretation of the constant must be defined. When a constant is declared to be an object in a function type, it is always the case that the interpretation is on λ form; hence they are η-convertible.

Interpretation of Π. The set theoretic constant Π is declared by

$$\Pi \,:\, (X : \mathsf{set})(Y : (\mathit{El}(X))\mathsf{set})\mathsf{set}$$

The code $\widehat{\Pi}$ for Π is declared by

$$\widehat{\Pi} \,:\, (X : \mathit{El}(\mathsf{U}))(Y : (\mathit{El}(\mathsf{Set}(X)))\mathit{El}(\mathsf{U}))\mathit{El}(\mathsf{U}) \tag{1}$$

The interpretation of the type of Π is, according to the definitions above,

$$(\Pi X : \mathsf{U})(\Pi Y : \mathsf{Set}(X) \to \mathsf{U})\mathsf{U}$$

where we have used the notation $\mathsf{Set}(X) \to \mathsf{U}$ for $\Pi(\mathsf{Set}(X), (x)\mathsf{U})$ since x does not occur in U. Using function application, we obtain from (1)

$$\widehat{\Pi}\,(X, Y) \,:\, \mathit{El}(\mathsf{U})\ [X \,:\, \mathit{El}(\mathsf{U}),\ Y \,:\, (\mathit{El}(\mathsf{Set}(X)))\mathit{El}(\mathsf{U})] \tag{2}$$

Since $\mathsf{apply}(Y, x) \,:\, \mathit{El}(\mathsf{U})\ [X \,:\, \mathit{El}(\mathsf{U}),\ x \,:\, \mathit{El}(\mathsf{Set}(X)),\ Y \,:\, \mathit{El}(\mathsf{Set}(X) \to \mathsf{U})]$, we get from (2)

$$\widehat{\Pi}\,(X, (x)\mathsf{apply}(Y, x)) \,:\, \mathit{El}(\mathsf{U})\ [X \,:\, \mathit{El}(\mathsf{U}),\ Y \,:\, \mathit{El}(\mathsf{Set}(X) \to \mathsf{U})]$$

which gives

$$\lambda XY\,.\,\widehat{\Pi}\,(X,(x)\mathsf{apply}(Y,x))\ :\ (\Pi X:\mathsf{U})(\Pi Y\ :\ \mathsf{Set}(X)\to\mathsf{U})\mathsf{U}$$

So we define Π' by

$$\Pi'\ \equiv\ \lambda XY\,.\,\widehat{\Pi}\,(X,(x)\mathsf{apply}(Y,x))$$

Interpretation of λ. The constant λ is introduced by

$$\lambda\ :\ (X:\mathsf{set})(Y:(\mathit{El}(X))\mathsf{set})(z:(x:\mathit{El}(X))Y(x))\mathit{El}(\Pi(X,Y)) \tag{3}$$

The type of λ is translated to

$$\begin{aligned}(\Pi X:\mathsf{U})(\Pi Y:\mathsf{Set}(X)\to\mathsf{U})(\Pi z:(\Pi x:\mathsf{Set}(X))\mathsf{apply}(Y,x))&\\ \mathsf{Set}(\widehat{\Pi}(X,(x)\mathsf{apply}(Y,x)))&\end{aligned}$$

From (3) we get

$$\lambda(X,Y,z)\ :\ \mathit{El}(\Pi(X,Y))$$

under the assumptions $X:\mathsf{set}$, $Y:(\mathit{El}(X))\mathsf{set}$ and $z:(x:\mathit{El}(X))\mathit{El}(Y(x))$. From the assumptions

$$X\ :\ \mathit{El}(\mathsf{U}),\qquad Y\ :\ \mathit{El}(\mathsf{Set}(x)\to\mathsf{U}),\qquad z\ :\ \mathit{El}(\Pi x:\mathsf{Set}(X))\mathsf{Set}(\mathsf{apply}(Y,x)))$$

we obtain

$$\mathsf{Set}(X)\ :\ \mathsf{set},\qquad (x)\mathsf{apply}(Y,x)\ :\ (\mathit{El}(\mathsf{Set}(X)))\mathsf{set}$$

and

$$(x)\mathsf{apply}(z,x)\ :\ (x:\mathit{El}(\mathsf{Set}(X)))\mathit{El}(\mathsf{Set}(\mathsf{apply}(Y,x)))$$

Hence, we define the interpretation of λ by

$$\lambda'\ \equiv\ \lambda XYz\,.\,\lambda(\mathit{El}(\mathsf{Set}(X)),(x)\mathsf{apply}(Y,x),(x)\mathsf{apply}(z,x))$$

Interpretation of apply. The constant apply is introduced by

$$\mathsf{apply}\ :\ (X:\mathsf{set})(Y:(\mathit{El}(X))\mathsf{set})(z:\mathit{El}(\Pi(X,Y)))(u:\mathit{El}(X))\mathit{El}(Y(u))$$

The type of apply is translated to

$$\begin{aligned}(\Pi X:\mathsf{U})(\Pi Y:\mathsf{Set}(X)\to\mathsf{U})(\Pi z:\mathsf{Set}(\widehat{\Pi}(X,(x)\mathsf{apply}(Y,x))))&\\ (\Pi u:\mathsf{Set}(X))(\mathsf{Set}(\mathsf{apply}(Y,u)))&\end{aligned}$$

From the assumptions

$$X : El(\mathsf{U}), \qquad Y : El(\mathsf{Set}(X) \to \mathsf{U})$$

we get

$$\mathsf{Set}(X) : \mathsf{set}, \qquad (x)\mathsf{Set}(\mathsf{apply}(Y, x)) : (El(\mathsf{Set}(X)))\mathsf{set}$$

We also make the assumptions

$$z : El(\mathsf{Set}(\widehat{\Pi}(X, (x)\mathsf{apply}(Y, x)))), \qquad u : El(\mathsf{Set}(X))$$

Since

$$\mathsf{Set}(\widehat{\Pi}(A, B)) = \Pi(\mathsf{Set}(A), (x)\mathsf{Set}(B(x)))$$

when $A : El(\mathsf{U})$ and $B(x) : El(\mathsf{U})\ [x : El(\mathsf{Set}(A))]$, we obtain

$$\mathsf{apply}(\mathsf{Set}(X), (x)\mathsf{Set}(\mathsf{apply}(Y, x)), z, u) : \mathsf{Set}(\mathsf{apply}(Y, x))$$

Hence, we define apply' by

$$\mathsf{apply}' \equiv \lambda XYzu\,.\,\mathsf{apply}(\mathsf{Set}(X), (x)\mathsf{Set}(\mathsf{apply}(Y, x), z, u))$$

Interpretation of N, $\mathsf{0}$ and succ. The constant N is declared by $\mathsf{N} : \mathsf{set}$. So, N' can simply be defined to be the code for N:

$$\mathsf{N}' \equiv \widehat{\mathsf{N}}$$

Since $\mathsf{0} : El(\mathsf{N})$ we just interpret $\mathsf{0}$ by

$$\mathsf{0}' \equiv \mathsf{0}$$

The constant succ is introduced by the declaration $\mathsf{succ} : (El(\mathsf{N}))El(\mathsf{N})$ whose type is interpreted by $\mathsf{Set}(\widehat{\mathsf{N}}) \to \mathsf{Set}(\widehat{\mathsf{N}})$ which is equal to $\mathsf{N} \to \mathsf{N}$. Hence, we define succ' by

$$\mathsf{succ}' \equiv \lambda x.\mathsf{succ}(x)$$

The constant natrec interpreted in a similar way.

Interpretation of Natrec_γ. Let γ be a family of types on the set of natural numbers:

$$\gamma : type\ [v : El(\mathsf{N})]$$

The recursion operator Natrec_γ is introduced by the declaration

$$\mathsf{Natrec}_\gamma : (n : El(\mathsf{N}))(D : \gamma(\mathsf{0}/v))(E : (x : El(N))(Y : \gamma(x/v))\gamma(\mathsf{succ}(x)/v)\gamma(n)$$

The interpretation of the type of Natrec_γ is

$$(\Pi n : \mathsf{N})(\Pi D : \gamma'(\mathsf{0}/v))(\Pi E : (\Pi x : \mathsf{N})(Y : \gamma'(x))\gamma'(\mathsf{succ}(x)/v)\gamma'(n/v)$$

In a similar way as for the constants for the cartesian product, we see that Natrec'_γ must be defined by

$$\mathsf{Natrec}'_\gamma \;\equiv\; \lambda nDE\,.\,\mathsf{natrec}((v)\gamma', n, D, (x, Y)\mathsf{apply}(\mathsf{apply}(E, x), Y))$$

References

[1] Peter Aczel. The strength of Martin-Löf's type theory with one universe. In *Proceedings of the symposium on mathematical logic (Oulu, 1974)*, pages 1–32, Department of Philosophy, University of Helsinki, 1977.

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

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

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

[5] S. Feferman. Iterated Inductive, Fixed Point Theories. In *Patras Logic Symposion*, North-Holland, 1982.

[6] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. In *Proceedings of the Symposium on Logic in Computer Science*, pages 194 – 204, Ithaca, New York, June 1987.

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

[8] 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.

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

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

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

[12] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118, North-Holland, Amsterdam, 1975.

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

[14] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* To be published by Oxford University Press, 1989.

[15] Bengt Nordström and Jan Smith. Propositions, Types and Specifications in Martin-Löf's Type Theory. BIT, 24(3):288–301, October 1984.

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

[17] Kent Petersson and Jan M. Smith. A chapter on Martin-Löf's type theory in vol III of Handbook of Logic in Computer Science, to be published by Oxford University Press. In preparation.

[18] Anne Salvesen. *Polymorphism and Monomorphism in Martin-Löf's Type Theory.* Technical Report, Norwegian Computing Center, P.b. 114, Blindern, 0316 Oslo 3, Norway, December 1988.

[19] Peter Schroeder-Heister. A Natural Extension of Natural Deduction. *Journal of Symbolic Logic*, 49(4), 1984.

[20] Peter Schroeder-Heister. Generalized Rules for Operators and the Completeness of the Intuitionistic Operators $\&$, $\vee$, $\supset$, $\bot$, $\forall$, $\exists$. Lecture Notes in Mathematics, Vol 1104. In Richter et al, editor, *Computation and Proof Theory*, Springer-Verlag, 1984.

[21] Jan M. Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.

[22] 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.

[23] Dan Synek. *Deriving Rules for Inductive Sets in Martin-Löf's Type Theory.* Technical Report, Programming Methodology Group, Dept. of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, February 1989.