On the Meaning and Construction of the Rules in Martin-Löf's Theory of Types

Roland Backhouse
Department of Mathematics and Computing Science
University of Groningen
PO Box 800
9700 AV GRONINGEN
The Netherlands

Abstract We describe a method to construct the elimination and computation rules from the formation and introduction rules for a type in Martin-Löf's theory of types. The construction is based on an understanding of the inference rules in the theory as judgements in a pre-theory. The motivation for the construction is to permit disciplined extensions to the theory as well as to have a deeper understanding of its structure.

0 Introduction

Martin-Löf's theory of types [ML0] has attracted considerable attention from both logicians and computing scientists, and for a variety of reasons. First, it has considerably enhanced our understanding of constructive proof and the relationship between such proofs and programs. Second, it anticipated the notion of dependent type introduced for example in the language Pebble [BL]. Third, as a formal system it has an elegant structure that is worthy of study in its own right. This paper is largely concerned with the latter aspect, the motivation being that by gaining a deeper understanding of its structure we will be better equipped to adapt the theory to individual needs

The present work grew out of a feeling of discontent with the theory. On first encounter the universal reaction among computing scientists appears to be that the theory is formidable. Indeed, several have specifically referred to the overwhelming number of rules in the theory. On closer examination, however, the theory betrays a rich structure — a structure that is much deeper than the superficial observation that types are defined by introduction, elimination and computation rules. Once recognised this structure considerably reduces the burden of understanding. And yet, to my knowledge, the structure of the theory has not been properly discussed or documented; Martin-Löf, himself, alludes to the fact that there is a "pattern... in the type forming operations" in the preface to the notes prepared by Giovanni Sambin [ML1], but he does not give a detailed account of the pattern.

So much for the ideological motivations for this paper. At a more practical level it has become increasingly clear to us that there is a need to freely permit *disciplined* extensions to the theory. That the theory is open to extension is a fact that was clearly intended by Martin-Löf. Indeed, it is a fact that has been exploited by several individuals; Nordström, Petersson and Smith [NPS] have extended the theory to include lists, they and Constable et al [Co] have added subset types and Constable et al have introduced quotient types, Nordström has introduced multi-level functions [No], Chisholm has introduced a very special-purpose type of tree structure [Ch] and Dyckhoff [Dy] has defined the type of categories.

Initially we were against such extensions on the grounds that it is often possible to define them in terms of the W-type (for examples see [Kh]), because they add to the complexity of the theory and because they might undermine the quality of the theory even to the extent of introducing inconsistencies. The experiences and arguments of others have now convinced us that this view is wrong. The view that we now hold is that implementations of type theory (proof checkers, proof editors etc. like Nuprl and the Gothenburg Type Theory System) should permit user-defined extensions to the theory but in a disciplined way. This paper is therefore a first attempt at formulating such a discipline.

The main contribution that we make here is to describe a scheme for computing the elimination rule and computation rules for a newly introduced type. In other words, we show that it suffices to provide the type formation rule and the introduction rules for a new type; together these provide sufficient information from which the remaining details can be deduced. (At this stage in our work we cannot provide such a scheme to cover all type constructors; the limitations of our work are discussed in the conclusion.) The significance of this result is that it has the twin benefits of reducing the burden of understanding and

the burden of definition. It reduces the burden of understanding since we now need to understand only the formation and introduction rules and the general scheme for inferring the remaining rules. Conversely, the burden of definition is reduced since it suffices to state the formation and introduction rules, the others being inferred automatically.

A necessary preliminary was to give an explanation of the meaning of the formal rules in the theory. Such an explanation is notably absent from the seminal account of Martin-Löf's theory [ML0]; although the paper gives a very careful account of the meaning of the various judgement forms, nowhere is it stated how to interpret the rules. Yet, it is fundamental that a type be defined by its rules and that the rules be meaningful in some precise sense. We therefore begin this paper by providing an account, in section 2, of the rules in type theory as judgements in a "pre-theory", that is, a theory that precedes the theory of types itself. Also in section 2 we introduce the notion of internal consistency of a rule. The pre-theory is taken from [NPS], with which we assume some familiarity, and is summarised in section 1.

The main body of the paper is contained in section 3. Here we detail the scheme for computing elimination and computation rules. Several examples of the scheme are also included in this section.

There are many shortcomings in this stage of our work. Some of those of which I am aware are discussed in the conclusions. Needless to say I would be grateful for further criticism and comments.

1 The Pre-Theory

The pre-theory that we need involves an understanding of the theory of expressions and the notion of a category as discussed by Nordström, Petersson and Smith [NPS], and to which we refer the reader for complete details.

The theory of expressions defines the arity of expressions and definitional equality of expressions. For understanding the rules that follow it is necessary to know that different occurences of the same variable in a rule denote definitionally equal expressions. Identical expressions are, of course, definitionally equal but also $((x)P)(x)$ is definitionally equal to P for any expression P and variable x, and $((x)c)(y)$ is definitionally equal to c for any constant c and variables x and y. In particular $((x)\mathit{Type})(y)$ is definitionally equal to *Type*, since *Type* is a constant.

The rules of the pre-theory (and of the theory) prescribe the formation of derivations and from derivations one may abstract judgements. The syntactic form of derivations and judgements is described in essence by the following BNF syntax

$$
\begin{aligned}
\langle \mathit{derivation} \rangle &::= \langle \mathit{statement} \rangle * \\
\langle \mathit{statement} \rangle &::= \langle \mathit{primitive\ statement} \rangle \mid \langle \mathit{context} \rangle \\
\langle \mathit{context} \rangle &::= \text{“}|[\text{”}\langle \mathit{assumption} \rangle \text{“}\triangleright\text{”}\langle \mathit{derivation} \rangle \text{“}]|\text{”} \\
\langle \mathit{assumption} \rangle &::= \langle \mathit{statement} \rangle \\
\langle \mathit{judgement} \rangle &::= \langle \mathit{primitive\ statement} \rangle \mid \text{“}|[\text{”}\langle \mathit{assumption} \rangle \text{“}\triangleright\text{”}\langle \mathit{judgement} \rangle \text{“}]|\text{”}
\end{aligned}
$$

A derivation is thus a sequence of statements each of which is either a primitive statement or a context. Contexts are delimited by the scope brackets “|[” and “]|” and consist of an assumption followed by a (sub-) derivation. A judgement is formed from a derivation by the simple process of eliding all but the last statement in the derivation and in all its sub-derivations. For example consider a derivation of the form

$$
\begin{array}{l}
p_0 \\
|[\ a_0 \\
\triangleright\ p_1 \\
\quad |[\ a_1 \\
\quad \triangleright\ p_2 \\
\quad]| \\
]|
\end{array}
$$

where p_0,p_1 and p_2 are primitive statements and a_0,a_1 are assumptions. Then the judgement obtained by eliding all but the last statement in each derivation is the following.

$$
\begin{array}{l}
|[\ a_0 \\
\triangleright\ |[\ a_1 \\
\quad \triangleright\ p_2 \\
\quad]| \\
]|
\end{array}
$$

which may be read as "*assuming* a_0 *and assuming* a_1 *then* p_2".

We say that a statement p *precedes* a statement q within a derivation if p is the ith statement of the derivation, for some i, and either (a) q is the jth statement of the derivation for some $j > i$ or (b) the jth statement, for some $j > i$, is a context that includes the statement q. The statement p also precedes the statement q in a derivation if p precedes q in a subderivation of the derivation. Thus in the example above statement p_0 precedes statements a_0,p_1,a_1 and p_2. Also p_1 precedes $[\![\ a_1\ \triangleright\ p_2\ ]\!]$ and p_2, and so on.

Each rule in the pre-theory (and in the theory) consists of a set of premises and a conclusion, in the usual way. The application of a rule permits a derivation to be *extended* by adding a statement to the end of the derivation or to the end of a subderivation provided that the extended derivation includes statements preceding the added statement that match the premises in the same way that the added statement matches the conclusion. An axiom is a rule that has no premises; thus application of an axiom permits a derivation to be extended at an arbitrary point.

Note that there is considerable freedom in the order of construction of statements in a derivation. The form in which derivations are presented on the printed page will suggest one particular order but it should not be supposed that this is the only order.

Just those rules that we explicitly employ are given below. For these rules we explain their meaning in an ad hoc way. We do not, however, attempt to give any meaning to the word *category* : the reader must accept that certain expressions denote "categories", which expressions being determined by application of the rules. Thus the first rule must be accepted as an axiom - "*Type*" denotes a category.

$$\frac{}{Type\ cat}$$ Type Formation

"$Type\ cat$" is a primitive statement and therefore a derivation and a judgement.

Contexts may be introduced into a derivation via the assumption rule.

$$\frac{C\ cat}{\begin{array}{l}[\![\ x : C\\ \triangleright\\ ]\!]\end{array}}$$ Assumption

If in a derivation we have a primitive statement of the form $C\ cat$ then it is possible to extend the derivation by adding an assumption of the form $x : C$ where x is a variable. Note that the assumption is a particular sort of primitive statement. For clarity it is separated from following statements by the symbol "$\triangleright$".

For each type A the elements of A form a category. Thus we have the rule of element formation.

$$\frac{A : Type}{El(A)\ cat}$$ Element formation

The rule permits a derivation that includes a statement of the form $A : Type$ to be extended by adding the statement $El(A)\ cat$ to the derivation. In so doing the context of both statements must be identical.

Function categories are obtained by discharging assumptions.

$$\begin{array}{l} A\ cat \\ \|[\ x :\ A \\ \triangleright\ B(x)\ cat \\ ]\| \\ \hline F(A,B)\ cat \end{array} \qquad \text{Function formation}$$

$F(A,B)$ is the category of functions that map an object x of the category A into an object of the category $B(x)$. Note that $B(x)$ does not denote an expression containing free occurrences of x, as it would in conventional mathematics, but an expression that is definitionally equal to the application of some expression B of arity $\mathbf{0} \rightarrow \mathbf{0}$ to some variable x. For instance $Type$ takes the form $B(x)$ since it is definitionally equal to $((y)Type)(x)$.

The final rule we need in the pre-theory is the rule of function elimination.

$$\begin{array}{l} a : A \\ c :\ F(A,B) \\ \hline c(a) :\ B(a) \end{array} \qquad \text{Function Elimination}$$

An example of a derivation using these rules is as follows. Note that the line numbers and material within percent signs are not part of the derivation but are only included as aids to the reader. Also, the symbol "$\equiv$" has been used to denote definitional equality.

```
                % Type formation %
0       Type cat
                % 0, assumption %
1.0     |[      X : Type
        ▷           % 1.0, El-formation %
1.1             El(X) cat
                    % 1.1, assumption %
1.2.0           |[      x : El(X)
                ▷           % Type formation %
1.2.1               Type cat
                ]|
                    % 1.1, 1.2, ((x)Type)(x) ≡ Type, fun-formation %
1.3             F(El(X),(x)Type) cat
                    % 1.1, assumption %
1.4.0           |[      y : El(X)
                ▷           % 1.3, assumption %
1.4.1.0             |[      Y : F(El(X),(x)Type)
                    ▷           % 1.4.0, 1.4.1.0, ((X)Type)(y) ≡ Type, fun-elim %
1.4.1.1                 Y(y) : Type
                    ]|
                ]|
        ]|
```

The judgement obtained from this derivation by eliding all but the last statement in every subderivation is the following.

$$
\begin{array}{l}
\|[\ X : Type \\
\triangleright\ \|[\ y : El(X) \\
\quad \triangleright\ \|[\ Y : F(El(X), (x)Type) \\
\quad\quad \triangleright\ Y(y) : Type \\
\quad\quad]\| \\
\quad]\| \\
]\|
\end{array}
$$

In words, assuming X is a type, y is an element of X and Y is a function mapping elements of X into the category of types, then Y applied to y is a type.

2. The Rules of Type Theory

Now that we have discussed the pre-theory we may proceed to explicate the meaning of the rules in type theory itself. We do this by interpreting each rule of type theory as a judgement in the pre-theory. The premises of the rule become assumptions of the pre-theory judgement.

This rather simple idea has far-reaching consequences. It means that we can decide whether the premises of a type-theory rule make sense by constructing a derivation in the pre-theory. We can also check that the conclusion of the rule obeys a certain consistency requirement (called *internal consistency* in the sequel).

Some preliminary examples may help to convey the idea. Let us consider the formation, introduction and elimination rules for the disjoint-sum type.

Below we show the formation rule and the corresponding pre-theory judgement. Here the correspondence is immediate: premises become assumptions and P *type* is replaced by P: *Type*.

$$\begin{array}{l} A\ type \\ B\ type \\ \hline A \vee B\ type \end{array} \qquad \begin{array}{l} [\![\ A : Type \\ \triangleright\ [\![\ B : Type \\ \quad \triangleright\ A \vee B : Type \\ \quad]\!] \\ ]\!] \end{array}$$

type-theory rule — pre-theory judgement

$\vee$-formation

Next consider one of the introduction rules for the disjoint-sum type. Again we exhibit the type-theory rule and the corresponding pre-theory judgement.

$$\begin{array}{l} A\ type \\ B\ type \\ x \in A \\ \hline i(x) \in A \vee B \end{array} \qquad \begin{array}{l} [\![\ A : Type \\ \triangleright\ [\![\ B : Type \\ \quad \triangleright\ [\![\ x : El(A) \\ \quad\quad \triangleright\ i(x) : El(A \vee B) \\ \quad\quad]\!] \\ \quad]\!] \\ ]\!] \end{array}$$

type-theory rule — pre-theory judgement

$\vee$-introduction

This example is more illuminating because we can use it to give a preliminary account of what it means for an introduction rule to be internally consistent. Specifically, given an introduction rule with conclusion $e \in E$ we convert the rule into a judgement $El(E)$ *cat* under assumptions derived from (in a manner yet to be described) the premises of the rule. The rule is then said to be internally consistent if the judgement can be verified using the

rules of the pre-theory and the formation rules of the type. Thus for our example we verify internal consistency by establishing the judgement

$$
\begin{array}{l}
\|[\ A : Type \\
\quad \triangleright\ \|[\ B : Type \\
\qquad \triangleright\ \|[\ x : El(A) \\
\qquad\quad \triangleright\ El(A \vee B)\ cat \\
\qquad\quad]\| \\
\qquad]\| \\
\quad]\|
\end{array}
$$

This judgement has the following derivation.

```
                % Type formation %
0           Type cat
                % 0,assumption %
1.0         |[   A : Type
            ▷       % 0, assumption %
1.1.0           |[   B : Type
                ▷       % 1.0, element formation %
1.1.1               El(A) cat
                        % 1.1.1,assumption %
1.1.1.0             |[   x : El(A)
                    ▷       % 1.0,1.1.0,∨-formation %
1.1.1.1                 A ∨ B : Type
                            % 1.1.2.1,element formation %
1.1.1.1                 El(A ∨ B) cat
                    ]|
                ]|
            ]|
```

Note that the penultimate step makes use of the $\vee$-formation rule.

Finally consider the elimination rule for disjoint sum. The type-theory rule and corresponding judgement are shown below.

$A\ type$
$B\ type$
$[\![\ x \in A \vee B$
$\triangleright\ C(x)\ type$
$]\!]$
$[\![\ y \in A$
$\triangleright\ d(y) \in C(i(y))$
$]\!]$
$[\![\ y \in B$
$\triangleright\ e(y) \in C(j(y))$
$]\!]$
$f \in A \vee B$

$\omega(f,d,e) \in C(f)$

type-theory rule

$[\![\ A : Type$
$\triangleright\ [\![\ B : Type$
$\triangleright\ [\![\ C : F(El(A \vee B),(x)Type)$
$\triangleright\ [\![\ d : F(El(A),(y)El(C(i(y))))$
$\triangleright\ [\![\ e : F(El(B),(y)El(C(j(y))))$
$\triangleright\ [\![\ f : El(A \vee B)$
$\triangleright\ \omega(f,d,e) : El(C(f))$
$]\!]$
$]\!]$
$]\!]$
$]\!]$
$]\!]$
$]\!]$

pre-theory judgement

$\vee$-elimination

The additional complexity of this example arises from the hypothetical premises (that is, premises involving assumptions). The specific translation process used converts a premise of the form $[\![\ x \in A \triangleright\ J\ ]\!]$ as follows. First convert the judgement J to, say, $b(x)$: $B(x)$. Then construct the judgement $b : F(El(A), B)$. Thus the premise $[\![\ x \in A \vee B \triangleright\ C(x)\ type\ ]\!]$ is converted by first converting $C(x)\ type$ to $C(x) : Type$, which is definitionally equal to $C(x) : ((x)Type)(x)$. Then the judgement $C : F(El(A \vee B), (x)Type)$ is constructed.

As an exercise the reader may wish to verify the internal consistency of the rule by constructing a derivation of the following judgement.

$[\![\ A : Type$
$\triangleright\ [\![\ B : Type$
$\triangleright\ [\![\ C : F(El(A \vee B),(x)Type)$
$\triangleright\ [\![\ d : F(El(A),(y)El(C(i(y))))$
$\triangleright\ [\![\ e : F(El(B),(y)El(C(j(y))))$
$\triangleright\ [\![\ f : El(A \vee B)$
$\triangleright\ El(C(f))\ cat$
$]\!]$
$]\!]$
$]\!]$
$]\!]$
$]\!]$
$]\!]$

2.1 Formalising the Conversion

Converting from type theory rules to pre-theory judgements is a purely syntactic process which we now summarise.

Each type theory rule consists of an ordered sequence of premises and a single inference. The pre-theory judgement consists of an ordered sequence of assumptions (each corresponding to a premise) and a single conclusion. Individual statements (premise or inference) of the rule are converted according to the following algorithm.

(a) "E *type*" is converted to "$E : Type$"
(b) "$e \in E$" is converted to "$e : El(E)$"
(c) given a hypothetical premise of the form $|[\ H\ \triangleright\ S\ ]|$ first convert H to the form "$x : E$" and S to the form "$d(x) : D$". Then the premise is converted to "$d : F(E,(x)D)$". Note that definitional equality may be required to complete the conversion of H and/or S to the required form. Note also that the construction permits the hypothesis H to be itself hypothetical.

Consider now the presentation of a new type in the theory. The order of presentation of the rules is (1) formation rules (2) introduction rules (3) elimination rule (4) computation rules. (We do not consider equality rules in this paper although they do not pose additional difficulties.) Suppose the inference of one of these rules is converted to a statement of the form "$e : E$" and the premises are converted to statements of the form "$a_1 : A_1$",...,"$a_n : A_n$". Then we say that the rule is *internally consistent* if and only if there is a derivation of the judgement

$$|[\ a_1 : A_1 \triangleright\ \ldots\ \triangleright\ |[\ a_n : A_n \triangleright\ E\ cat\ \ ]|\ \ldots\ \ ]|$$

in a system consisting of the pre-theory and those rules governing the type that precede the given rule in the above order.

3 Introducing New Types into the Theory

The mechanism for introducing a new type into the theory has three stages. First the formation rule for the type is prescribed followed by the introduction rules. Finally the elimination rules and computation rule are automatically inferred from the formation rule and the introduction rules (provided, of course, that the latter are internally consistent). This is the subject of the next three sections.

3.1 Formation Rules

Each formation rule introduces some new type constructor, Θ say. The premises of such a rule all take the form "*A type*", for some expression A and within some context. The corresponding pre-theory judgement therefore has the form

$$
\begin{array}{l}
|[\ A_1 : T_1 \\
\triangleright\ |[\ A_2 : T_2 \\
\quad \triangleright\ |[\ \cdots \\
\quad\quad \triangleright\ |[\ A_n : T_n \\
\quad\quad\quad \triangleright\ \Theta(A_1, \ldots, A_n) : \mathit{Type} \\
\quad\quad\quad]| \\
\quad\quad]| \\
\quad]| \\
]|
\end{array}
$$

where the expressions $T_1,\ldots,T_n$ are all elements of a class of expressions $TYPE$ where $TYPE$ has the following syntax

$$TYPE ::= \text{“}Type\text{”} \ \mid\ \text{“}F(\text{”}\ expression\ \text{“,”}\ TYPE\ \text{“)”}$$

For future reference we refer to $A_1,\ldots,A_n$ as the *formation variables.*

3.2 Introduction Rules

A type Θ may have several introduction rules, each one of which introduces a new canonical-object constructor. Consider one such constructor, θ say. Then the premises of the θ-introduction rule are in two parts. First there are the premises of the Θ-formation rule. (These are rarely stated explicitly.) Second there is a number, m say, of premises each of the form "$b \in B$" for some expressions b and B and within some context. Thus the corresponding pre-theory judgement takes the form

$$
\begin{array}{l}
[\![\ A_1 : T_1; \ldots; A_n : T_n \\
\triangleright\ [\![\ x_1 : S_1; \ldots; x_m : S_m \\
\quad \triangleright\ \theta(x_1, \ldots, x_m) : El(\Theta(A_1, \ldots, A_n)) \\
\quad]\!] \\
]\!]
\end{array}
$$

The first set of premises corresponds to the premises of the Θ-formation rule. The expressions $S_1,\ldots,S_m$ in the second set of premises all belong to the syntactic category *ELEMENT* defined as follows:

$$
\begin{array}{lll}
ELEMENT & ::= & \text{“}El\text{(”}\ expression\ \text{“)”} \\
 & & \mid\ \text{“}F\text{(}El\text{(”}\ expression\ \text{“),”}\ abstract\ ELEMENT\ \text{“)”}
\end{array}
$$

Again for future reference we call the variables $x_1,\ldots,x_m$ the *θ-introduction variables.*

The θ-introduction rule may be recursive: that is, one or more of the premises of the rule may take the form that, in a certain context, $b \in \Theta(l)$ for some expression b and some list of expressions l. (For such a premise to make sense the inference rule must be internally consistent as defined earlier.) If the ith premise is indeed recursive we refer to x_i as a *recursive θ-introduction variable.*

3.3 Elimination Rules

For each type constructor Θ there is exactly one elimination rule. Let us suppose the elimination rule introduces the non-canonical constant Θrec, and that there are k introduction rules defining canonical constants $\theta_1,\ldots,\theta_k$. Then the elimination rule is formed as follows.

$$
\begin{array}{ll}
0 & \langle\text{premises of } \Theta\text{-formation}\rangle \\
1 & a \in \Theta(A_1,\ldots,A_n) \\
2 & |[\ w \in \Theta(A_1,\ldots,A_n) \\
 & \triangleright\ C(w)\ type \\
 &]| \\
3 & |[\ \langle\text{context computed from } \theta_1\text{-introduction rule}\rangle \\
 & \triangleright\ z_1(l_1,s_1) \in C(\theta_1(l_1)) \\
 &]| \\
 & \vdots \\
2+k & |[\ \langle\text{context computed from } \theta_k\text{-introduction rule}\rangle \\
 & \triangleright\ z_k(l_k,s_k) \in C(\theta_k(l_k)) \\
 &]| \\
 & \hrulefill \qquad \Theta\text{-elimination} \\
 & \Theta rec(a,z_1,\ldots,z_k) \in C(a)
\end{array}
$$

The premises are divided into four parts. In the first part the premises of Θ-formation are repeated once again (and also once again rarely explicitly). The second part postulates the existence of an object "a" of type Θ (where "a" is a new identifier). The third part postulates that C (where "C" is a new identifier) is a family of types indexed by objects of type Θ. Finally the fourth part consists of a premise for each canonical-object constructor θ. (In the above schema $z_1,\ldots,z_k$ are new variables and $s_1,\ldots,s_k,l_1,\ldots,l_k$ are lists of variables constructed from the introduction rules in in a manner to be described shortly.) A summary of the Θ-elimination rule would be that the proof of a property $C(a)$ given object a of type Θ proceeds by structural induction, i.e. by case analysis on the possible form of the canonical value of a.

The premise (referred to later as p_θ) corresponding to the θ-introduction rule takes the form:

$$
\begin{array}{l}
|[\ \langle\text{context computed from } \theta\text{-introduction rule}\rangle \\
\triangleright\ z(l,s) \in C(\theta(l)) \\
]|
\end{array}
$$

where l is simply the list of θ-introduction variables but where the construction of the context and the list of θ-elimination variables, s, depends on whether the introduction rule is or is not recursive. The details of their construction are as follows.

Suppose that the θ-introduction rule has the following form.

⟨premises of Θ-formation⟩
|[⟨context 1⟩
▷ $b_1 \in B_1$
]|
...
|[⟨context m⟩
▷ $b_m \in B_m$
]|

$\theta(l) \in \Theta(t)$

where $b_1, \ldots, b_m, B_1, \ldots, B_m$ are expressions, l is the list of θ-introduction variables and t is the list of Θ-formation variables. Then the premise, p_θ, to be included in the Θ-elimination rule has the form

|[⟨*assumption*(s) 1⟩
; ...
; ⟨*assumption*(s) m⟩
▷ $z(l, s) \in C(\theta(l))$
]|

Here "⟨assumption(s) k⟩" refers to either one or two assumptions depending on whether x_k is or is not a recursive θ-introduction variable. In the case that x_k is not recursive then "⟨assumption(s) k⟩" is simply a repetition of the corresponding premise in the θ-introduction rule. That is "⟨assumption(s) k⟩" is

|[⟨context k⟩
▷ $b_k \in B_k$
]|

On the other hand if x_k is a recursive θ-introduction variable then, by definition, the corresponding premise of the θ-introduction rule takes the form

|[⟨context k⟩
▷ $x_k(u_k) \in \Theta(l)$
]|

for some list of variables u_k. In this case ⟨assumption(s) k⟩ consists of

(a) a repetition of the premise as in the case of a non-recursive θ-introduction variable, and

(b) the assumption

|[⟨context k⟩
▷ $y_k(u_k) \in C(x_k(u_k))$
]|

where y_k is a new variable.

The list, s, of θ-elimination variables is then just the list of variables, y_k, that are introduced by the recursive θ-introduction variables.

3.4 Computation Rules

The computation rules for type Θ are in (1-1) correspondence with the introduction rules for Θ. Thus for each canonical-object-constructor, θ say, there is exactly one computation rule which prescribes how to apply Θrec to a θ-object. Again the method of constructing the rule is complicated by the presence of recursive introduction variables.

For the purpose of this discussion let us identify θ with its index in the list of canonical-object constructors for the type Θ. Also let us denote by l_θ the list of θ-introduction variables.

In general the computation rule for θ-objects is a combination of the θ-introduction rule and the Θ-elimination rule. A schema for its construction is as follows.

0	⟨premises of Θ-formation⟩
1	⟨premises of θ-introduction (excluding Θ-formation premises)⟩
2...2+k	⟨premises $2 \ldots 2+k$ of Θ-elimination⟩

$$\frac{}{\Theta rec(\theta(l_\theta), z_1, \ldots, z_k) = z_\theta(l_\theta, v) \in C(\theta(l_\theta))} \quad \theta\text{-computation}$$

Apart from the construction of the list of expressions v (which we have yet to describe) the construction of the θ-computation rule is thus very straightforward.

There is an expression in the list v for each recursive θ-introduction variable. Suppose $x_i (1 \leq i < m_\theta)$ is such a variable and the corresponding premise of the θ-introduction rule is

$$\begin{array}{l} [\![\ \langle \text{context } \ i \rangle \\ \triangleright \ \ x_i(u_i) \in \Theta(t) \\ ]\!] \end{array}$$

Then the entry in the list v takes the form

$$(u_i)\Theta rec(x_i(u_i), z_1, \ldots, z_k)$$

3.5 Examples

We present several examples of the construction of the elimination and computation rules. First consider the disjoint sum type. This has the formation rule:

$$\frac{\begin{array}{l}A_1\ type\\ A_2\ type\end{array}}{A_1 \vee A_2\ type} \qquad \vee\text{-formation}$$

and two introduction rules:

$$\frac{\begin{array}{l}A_1\ type\\ A_2\ type\\ x \in A_1\end{array}}{i(x) \in A_1 \vee A_2} \qquad \text{i-introduction}$$

$$\frac{\begin{array}{l}A_1\ type\\ A_2\ type\\ x_1 \in A_2\end{array}}{j(x) \in A_1 \vee A_2} \qquad \text{j-introduction}$$

The list of $\vee$-formation variables is thus (A_1, A_2), the list of i-introduction variables is (x) and the list of j-introduction variables is also (x).

Referring back to section 3.3 we construct the following elimination rule.

$$\frac{\begin{array}{ll}0 & A_1\ type\\ & A_2\ type\\ 1 & a \in A_1 \vee A_2\\ 2 & [\!|\ w \in A_1 \vee A_2\\ & \triangleright\ C(w)\ type\\ & |\!]\\ 3 & [\!|\ x \in A_1\\ & \triangleright\ z_1(x) \in C(i(x))\\ & |\!]\\ 4 & [\!|\ x \in A_2\\ & \triangleright\ z_2(x) \in C(j(x))\\ & |\!]\end{array}}{\vee\text{-}elim(a, z_1, z_2) \in C(a)} \qquad \vee\text{-elimination}$$

Also, referring to section 3.4 we construct the following computation rules

$$\begin{array}{ll} 0 & A_1\ type \\ & A_2\ type \\ 1 & x \in A_1 \\ 2\ldots 4 & \langle \text{as in } \vee\text{-elimination} \rangle \\ \multicolumn{2}{l}{\hrulefill} \quad \text{i-computation} \\ \multicolumn{2}{l}{\vee\text{-}elim(i(x), z_1, z_2) = z_1(x) \in C(i(x))} \end{array}$$

$$\begin{array}{ll} 0 & A_1\ type \\ & A_2\ type \\ 1 & x \in A_2 \\ 2\ldots 4 & \langle \text{as in } \vee\text{-elimination} \rangle \\ \multicolumn{2}{l}{\hrulefill} \quad \text{j-computation} \\ \multicolumn{2}{l}{\vee\text{-}elim(j(x), z_1, z_2) = z_2(x) \in C(i(x))} \end{array}$$

Our second example is concocted to illustrate the problems of recursive introduction variables. The formation rule is as follows.

$$\frac{A\ type}{H(A)\ type} \quad \text{H-formation}$$

The type has one introduction rule

$$\frac{\begin{array}{l} A\ type \\ [[\ v \in A \\ \triangleright\ x(v) \in H(A) \\ ]] \end{array}}{h(x) \in H(A)} \quad \text{h-introduction}$$

Note that the h-introduction variable x is recursive by virtue of the premise "$x(v) \in H(A)$".

From these two rules we compute the H-elimination rule according to the schema described in section 3.3

$$
\begin{array}{ll}
0 & A\ type \\
1 & a \in H(A) \\
2 & |[\ w \in H(A) \\
 & \triangleright\ C(w)\ type \\
 &]| \\
3 & |[\ |[\ v \in A \\
 & \quad \triangleright\ x(v) \in H(A) \\
 & \quad]| \\
 & ;\ |[\ v \in A \\
 & \quad \triangleright\ y(v) \in C(x(v)) \\
 & \quad]| \\
 & \triangleright\ z(x,y) \in C(h(x)) \\
 &]| \\
\hline
 & H\text{-}rec(a,z) \in C(a)
\end{array}
\qquad \text{H-elimination}
$$

Finally the computation rule takes the following form.

$$
\begin{array}{ll}
0 & A\ type \\
1 & |[\ v \in A \\
 & \triangleright\ x(v) \in H(A) \\
 &]| \\
2\ldots 3 & \langle \text{as in } H\text{-elimination} \rangle \\
\hline
 & H\text{-}rec(h(x),z) = z(x,(v)H\text{-}rec(x(v),z)) \in C(h(x))
\end{array}
\qquad \text{h-computation}
$$

We conclude our list of examples with the definition of the W-type.

The W-type stands out in Martin-Löf's presentation of his theory [ML0] because it is the only one that appears to require an understanding of other types in the theory. In particular the rules given there appeal to an understanding of the Π-type and of "$\rightarrow$", a symbol that is not defined (although Martin-Löf does make its meaning clear elsewhere [ML1]). One important aspect of the rationalisation of the theory, mentioned by Martin-Löf in the Padova Notes [ML1], detailed by Nordström, Peterson and Smith [NPS] and exploited by us in this paper, is that this lacuna has been overcome. The W-type is also the one that is considered to be the hardest to understand. Our contribution is thus to show that it may be understood solely by understanding its introduction rule and the general scheme for inferring elimination and computation rules.

The W-formation rule is as follows.

$$
\begin{array}{l}
A_1\ type \\
|[\ x \in A_1 \\
\triangleright\ A_2(x)\ type \\
]| \\
\hline
W(A_1,A_2)\ type
\end{array}
\qquad \text{W-formation}
$$

The W-type also has just one (recursive) introduction rule:

$$\frac{\begin{array}{l}\langle\text{premises of } W\text{-formation}\rangle\\ x_1 \in A_1\\ [\!|\ v \in A_2(x_1)\\ \triangleright\ x_2(v) \in W(A_1, A_2)\\ |\!]\end{array}}{sup(x_1, x_2) \in W(A_1, A_2)}\ \text{sup-introduction}$$

According to the schema for its construction the W-elimination rule therefore takes the following form.

$$\frac{\begin{array}{ll}0 & \langle\text{premises of } W\text{-formation}\rangle\\ 1 & a \in W(A_1, A_2)\\ 2 & [\!|\ w \in W(A_1, A_2)\\ & \triangleright\ C(w)type\\ & |\!]\\ 3 & [\!|\ x_1 \in A_1\\ & ;\ [\!|\ v \in A_2(x_1)\\ & \quad \triangleright\ x_2(v) \in W(A_1, A_2)\\ & \quad |\!]\\ & ;\ [\!|\ v \in A_2(x_1)\\ & \quad \triangleright\ y(v) \in C(x_2(v))\\ & \quad |\!]\\ & \triangleright\ z(x_1, x_2, y) \in C(sup(x_1, x_2))\\ & |\!]\end{array}}{W\text{-}rec(a, z) \in C(a)}\ \text{W-elimination}$$

Finally the single computation rule takes the following form.

$$\frac{\begin{array}{ll}0 & \langle\text{premises of } W\text{-formation}\rangle\\ 1 & x_1 \in A_1\\ & [\!|\ v \in A_2(x_1)\\ & \triangleright\ x_2(v) \in W(A_1, A_2)\\ & |\!]\\ 2..3 & \langle\text{as in } W\text{-elimination}\rangle\end{array}}{W\text{-}rec(sup(x_1, x_2), z) = z(x_1, x_2, (v)W\text{-}rec(x_2(v), z))}\ \text{sup-computation}$$

Conclusions

Martin-Löf's theory of types has a rich structure which we hope this paper has helped to expose. Our account must, however, be recognised as very preliminary. This section is therefore devoted to a description of the work that we plan to do in the near future.

To begin with there are certain flaws in the above account. In particular, it has been pointed out to us that additional constraints apply to the use of recursive introduction variables. Thus in the first draft of this paper our example of h-introduction (see section 3.5) had the premise

$$
\begin{array}{ll}
[\![& v \in H(A) \\
\triangleright & x(v) \in H(A) \\
]\!] &
\end{array}
$$

which should be prohibited on account of the fact that there is a negative occurrence of a recursive introduction variable leading to nonterminating programs. (I am grateful to Per Martin-Löf for pointing this out.) This highlights a lack of a semantic justification of the scheme we have described, but which we intend to remedy.

Secondly we intend to describe schemes for the construction of derivations of closure properties and uniqueness properties of a type. Closure properties are properties like "every element of a disjoint sum is either of the form $i(a)$ or $j(b)$ for elements a and b of the appropriate type" and uniqueness properties express the fact that objects introduced by distinct introduction rules are always distinct. Thus the two sets of properties reflect the fact that types introduced into the theory are extremal, and, of course, they are fundmental to our understanding. For particular instances of such derivations the reader is referred to [Ba].

Thirdly, we intend to extend the construction to novel type structures such as the subset type [Co,NPS] in which the type introduction rules incur information loss. For the subset type and similar type constructors that we have in mind the way ahead is clear. The quotient type introduced by the PRL group [Co] is much less clear to us.

Finally, we intend to try to provide a collection of examples that illustrate our thesis that the ability to introduce new type constructors is an indispensable feature of the theory - and, consequently, of implementations of the theory.

References

[Ba] R.C. Backhouse "Notes on Martin-Löf's Theory of Types"
Department of Mathematics and Computing Science,
Rijksuniversiteit Groningen, Sections 1 and 2, August 1987.

[Be] M.J. Beeson *Foundations of Constructive Mathematics*
Springer-Verlag, Berlin (1985).

[BL] R. Burstall and B.Lampson
"A kernel language for abstract data types and modules,"
In *Semantics of Data Types*, Eds. G.Kahn, D.B.MacQueen and G.Plotkin,
Springer-Verlag Lecture Notes in Computer Science *173*, 1-50 (1984).

[Ch] P. Chisholm "Derivation of a parsing algorithm in Martin-Löf's Theory of Types,"
Science of Computer Programming *8* (1987) 1-42.

[Co] R.L. Constable et al
Implementing Mathematics with the Nuprl Proof Development System
Prentice-Hall, Inc., Englewood Cliffs, N.J. (1986).

[DF] E.W. Dijkstra and W.H.J. Feijen
Een methode van programmeren,
Academic Service, 's-Gravenhage (1984)

[Dy] R. Dyckhoff "Category theory as an extension of Martin-Löf's Type Theory,"
University of St. Andrews (1985).

[Kh] A.M.A. Khamiss "Program Construction in Martin-Löf's Theory of Types,"
Ph.D. Thesis, University of Essex, Dept. of Computer Science (1986).

[ML0] P. Martin-Löf "Constructive Mathematics and Computer Programming,"
pp. 153-175 in Logic, Methodology and Philosophy of Science, VI,
North-Holland Publishing Company, Amsterdam (1982),
Proceedings of the 6th International Congress, Hannover, 1979.

[ML1] P. Martin-Löf "Intuitionistic Type Theory,"
Notes by Giovanni Sambin of a series of lectures given in Padova,
June 1980.

[No] B. Nordström "Multilevel Functions in Type Theory,"

[NPS] B. Nordström, K. Peterson and J. Smith
"An Introduction to Martin-Löf's Type Theory," Draft, midsummer 1986,
Programming Methodology Group, Chalmers University of Technology,
S-41296 Göteborg, Sweden.

Acknowledgements My thanks go to Bengt Nordström and Per Martin-Löf both of whom have provided helpful criticisms of this report. Thanks go also to Kees Straatman for helping me to win the battle with TEX.

This work was begun whilst I was still employed by the Department of Computer Science of the University of Essex and was supported at that time by a grant from the Science and Engineering Research Council. I am grateful to both organisations for their support.