BCS FACS

ONE DAY SEMINAR

Professor Roland Backhouse
University of Groningen

at Imperial College
Room 145 Huxley

on Monday 14 September 1987
1030 - 1600

THE MARTIN-LOF THEORY OF TYPES
A COMPUTING SCIENTIST'S PERSPECTIVE

1. **Introduction - An insight into constructive proofs and their relation to programs**
2. **The construction of the rules with examples, lists and finite bags**
3. **An example of algorithm development**

This list of topics is provisional and may change

£15 to include coffee, lunch and tea
(Cheques payable to BCS FACS)

Enquiries to: Jennifer Eastwood, Dept of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ
01-589 5111 ext 5021 Telex 261503

Notes on Martin-Löf's Theory of Types

Roland Backhouse
Subfaculteit Wiskunde en Informatica
Rijksuniversiteit Groningen
Postbus 800
9700 AV GRONINGEN
The Netherlands

Every so often in the development of a science a new topic emerges that captures everyone's imagination and instigates a furore of activity. A topic that has recently captured my imagination and the imagination of a small, but growing, number of computing scientists is Per Martin-Löf's theory of types [M-L1]. It has yet to capture the imagination of a large body of computing scientists, although I believe it will eventually do so, partly because of seemingly inevitable difficulties of communication between logicians and computing scientists. These notes are an attempt to overcome some of these difficulties by explicating the theory with a substantial number of examples. The examples I have chosen, and the order and style of presentation, will, I hope, appeal to computing scientists with an interest in programming as a mathematical activity.

By way of motivation, and in order to explain why I, personally, am impressed by the theory, I shall begin with a very brief review of some of the more important advances that have been made in the "mathematics of programming" since 1968. I have taken 1968 as the starting point since that was the year of a now-famous NATO conference on Software Engineering in Garmisch, West Germany *.It was at that conference, I believe, that the term "software crisis" was coined. More importantly, it was at that conference that the computing community became publicly aware of the vital need for a theory of programming.
The four developments that I discuss are these.

Data Structuring
Functional Programming
Logic Programming
Program Verification

The first of these, the introduction of type declarations (enumerated types, record structures etc.) into programming languages is also, historically, the first to have had a significant impact on the way we program. C. A. R. Hoare's suggestions on data structuring [Ho], which were subsequently realised in the programming language Pascal, were made with the expressed aim of *"extend(ing) the range of programming errors which logically cannot be made"*.

That this objective was achieved is undoubtedly true, but the notion of *strong typing* — the requirement that the left and right sides of all assignments (implicit or explicit) have identical type — introduced in Pascal to achieve that objective involved a severe penalty of inflexibility. For example, it is impossible to define in Pascal an identity function — a

* 1968 was also the year that C.A.R.Hoare took up a chair at Queen's University, Belfast.

function that takes an arbitrary value as argument and returns the same value as result. It is, however, possible to write *separate* identity functions for integers, for Booleans etc. This may not seem to be a very significant example; its significance becomes more apparent when one realises that separate, but essentially identical, procedures are needed to search a list of widgets for a widget and to search a list of thingummyjigs for a thingummyjig.

One of the benefits of some functional programming languages was to liberate us from the strait-jacket of strong-typing without compromising Hoare's stricture on extending the range of errors which logically cannot be made. Thus in the language ML, developed by Robin Milner and his colleagues as part of the Edinburgh LCF system [GMW] one *can* define the identity function — it takes the form $id = \lambda x.x$ and has the *polymorphic* type $* \rightarrow *$, meaning that it maps an element of arbitrary type $*$ into an element of the same type $*$. Nevertheless, there is a strict regime governing type correctness of programs that prevents many involuntary errors. ("Polymorphic" means "having many forms" and, indeed, polymorphic functions appear in many languages but in the role of second-class citizens. For instance, the function new in Pascal is polymorphic since it returns a pointer of arbitrary type. The term was apparently invented by Christopher Strachey and is distinct from "overloading" such as occurs in the use of "+" to denote both integer and real addition [Mi1]).

In spite of its undoubted advances there are still shortcomings in the type-definition mechanism in ML (and in Standard ML [Mi2]). One such is that, for example, the addition and multiplication functions on integers both have the *same* type $int \times int \rightarrow int$ as does the integer division function, div. There is, thus, no mechanism in the language to record the different algebraic properties of addition and multiplication (the fact that 0 is the identity of the former and 1 the identity of the latter, etc.); nor is there any mechanism (in the type structure) to indicate that addition and multiplication are everywhere-defined functions whereas integer division is undefined when its second argument is zero.

Another shortcoming of the type-mechanism in ML is that there is no notion of a *dependent* type, in which components of a type may depend on the values held by previously-defined types. An example of a dependent type is the type semigroup. An element of the type semigroup is a set S, say, together with an associative binary operator $+$, say, defined on the elements of S. The point to note about this definition is that a semigroup has two components, the second of which has type $S \times S \rightarrow S$ which *depends* on the set, S, defined in the first component.

The third topic on my list, logic programming, is often identified with programming in Prolog. Prolog allows statements to be made in a limited form of the predicate calculus called Horn-clause form. Horn clauses are interpreted procedurally so that a set of one or more clauses describes a set of one or more recursive procedures.

There is no doubt that Prolog has achieved a great deal in highlighting the value of formal logic to programming; my reference to logic programming is, however, to a rather broader understanding of the nature of programming as a mathematical activity requiring an unusual degree of formality and rigour.

The final topic on my list, program verification, is for me the most fundamental. But, although its development began about the same time as the development of data-structuring techniques, it is probably the topic that has had the least effect on the way that practising

programmers develop software. Its effects have been emasculated because the techniques of program verification have never been properly integrated into a programming language. It is still possible to write programs without having the slightest clue about program proofs, invariant properties etc. and those few programmers who do have such knowledge often regard program proofs as a gross incumbrence and impossible to use except for "toy" problems.

I am impressed by Martin-Löf's theory because it seems to combine within the same framework many of the advances I have been discussing. It is a logical system, developed from Gentzen's system of natural deduction [Ge], that formalises constructive mathematics in the style of Bishop [Bi]. It incorporates very powerful type-definition facilities, including the notion of dependent types mentioned earlier and it embodies an extremely important principle, the so-called principle of "propositions as types".

In Martin-Löf's theory propositions are identified with types or specifications, and programs are identified with proofs. For example, for an arbitrary type A, the proposition $A \Rightarrow A$ (A "implies" A) is identified with the type $A \rightarrow A$ of all total functions mapping objects of type A into objects of type A. A proof of the validity of $A \Rightarrow A$ would be accomplished by exhibiting the identity function $\lambda x.x$. Thus, in type theory the construction of a proof and the construction of a program are one and the same thing, and it is no longer the case that one is a tiresome and unnecessary addendum to the other. Moreover, the type structure described by the theory is sufficiently rich that it can be usefully used as the basis for a practical specification language.

These notes are an attempt to present the theory from a computing scientists perspective. For this reason, I have taken extreme liberties both with notation and presentation, so much so that the uninitiated may not be able to recognise the original. The notes proceed rather slowly at first and it may be some time before the reader sees any "real" examples. Moreover, the early sections stress the formality of the system. I make no apoligies; take it or leave it.

1. Propositions as Types

We begin our account of Martin-Löf's theory by explicating the notion of "propositions as types". This is an important notion that originated in the development of intuitionistic logic [M-L2]. It is not an easy notion to grasp and its full explanation extends beyond this section.

In outline, Martin-Löf's theory is a formal system for making judgements about certain well-formed formulae. Such judgements take one of four possible forms, the most important — to us — being the forms

$$p \in P$$

A judgement of the form $p \in P$ can be read in several different ways. In the conventional computing science sense it is read as "p has type P", or "p is a member of the set P". For example, we write $0 \in \mathcal{N}$ meaning "zero has the type natural number", or "zero is a member of the set of natural numbers". In intuitionistic logic it admits a different reading. If P is a proposition (i.e. a well-formed formula constructed from the propositional connectives $\wedge$, $\vee$ etc.) then the judgement form $p \in P$ means that p is a summary of a (constructive) proof of P. This section contains a large number of examples to illustrate this reading. Finally, we shall also see that propositions may be regarded as problem specifications. In this sense the judgement form $p \in P$ may be read as "p is a program that achieves the specification P". Examples of this interpretation of the judgement form do not emerge until section??? We use whichever of the four readings seems natural or appropriate at the time. We also use the more neutral terminology "p is an object of P" when no particular interpretation is intended.

This identification between objects of a type, proofs of propositions, and programs achieving specifications is an extremely attractive idea relying on a constructive, rather than classical, interpretation of the propositional connectives. Readers familiar with classical accounts of the propositional and predicate calculi should not be put off by this, but it may help to interpose some brief remarks on the difference between constructive and non-constructive proofs.

Implicit in any constructive proof of a proposition is an algorithm that constructs a witness to the truth of the proposition. For example, a constructive proof of the statement that there are infinitely many primes must (implicitly or explicitly) show how to construct an infinite sequence of primes. The following "classical" proof given by Courant and Robbins [CR] is non-constructive.

> "The proof of the infinitude of the class of primes as given by Euclid remains a model of mathematical reasoning. It proceeds by the "indirect method". We start with the tentative assumption that the theorem is false. This means that there would be only a finite number of primes, perhaps very many — a billion or so — or, expressed in a general and non-commital way, n. Using the subscript notation we may denote these primes by $p_1, p_2, \ldots, p_n$. Any other number will be composite, and must be divisible by at least one of the primes $p_1, p_2, \ldots, p_n$. We now produce a contradiction by constructing a number A which differs from every one of the primes $p_1, p_2, \ldots, p_n$

because it is larger than any of them, and which nevertheless is not divisible by any of them. This number is

$$A = p_1 p_2 ... p_n + 1,$$

i.e. 1 plus the product of what we supposed to be all the primes. A is larger than any of the p's and hence must be composite. But A divided by p_1 or by p_2, etc., always leaves the remainder 1; therefore A has none of the p's as a divisor. Since our initial assumption that there is only a finite number of primes leads to this contradiction, the assumption is seen to be absurd, and hence its contrary must be true. This proves the theorem."

The proof is non-constructive because A may or may not be prime, and therefore a method has not been given for continuing the sequence beyond p_n. However, as Courant and Robbins observe, it can easily be converted into a constructive proof.

"Although this proof is indirect, it can easily be modified to give a method for constructing, at least in theory, an infinite sequence of primes. Starting with any prime number, such as $p_1 = 2$, suppose we have found n primes $p_1, p_2, \ldots, p_n$; we then observe that the number $p_1 p_2 \ldots p_n + 1$ either is itself a prime or contains as a factor a prime which differs from those already found. Since this factor can always be found by direct trial, we are sure in any case to find at least one new prime p_{n+1}; proceeding in this way we see that the sequence of constructible primes can never end."

Note that the transformation of the non-constructive proof into a constructive proof hinges on the fact that one can always test whether A is or is not prime. The statement that A is either prime or is not prime is an instance of the law of the excluded middle.

In constructive mathematics the law of the excluded middle is not universally valid. That is, if P is an arbitrary proposition it is not the case that $P \vee \neg P$ is necessarily true. For such a statement to be true it is required that a constructive proof of P or of $\neg P$ be supplied, together with information as to which has been proved. In other words, as in the case of the primality of A, we have to provide a method for testing the truth or falsity of P.

Another example of the role of the law of the excluded middle in constructive proofs arises from the following proof of the pigeon-hole principle [PRL]. The principle can be formulated as "given a function f from the interval $\{1 \ldots m\}$ into the interval $\{1 \ldots n\}$, where $n < m$, then there exist i and j such that $1 \leq i < j \leq m$ and $f(i) = f(j)$". A classical proof uses proof by contradiction. Assume that $n < m$ and, for all i,j such that $1 \leq i < j \leq m$, $f(i) \neq f(j)$. Then it is also the case that, for all i,j, such that $1 \leq i < j \leq n$, $f(i) \neq f(j)$. Clearly, therefore, for each k, $1 \leq k \leq n$, there is some i such that $1 \leq i \leq n$, and $f(i) = k$. In particular, there is some i, $1 \leq i \leq n$ such that $f(i) = f(n+1)$. This contradicts the initial assumption which must therefore be invalid. We thus conclude the pigeon-hole principle.

The proof is non-constructive because no method is exhibited for computing distinct i and j in the range $\{1 \ldots m\}$ for which $f(i) = f(j)$. However, consider the following instance of the law of the excluded middle. "For all $f : \{1 \ldots m\} \rightarrow \{1 \ldots n\}$ it is the case that $f(i) \neq f(j)$ for all distinct i and j or it is not the case that $f(i) \neq f(j)$ for all distinct i and j". A constructive proof of this proposition will entail an algorithm to determine

distinct i and j for which $f(i) = f(j)$, if such exist. Using such an algorithm our earlier non-constructive proof becomes constructive: apply the algorithm to determine whether distinct i and j in the range $1 \ldots n$ exist for which $f(i) = f(j)$. If such i and j exist, exhibit them as witnesses to the pigeon-hole principle. If not, exhibit the pair whose first component is $n+1$ and whose second component is the value i in the range $1 \ldots n$ satisfying $f(i) = f(n+1)$.

The standard example of a non-constructive proof is the following. The proposition is "there exists a pair of irrational numbers a and b such that a^b is rational". The proof is to consider $\sqrt{2}^{\sqrt{2}}$. If this is rational then let $a = \sqrt{2}$ and $b = \sqrt{2}$. If it is not rational then it is irrational, and since $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = (\sqrt{2})^2 = 2$, the assignment $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$ satisfies the proposition. Thus in either case we have exhibited a pair of irrational numbers a and b such that a^b is rational. This proof is non-constructive because we have no way of telling whether the pair $\langle \sqrt{2}, \sqrt{2} \rangle$ or the pair $\langle \sqrt{2}^{\sqrt{2}}, \sqrt{2} \rangle$ satisfies the proposition.

Consider now the proposition "(for all numbers a and b) either a^b is rational or a^b is irrational". A constructive proof of this proposition is effectively a method (or algorithm) for determining, for arbitrary numbers a and b, whether a^b is rational or irrational. Were such an algorithm available, in particular had we a proof that $\sqrt{2}^{\sqrt{2}}$ is rational or had we a proof that $\sqrt{2}^{\sqrt{2}}$ is irrational, then the earlier "non-constructive" proof of the existence of irrational numbers a and b for which a^b is rational would indeed be constructive. In this case, however, it is believed that there is no general algorithm for resolving whether a given real number is rational or irrational, and the particular case of $\sqrt{2}^{\sqrt{2}}$ is (to my knowledge) as yet unresolved.

These examples have been included to assuage the reader's fears that intuitionistic reasoning is awkward. It is certainly not the case that the law of the excluded middle is *never* valid. We refer to a proposition P for which a constructive proof of $P \vee \neg P$ can be proved as a *Boolean* or *decidable* proposition. When working with such propositions, classical reasoning is perfectly acceptable. Moreover, we shall formulate several programming problems as instances of the law, thus taking advantage of its absence.

1.0 Preliminaries.

Making a start on a new topic is always difficult, both for authors and for students, since any 'start' is never truly a beginning but relies on some minimal, shared understanding. Making a start on the philosophical foundations of mathematics, which was Martin-Löf's objective, is especially difficult. I shall make a start by simply asking the reader to take for granted, at least throughout this first section, a number of apparently ad hoc rules. The sole purpose of introducing them now is so that we can quickly progress, in a completely formal way, to non-trivial examples; all the rules in this section will be introduced again in a more appropriate context. (In so doing, however, I am aware that I shall be criticised by at least one referee for not discussing immediately whether there is a type of all types — but such a discussion will have to wait until much later.)

We assume that the reader is familiar with the use of types and the notion of scope in conventional programming languages. We also assume knowledge of and experience with the λ-notation for naming functions in the λ-calculus, the notion of free and bound variables, and the α, β and η conversion rules. (Stoy's well-known book [St] is more than adequate for our purposes.)

The primitive types in Martin-Löf's theory include the set of natural numbers, denoted $\mathcal{N}$, and enumerated types like $\{nil\}$ and $\{red, yellow, blue\}$. Formally, we have the rule

(0) $\quad \mathcal{N}\ type$

which is read as "$\mathcal{N}$ (the set of natural numbers) is a type". The statement $\mathcal{N}\ type$ is called a *judgement*. (There are four judgement forms but only two are considered until section 2.)

The objects of $\mathcal{N}$ are of course 0, 1, 2 etc. For the time being we only need the fact that 0 is an object of $\mathcal{N}$, which is written formally as the judgement

(1) $\quad 0 \in \mathcal{N}$

(and read "0 is an object of $\mathcal{N}$"). Similarly we have the judgements

(2) $\quad nil \in \{nil\}$

and

(3) $\quad red \in \{red, yellow, blue\}$

The judgement form "$p \in P$" is central to Martin-Löf's theory and can be read as "p is an object of P" or "p has type P". (As mentioned earlier, we also interpret it in other ways.)

The primitive types are objects of a universe of types denoted by U_1. (As the notation suggests there is, in fact, a hierarchy of universes U_1, U_2, ... but this need not trouble us just yet.) The universe, U_1, is itself a type so we have the judgements

(4) $\quad U_1\ type$

and (for example)

(5) $\quad \mathcal{N} \in U_1$

Already, by introducing U_1 at this stage, we have diverged from Martin-Löf's preferred order of presentation of his theory. Indeed, in the theory a judgement of the form $P\ type$ has a rather precise meaning to do with knowing how to form objects of P, and with knowing

when two objects of P are equal. Since we haven't said how to form objects of U_1, and we have deliberately postponed the consideration of equalities until much later in the text, we must ask the reader's forbearance in accepting half-formed notions until they can be explained properly. For now, we have a rather shallow understanding of the judgement P *type* as meaning simply that P is a well-formed formula, or P is a syntactically-correct type expression.

Our first example of an inference rule is the following.

$$\frac{P \in U_1}{P \ type} \qquad U_1\text{-elimination}$$

The rule states that if P is an object of U_1 then P is a type. Using it we can infer (0) from (5) as follows:

0 $\quad \mathcal{N} \in U_1$

$\quad\quad \%0, U_1\text{-elim.}\%$

1 $\quad \mathcal{N}$ *type*

The two steps 0 and 1 form a *proof-derivation.* The numbering of steps and the explanations enclosed within percent signs are not part of the derivation but appear only as aids to the reader.

Terminology. Objects of U_1 are generally referred to as the *small types.* Thus $\mathcal{N}$ is a small type. We shall adopt this terminology from now on.

1.1 Assumptions and Scope

An important aspect of the theory is the ability to make hypothetical judgements, that is judgements that depend on certain assumptions. An assumption is introduced using the following rule of inference.

$$\frac{P\ \mathit{type}}{\begin{array}{l}[\![\ x \in P \\ \triangleright \\ ]\!]\end{array}}\qquad \text{Assumption}$$

In the inference part of this rule the symbol x denotes a variable and can be any identifier, sign or mark that does not occur free in P. (It should however be readily distinguishable from other syntactic marks in the language.) The brackets [[and]] delimit the *scope* of the assumption and are pronounced "scopen" and "sclose", respectively.

The two occurrences of P stand for arbitrary expressions. In a valid use of the rule they must be replaced by *definitionally equal* expressions. For the moment you may read "definitionally equal" as identical but it has a broader meaning. (For example we shall shortly introduce a number of abbreviations; an expression and its abbreviation are then said to be definitionally equal.)

We verbalise the rule as the statement that if P is a type it is permissible to introduce a new *context* (delimited by the scope brackets) in which it is assumed that x is an object of P.

Assumptions look like variable declarations in conventional programming languages (particularly if one uses **begin** and **end** instead of [[and]]), and there is no harm in adopting this view. Indeed, a vital (meta-) rule of the formal system is that *any use of a variable must be within the scope of a declaration of the variable.*

Example 1.0

$$\begin{array}{l}[\![\ A \in U_1 \\ \triangleright\ A\ \mathit{type} \\ ]\!]\end{array}$$

In a context in which A is assumed to be a small type (an object of U_1), A is a type.

Derivation

```
0     U1 type
        % 0, assumption %
1.0   [[  A ∈ U1
      ▷     % 1.0, U1-elim. %
1.1       A type
      ]]
```

Step 0 is a statement of the primitive rule that U_1 is a type. This enables the use of the assumption rule, in step 1.0, to introduce a context in which A is assumed to be a small type. Note that step 1.0 simultaneously introduces the sclose bracket at the end of the derivation.

(End of example)

NB. The use of scope brackets in these notes is the most radical departure from Martin-Löf's system. The difference is one of presentation, and not of semantics, but it is a difference that I regard as very important. For illustrations of Martin-Löf's style of proof derivation see [Be] and [M-L2]. Petersson [Pe] has implemented this style as a replacement for PPLambda in the Edinburgh LCF system [GMW]. For an alternative, top-down, style of proof derivation see the NuPRL system [PRL].

1.2 Type Formation Rules

In this section we present a number of type-formation rules — rules for making judgements of the form P *type*. We also take the opportunity to introduce informally the notion of propositions as types. In presenting the rules —here and elsewhere — I use P, Q, R to denote type expressions, p, q, r, s to denote object expressions and w, x, y, z to denote variables.

(a) Implication

$$\frac{\begin{array}{l} P \text{ type} \\ [\![\ x \in P \\ \triangleright\ Q \text{ type} \\ ]\!] \end{array}}{P \Rightarrow Q \text{ type}} \qquad \Rightarrow\text{-formation}$$

The $\Rightarrow$-formation rule has two premises. The first premise is that P is a type. The second premise is that in a context in which it is assumed that x is an object of P it is possible to prove that Q is a type. If these two premises are satisfied then it is possible to infer that $P \Rightarrow Q$ is a type. (Note that the conclusion is read as $(P \Rightarrow Q)$ is a type, not $P \Rightarrow (Q$ is a type). Viewed as a unary operator, the keyword *type* has lowest precedence.)

The rule is said to *discharge* the assumption, $x \in P$, in the second premise, and x is not in scope in the conclusion. As a consequence, the expression Q cannot contain any free occurrences of x since, otherwise, the scope rule would be violated.

The complicated second premise permits the use of *conditional* or *nonstrict* implication in type expressions. For example, the expression $(\forall n \in \mathcal{N})(0 \neq n \Rightarrow 0 \textbf{ div } n = 0)$ * is a valid type expression even though $0 \textbf{ div } n$ is undefined when $0 = n$. This is a matter that we will return to in sections 2 and ??. For now, the $\Rightarrow$-formation rule is unnecessarily complicated for our purposes and we shall assume the following, simpler formation rule.

$$\frac{\begin{array}{l} P \text{ type} \\ Q \text{ type} \end{array}}{P \Rightarrow Q \text{ type}} \qquad \text{(simplified)}\Rightarrow\text{-formation}$$

Example 1.1

$$\begin{array}{l} [\![\ A \in U_1 \\ \triangleright\ A \Rightarrow A \text{ type} \\ ]\!] \end{array}$$

In a context in which A is a small type, $A \Rightarrow A$ is a type.

* The conventional notation for universal quantification has been used here. The notation actually used in these notes is somewhat different.

Derivation

```
0       U1 type
1.0     |[   A ∈ U1
        ▷       % U1-elim. %
1.1             A type
                  % 1.1,1.1, ⇒-form. %
1.2             A ⇒ A type
        ]|
```

(End of example)

In constructive mathematics, a proof of $A \Rightarrow B$ is a method of proving B given a proof of A. Thus $A \Rightarrow B$ is identified with the type $A \rightarrow B$ of (total) functions from the type A into the type B. Assuming that A is a small type, an elementary example would be the proposition $A \Rightarrow A$. (We use the words proposition and type interchangeably here and elsewhere.) A proof of $A \Rightarrow A$ is a method of constructing a proof of A given a proof of A. Such a method would be the identity function on A, $\lambda x.x$, since this is a function that, given an object of A, returns the same object of A.

Example 1.2

```
|[  A ∈ U1
▷ |[  B ∈ U1
  ▷   A ⇒ (B ⇒ A) type
    ]|
]|
```

In a context in which A and B are both small types, $A \Rightarrow (B \Rightarrow A)$ is a type.

Derivation

```
0       U1 type
1.0     |[   A ∈ U1
        ▷       % U1-elim. %
1.1             A type
1.2.0           |[   B ∈ U1
                ▷       % U1-elim. %
1.2.1                   B type
                          % 1.2.1,1.1, ⇒-form. %
1.2.2                   B ⇒ A type
                          % 1.1, 1.2.2, ⇒-form. %
1.2.3                   A ⇒ (B ⇒ A) type
                ]|
        ]|
```

(End of example)

The proposition $A \Rightarrow (B \Rightarrow A)$ provides a second, slightly more complicated, example of the constructive interpretation of implication. Assuming that A and B are small types, a proof of $A \Rightarrow (B \Rightarrow A)$ is a method that, given a proof of A, constructs a proof of $B \Rightarrow A$. Now, a proof of $B \Rightarrow A$ is a method that from a proof of B constructs a proof of A. Thus,

given that x is a proof of A, the constant function $\lambda y.x$ is a proof of $B \Rightarrow A$. Hence the function $\lambda x.\lambda y.x$ is a proof of $A \Rightarrow (B \Rightarrow A)$. (The function $\lambda x.\lambda y.x$ is known as the K-combinator (see e.g. [St,Tu]).)

(b) *Conjunction*

$$\begin{array}{l} P\ type \\ [\!|\ x \in P \\ \triangleright\ Q\ type \\ |\!] \\ \hline P \wedge Q\ type \end{array} \qquad \wedge\text{-formation}$$

Apart from the change of symbol from $\Rightarrow$ to $\wedge$ there is no difference in the formation rules for implication and conjunction. The complicated second premise permits the use of a conditional or non-strict conjunction (sometimes denoted **cand** in programming languages). As before, we shall for the time being assume a simpler formation rule as follows:

$$\begin{array}{l} P\ type \\ Q\ type \\ \hline P \wedge Q\ type \end{array} \qquad \text{(simplified) } \wedge\text{-formation}$$

Example 1.3

$$\begin{array}{l} [\!|\ A \in U_1 \\ \triangleright\ [\!|\ B \in U_1 \\ \quad \triangleright\ A \wedge B \Rightarrow A\ type \\ \quad |\!] \\ |\!] \end{array}$$

In a context in which A and B are small types, $A \wedge B \Rightarrow A$ is a type.

Derivation

$$\begin{array}{lll} 0 & U_1\ type & \\ 1.0 & [\!|\ A \in U_1 & \\ & \triangleright\ \ \%\ U_1\text{-elim. }\% & \\ 1.1 & \quad A\ type & \\ 1.2.0 & \quad [\!|\ B \in U_1 & \\ & \quad \triangleright\ \ \%\ U_1\text{-elim. }\% & \\ 1.2.1 & \quad\quad B\ type & \\ & \quad\quad \%\ 1.1, 1.2.1,\ \wedge\text{-formation }\% & \\ 1.2.2 & \quad\quad A \wedge B\ type & \\ & \quad\quad \%\ 1.1, 1.2.2,\ \Rightarrow\text{-form. }\% & \\ 1.2.3 & \quad\quad A \wedge B \Rightarrow A\ type & \\ & \quad |\!] & \\ & |\!] & \end{array}$$

(End of example)

The proposition $A \wedge B \Rightarrow A$ is true in classical mathematics and in constructive mathematics. The constructive proof goes as follows: Assume that A and B are small types. We have to exhibit a method that given a pair $\langle x, y \rangle$, where x is an object of A and y is an object of B, constructs an object of A. Such a method is clearly the projection function $\mathrm{fst}_{A,B}$ that projects an object of $A \wedge B$ onto its first component.

Another example of a constructive proof involving conjunction is the proof of the proposition $A \wedge B \Rightarrow B \wedge A$. Such a proof is a function that with argument an ordered pair $\langle x, y \rangle$ of objects x, y in A, B, respectively, reverses their order to construct the pair $\langle y, x \rangle$.

(c) Disjunction

$$\frac{P \ \mathit{type} \qquad Q \ \mathit{type}}{P \vee Q \ \mathit{type}} \qquad \vee\text{-formation}$$

A constructive proof of $P \vee Q$ consists of either a proof of P or a proof of Q together with information indicating which of the two has been proved. Thus $P \vee Q$ is identified with the disjoint sum of the types P and Q. That is, objects of $P \vee Q$ take one of the two forms $\mathbf{inl}\ x$ or $\mathbf{inr}\ y$, where x is an object of P, y is an object of Q, and the reserved words **inl** (inject left) and **inr** (inject right) indicate which operand has been proved.

As elementary examples of provable propositions involving disjunction we take $A \Rightarrow A \vee A$, $A \Rightarrow A \vee B$ and $A \vee B \Rightarrow B \vee A$, where A and B are assumed to be small types. There are two possible proofs of $A \Rightarrow A \vee A$. The first, $\lambda x.\mathbf{inl}\ x$, maps an argument x of type A into the left operand of $A \vee A$, and the second, $\lambda x.\mathbf{inr}\ x$, maps an argument x of type A into the right operand of $A \vee A$. In a proof of $A \Rightarrow A \vee B$ there is no choice but to map an object x of A into the left operand of $A \vee B$. Thus the proof takes the form $\lambda x.\mathbf{inl}\ x$. A proof of $A \vee B \Rightarrow B \vee A$ must map an argument w, say, of type $A \vee B$ into $B \vee A$. This is achieved by examining w to discover whether it takes the form $\mathbf{inl}\ x$ or whether it takes the form $\mathbf{inr}\ y$. In the former case w is mapped into $\mathbf{inr}\ x$, in the latter case w is mapped into $\mathbf{inl}\ y$. Thus, in the notation introduced later, a proof of $A \vee B \Rightarrow B \vee A$ is $\lambda w.\mathbf{when}(w, x.\mathbf{inr}\ x, y.\mathbf{inl}\ y)$.

Note that the $\vee$-formation rule does not permit the use of conditional disjunction. The reason is that the conventional meaning of $A\ \mathbf{cor}\ B$ assumes the law of the excluded middle — $A\ \mathbf{cor}\ B$ is true if A is true or A is false and B is true. It is of course feasible in type theory to define $A\ \mathbf{cor}\ B$ for Boolean types A, but this is not a primitive notion.

Exercise 1.0 Prove the following

$$\begin{array}{l} [\![\ A \in U_1 \\ \triangleright\ [\![\ B \in U_1 \\ \quad \triangleright\ A \vee B \Rightarrow B \vee A \ \mathit{type} \\ \quad]\!] \\ ]\!] \end{array}$$

(End of Exercise)

(d) Universal Quantification

$$\frac{\begin{array}{l} P\ type \\ [\![\ x \in P \\ \triangleright\ Q(x)\ type \\ ]\!] \end{array}}{\forall(P, x.Q(x))\ type} \qquad \forall\text{-formation}$$

The $\forall$-formation rule also has two premises, but now the complication in the second premise is of unavoidable importance. The first premise states simply that P is a type. The second premise is that, in a context in which it is assumed that x has type P, it is the case that $Q(x)$ is a type. The conclusion is that $\forall(P, x.Q(x))$ (pronounced "for all P x it is the case that $Q(x)$") is a type.

Typically $Q(x)$ will be an expression containing free occurrences of the variable x. Such occurrences become bound in the expression $\forall(P, x.Q(x))$. Note that the introduction of a binder accompanies the discharge of an assumption. (Compare the $\forall$-formation rule with the $\Rightarrow$-formation rule. In the former the variable x becomes bound by $\forall$, and it is therefore legitimate — and usual — for $Q(x)$ to contain free occurrences of x; in the latter there is no binder for x and hence, to comply with scope rules, Q must *not* contain free occurrences of x.)

We prefer the notation $\forall(P, x.Q(x))$ to the more conventional $(\forall x \in P)Q(x)$ because it makes clear the scope of the binding of the variable x. An expression of the form $x.R$ is called an *abstraction*. $Q(x)$ is called a *family of types dependent on* x.

In order to prove constructively the proposition $\forall(P, x.Q(x))$ it is necessary to provide a method that, given an object p of type P, constructs an object of $Q(p)$. (The notation $Q(p)$ denotes the result obtained by replacing all free occurrences of x in $Q(x)$ by p.) Thus proofs of $\forall(P, x.Q(x))$ are functions (as for implication), their domain being P and their range, $Q(p)$, being dependent on the argument p supplied to the function. The fact that the range depends on the argument is what distinguishes universal quantification from implication.

Example 1.4 $\quad \forall(U_1, A.A \Rightarrow A)\ type$

Derivation

0 $\quad U_1\ type$
% Example 1.1 %
1.0 $\quad [\![\ A \in U_1$
1.1 $\quad \triangleright\ A \Rightarrow A\ type$
$]\!]$
% 0,1, $\forall$-form. %
2 $\quad \forall(U_1, A.A \Rightarrow A)\ type$

(End of example)

According to the semantics of $\forall$ and $\Rightarrow$, a proof of $\forall(U_1, A.A \Rightarrow A)$ is a function that takes a small type A as argument and returns a function mapping A into A. Such a function

is the *polymorphic* identity function $\lambda A.\lambda x.x$. This function maps a small type A into the identity function on A.

Another example of a polymorphic function is the K-combinator $\lambda A.\lambda B.\lambda x.\lambda y.x$ which proves the proposition $\forall(U_1, A.\forall(U_1, B.A \Rightarrow (B \Rightarrow A)))$ — it is a function that takes two types A and B as arguments and returns a proof of $A \Rightarrow (B \Rightarrow A)$, which we recall from section (a) is the function $\lambda x.\lambda y.x$.

(e) Existential Quantification

$$\frac{\begin{array}{l} P\ type \\ \|[\ \ x \in P \\ \triangleright\ \ Q(x)\ type \\ ]\| \end{array}}{\exists(P, x.Q(x))\ type} \qquad \exists\text{-formation}$$

The $\exists$-formation rule is identical to the $\forall$-formation rule in that $\exists$ binds free occurrences of x in the expression $Q(x)$. (The notation we use for existential quantification takes the same form as that for universal quantification rather than the conventional $(\exists x \in P)Q(x)$.)

Example 1.5 $\exists(U_1, A.A)\ type$

Derivation

```
0     U1 type
1.0   ||[  A ∈ U1
      ▷      % U1-elim. %
1.1          A type
      ]||
         % 0,1, ∃-form. %
2     ∃(U1, A.A) type
```

(End of example)

A constructive proof of $\exists(P, x.Q(x))$ consists of exhibiting an object p of P together with a proof of $Q(p)$. Thus proofs of $\exists(P, x.Q(x))$ are pairs $\langle p, q\rangle$ where p is a proof of P and q is a proof of $Q(p)$.

The type $\exists(P, x.Q(x))$ is called a *dependent* type because the second component, q, in a pair $\langle p, q\rangle$ in the type depends on the first component, p (and thus distinguishes it from $P \wedge Q$). This is illustrated by our example. There are many objects of the type $\exists(U_1, A.A)$. Each consists of a pair $\langle A, a\rangle$ where A is a small type and a is an object of that type. (Thus the proposition is interpreted as the statement "there is a small type that is provable", or "there is a small type that is non-empty".) For example, $\langle \mathcal{N}, 0\rangle$ is an object of $\exists(U_1, A.A)$ since $\mathcal{N}$ is a small type (an object of U_1) and 0 is an object of $\mathcal{N}$. Two further examples are $\langle \{red, yellow, blue\}, red\rangle$ and $\langle \mathcal{N} \Rightarrow \mathcal{N}, \lambda x.x\rangle$.

Objects of the type $\exists(U_1, A.A)$ are the simplest possible examples of *algebras* (one or more sets together with a number of operations defined on the sets) since they each consist of a set A together with a single constant of A. A slightly more complicated algebra is specified

by the type $\exists(U_1, A.A \wedge A \Rightarrow A)$. An object of this type consists of a small type A together with a single binary operation defined on A. As yet, however, we cannot specify equational properties of algebras, for example the associativity of multiplication in a semigroup.

(f) Negation

Negation is not a primitive concept of type theory. It is defined via the *empty type*. The empty type, denoted $\emptyset$, is the type containing no elements. Its formation rule is very simple.

$$\frac{}{\emptyset \ type} \qquad \emptyset\text{-formation}$$

Moreover, $\emptyset$ is a small type.

$$\frac{}{\emptyset \in U_1} \qquad U_1\text{-introduction}$$

The negation $\neg P$ is defined to be $P \Rightarrow \emptyset$.

$$\neg P \equiv P \Rightarrow \emptyset$$

($\equiv$ stands for definitionally equal to.) This means that a proof of $\neg P$ is a method for constructing an object of the empty type from an object of P. Since it would be absurd to construct an object of the empty type this is equivalent to saying that it is absurd to construct a proof of P.

As an example of a provable negation, consider the proposition $\neg\forall(U_1, A.A)$. The proposition states that not every small type is provable, or not every small type is non-empty. The basis for its proof is very ordinary — we exhibit a counter-example, namely the empty type $\emptyset$. Formally, we have to construct a function that maps an argument f, say, of type $\forall(U_1, A.A)$ into $\emptyset$. Now f is itself a function mapping objects, A, of U_1 into objects of A. So, for any small type A, the application of f to A, denoted fA, has type A. In particular, $f\emptyset$ has type $\emptyset$. Thus the proof object we require is $\lambda f.f\emptyset$.

(g) Families of Types

An important concept in the theory — that has already been illustrated in several ways — is that of a *family of types*. For example, $A \Rightarrow A$ is a family of types that includes the particular instances $\Phi \Rightarrow \Phi$ and $\mathcal{N} \Rightarrow \mathcal{N}$. Generally we say that R is a family of types indexed by $x \in P$ if the judgement $R\ type$ can be made in a context in which x is assumed to be an object of P.

The rich type-definition mechanism is reflected in the construction of such families of types. Further examples (with hopefully obvious meanings) are

$$\begin{array}{l}\textsf{case } x \textsf{ is} \\ \quad male \rightarrow \{Bill, Joe\} \\ \quad female \rightarrow \{Mary, Jane\}\ )\end{array}$$

indexed by $x \in \{male, female\}$, and

$$\begin{array}{l}\textbf{when } x \textbf{ is}\\ \quad \textbf{inl } a \rightarrow C\\ \quad \textbf{inr } b \rightarrow \{error\}\end{array}$$

indexed by $x \in A \vee B$. The first example has two instances (a) $\{Bill,\ Joe\}$ and (b) $\{Mary,\ Jane\}$. The second example also has two instances (a) C and (b) $\{error\}$. Such dependent type structures have obvious programming applications: we might use the first type structure to specify a database in which the names of males and females are taken from different sets; the second example might be used to specify a function that returns a value of type C when its argument is of one type and returns error when its argument is of another type (eg. the function **div**).

The notion of a family of types is central to many of the inference rules in the theory but, for reasons that do not emerge until section 2 and are not completely resolved until section 3, the examples given in this section are grossly impoverished in this respect. This isn't a major disadvantage in a tutorial presentation but the reader must be prepared to wait for section 2 before the full generality of the rules can be illustrated.

1.3 Introduction and Elimination Rules

For some time now, computing scientists have recognised the need for program structure to reflect data structure. In the words of C.A.R. Hoare [Ho]

> "There are certain close analogies between the methods for structuring data and the methods for structuring a program which processes that data. Thus, a Cartesian product corresponds to a compound statement, which assigns values to its components. Similarly, a discriminated union corresponds to a conditional or case construction, selecting an appropriate processing method for each alternative ..."

This has had an important influence on program design methodologies, a notable example being Jackson's program design method [Ja].

In Martin-Löf's theory the "analogy" between data structure and program structure is made explicit. With each type constructor is associated zero or more so-called *introduction rules* together with a single *elimination rule.* The introduction rules show how to construct the primitive objects of the type; they thus define the data structure. The elimination rule shows how to manipulate or "destruct" objects of the type.

In Martin-Löf's terminology, the introduction rules define *canonical* objects and the elimination rules define *non-canonical* objects. The canonical objects of a type are the primitive objects of the type in the sense that they may be viewed as programs that have themselves as value. The non-canonical objects are not primitive; for each non-canonical object a computation rule is supplied that shows how to evaluate the object.

The terminology "introduction rule" and "elimination rule" is borrowed from Gentzen's account of natural deduction [Ge]. Indeed, the form of Martin-Löf's rules for the propositional connectives follows closely the form of Gentzen's rules. Martin-Löf's contribution has been to add proof objects which summarise the steps used to establish a proposition and which can be given computational meaning. Some familiarity with proofs in Gentzen's system may therefore be helpful to the reader.

In this section we exhibit the introduction and elimination rules for the propositional connectives, and use them to establish a number of familiar tautologies of the (classical) propositional calculus. Verbal descriptions accompanying the examples interpret them both in a logical sense and in a computational sense.

Throughout this section we assume that A, B and C are small types. Formally, we write

$|[\ A, B, C \in U_1$
$\triangleright$

The sclose terminating the scope of these assumptions will be found at the end of the section. The notation $|[\ A, B \in U_1\ \triangleright\ ]|$ is just an ad hoc abbreviation for $|[\ A \in U_1\ \triangleright\ |[\ B \in U_1\ \triangleright\ ]|\ ]|$.

(a) Implication

Our first rule is the introduction rule for implication.

$$\frac{\begin{array}{l}\langle\text{premises of } \Rightarrow\text{-formation}\rangle \\ |[\ x \in P \\ \triangleright\ q(x) \in Q \\ ]|\end{array}}{\lambda x.q(x) \in P \Rightarrow Q} \quad \Rightarrow\text{-introduction}$$

In a logical sense the rule may be read as "if assuming that x is a proof of P it is possible to construct a proof of Q then $\lambda x.q(x)$ is a proof of $P \Rightarrow Q$". In a computational sense the rule is read differently. "If in a context in which x is an object of type P the object $q(x)$ has type Q then the function $\lambda x.q(x)$ is an object of type $P \Rightarrow Q$".

In general, $q(x)$ will be an expression containing zero or more free occurrences of x. Such occurrences of x become bound in the expression $\lambda x.q(x)$. As we remarked earlier, binding of variables is always associated with the discharge of assumptions.

Note Martin-Löf omits several premises in his presentation of the inference rules and this has led to several misunderstandings and errors. In particular, in a rule with inference of the form $p \in P$ he omits premises of the form P *type*. Other authors have tried to remedy this by specifying a minimal set of additional premises. I prefer the more systematic approach of simply prefacing each rule with a reminder of the missing premises. Thus, it can be argued that the premises of $\Rightarrow$-formation are all superfluous in the above rule; however, such premises are essential to the correct application of later rules.

Example 1.6 $\lambda x.x \in A \Rightarrow A$

This is a continuation of example 1.1. We now show formally how to derive the judgement that the identity function is a proof of the proposition $A \Rightarrow A$.

Derivation

0.0 |[$x \in A$
▷ % 0.0 %
0.1 $x \in A$
]|
% Example 1.1,0, $\Rightarrow$-intro. %
1 $\lambda x.x \in A \Rightarrow A$

Note that the numeral 0 in the explanation of step 1 stands for the combination of the assumption 0.0 and the conclusion 0.1. Example 1.1 is quoted in the explanation of step 1 because it includes the required premises of the form P *type*.

(End of example)

Example 1.7 $\lambda x.\lambda y.x \in A \Rightarrow (B \Rightarrow A)$

Derivation

```
0.0     |[  x ∈ A
0.1.0   ▷ |[  y ∈ B
          ▷     % 0.0 %
0.1.1         x ∈ A
          ]|
            % Example 1.2, 0.1, ⇒-intro. %
0.2       λy.x ∈ B ⇒ A
        ]|
          % Example 1.2, 0, ⇒-intro. %
1       λx.λy.x ∈ A ⇒ (B ⇒ A)
```

(End of example)

The ⇒-elimination rule is easily recognisable as the rule of modus ponens.

$$\frac{\begin{array}{l}\langle\text{premises of } \Rightarrow\text{-formation}\rangle \\ p \in P \\ r \in P \Rightarrow Q\end{array}}{rp \in Q} \quad \Rightarrow\text{-elimination}$$

In a logical sense the rule states that if p is a proof of P and r is a proof of $P \Rightarrow Q$ then rp is a proof of Q. In a computational sense it states that if p has type P and r is a function from P to Q then rp, the result of applying r to p, has type Q.

The notation we are using for λ-expressions and function application is the conventional one [St]. That is, function application is denoted by juxtaposition and associates to the left, and we assume that when a λ-term such as $\lambda x.q$ occurs in a larger expression q is taken as extending as far as possible — to the first unmatched closing bracket or the end of the expression, whichever is first. Corresponding to the convention that function application associates to the left we have the convention that implication associates to the right. Thus $P \Rightarrow Q \Rightarrow R$ is read as $P \Rightarrow (Q \Rightarrow R)$. Occasionally, but not always, we use this convention to reduce the number of parentheses. However, we prefer to retain the parentheses when it makes the import of the examples and exercises clearer.

Example 1.8 $\quad \lambda f.\lambda g.\lambda x.g(fx) \in (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$

The example asserts that implication is transitive. I.e. given a proof f of $A \Rightarrow B$, and given a proof g of $B \Rightarrow C$, the object $\lambda x.g(fx)$ is a proof of $A \Rightarrow C$.

Derivation

In the following derivation we omit judgements such as $A \Rightarrow B$ *type*, $B \Rightarrow C$ *type*. Strictly, these are necessary to the use of the assumption rule and the ⇒-introduction rule but their inclusion would obscure the main details.

```
0.0      ||[  f ∈ A ⇒ B
0.1.0    ▷  ||[  g ∈ B ⇒ C
0.1.1.0     ▷  ||[  x ∈ A
                ▷     % 0.0, 0.1.1.0, ⇒-elim. %
0.1.1.1            fx ∈ B
                      % 0.1.0, 0.1.1.1, ⇒-elim. %
0.1.1.2            g(fx) ∈ C
                ]||
                   % 0.1.1 ⇒-intro. %
0.1.2           λx.g(fx) ∈ A ⇒ C
             ]||
                % 0.1, ⇒-intro. %
0.2          λg.λx.g(fx) ∈ (B ⇒ C) ⇒ (A ⇒ C)
          ]||
             % 0 ⇒-intro. %
1         λf.λg.λx.g(fx) ∈ (A ⇒ B) ⇒ (B ⇒ C) ⇒ (A ⇒ C)
```

(End of example)

Exercise 1.1 Verify the following.

(a) $\lambda f.\lambda g.\lambda x.fx(gx) \in [A \Rightarrow (B \Rightarrow C)] \Rightarrow [(A \Rightarrow B) \Rightarrow (A \Rightarrow C)]$

(b) $\lambda f.\lambda x.\lambda y.f(\lambda z.y)x \in [(A \Rightarrow B) \Rightarrow (A \Rightarrow C)] \Rightarrow [A \Rightarrow (B \Rightarrow C)]$

(Equivalence is defined as the conjunction of two implications. Exercise 1.1 is therefore a proof that implication distributes over implication. I.e.

$[A \Rightarrow (B \Rightarrow C)] \Longleftrightarrow [(A \Rightarrow B) \Rightarrow (A \Rightarrow C)]$.)

Exercise 1.2 Prove, and construct proof objects for, the propositions:

(a) $(A \Rightarrow B \Rightarrow C) \Rightarrow (B \Rightarrow A \Rightarrow C)$

(b) $(B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$

(c) $[((A \Rightarrow B) \Rightarrow B) \Rightarrow B] \Rightarrow [A \Rightarrow B]$

(d) $A \Rightarrow ((A \Rightarrow B) \Rightarrow B)$

(End of exercise)

Note: Example 1.7, exercise 1.1(a) and exercises 1.2(a) and (b) are motivated by the K, S, B and C combinators [Tu]. See later for more general forms of S and K. Exercises (c) and (d) are discussed again in section 1.3(f).

(b) Conjunction

The semantics of conjunction in constructive mathematics identifies $P \wedge Q$ with the Cartesian product $P \times Q$. That is, a proof of $P \wedge Q$ is an ordered pair $\langle p, q \rangle$ where p is a proof of P and q is a proof of Q. The introduction rule is therefore very straightforward.

$$\frac{\langle \text{premises of } \wedge\text{-formation} \rangle \quad p \in P \quad q \in Q}{\langle p, q \rangle \in P \wedge Q} \qquad \wedge\text{-introduction}$$

Example 1.9 $\lambda x.\langle x, x\rangle \in A \Rightarrow A \wedge A$

A proof of $A \wedge A$ can be constructed from a proof of A simply by repeating the proof object. (Note that conjunction takes precedence over implication in the expression $A \Rightarrow A \wedge A$.)

Derivation

0.0 |[$x \in A$
▷ % 0.0, 0.0, $\wedge$-intro %
0.1 $\langle x, x\rangle \in A \wedge A$
]|
% 0, $\Rightarrow$-intro %
1 $\lambda x.\langle x, x\rangle \in A \Rightarrow A \wedge A$

(End of example)

Example 1.10 (Currying)

$\lambda f.\lambda x.\lambda y.f\langle x, y\rangle \in (A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$

This is one of my favourite examples of the principle of propositions-as-types. In a logical sense the example is very familiar to all English speakers since it is legitimate to say either "if A *and* if B then C" or "if A *then* if B then C". In a computational sense it also expresses a very familiar rule, namely that any function f of a pair of arguments $\langle x, y\rangle$ can be "curried" (after Haskel B. Curry), i.e. converted into a function that maps the single argument x into a function that maps the second argument y into the result of applying f to the pair $\langle x, y\rangle$.

Derivation

0.0 |[$f \in A \wedge B \Rightarrow C$
0.1.0 ▷ |[$x \in A$
0.1.1.0 ▷ |[$y \in B$
▷ % 0.1.0, 0.1.1.0, $\wedge$-intro. %
0.1.1.1 $\langle x, y\rangle \in A \wedge B$
% 0.0, 0.1.1.1, $\Rightarrow$-elim. %
0.1.1.2 $f\langle x, y\rangle \in C$
]|
% 0.1.1, $\Rightarrow$-intro. %
0.1.2 $\lambda y.f\langle x, y\rangle \in B \Rightarrow C$
]|
% 0.1, $\Rightarrow$-intro. %
0.2 $\lambda x.\lambda y.f\langle x, y\rangle \in A \Rightarrow B \Rightarrow C$
]|
% 0, $\Rightarrow$-intro. %
1 $\lambda f.\lambda x.\lambda y.f\langle x, y\rangle \in (A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$

(End of example)

Many of the propositions we prove have implication as the principal connective. Correspondingly, derivations often end with at least one application of the $\Rightarrow$-introduction rule. We hope that by now the reader is familiar with this rule and we shall omit it, except in a

few places. We shall also compact several consecutive assumptions into one. As illustration the following is an abbreviation of the last derivation.

Abbreviated Derivation

0.0 |[$f \in A \wedge B \Rightarrow C; x \in A; y \in B$
▷ % 0.0, $\wedge$-intro. %
0.1 $\langle x, y \rangle \in A \wedge B$
% 0.0, 0.1, $\Rightarrow$-elim. %
0.2 $f\langle x, y \rangle \in C$
]|

End of abbreviated derivation

The latter derivation may be summarised as the statement that in a context in which f has type $A \wedge B \Rightarrow C$, x has type A and y has type B, it is the case that $f\langle x, y \rangle$ has type C. This is semantically equivalent to the statement that $\lambda f.\lambda x.\lambda y.f\langle x, y \rangle$ has type $(A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$.

In Gentzen's system there are two elimination rules for conjunction. The first states that it is possible to infer A from $A \wedge B$, the second states that it is possible to infer B from $A \wedge B$. Martin-Löf has combined these two rules into one, which takes the following form:

$$\begin{array}{l} \langle \text{premises of } \wedge\text{-formation} \rangle \\ |[\ w \in P \wedge Q \\ \triangleright\ R(w)\ \mathit{type} \\ ]| \\ r \in P \wedge Q \\ |[\ x \in P; y \in Q \\ \triangleright\ s(x, y) \in R(\langle x, y \rangle) \\ ]| \\ \hline \\ \mathrm{split}(r, (x, y).s(x, y)) \in R(r) \end{array} \qquad \wedge\text{-elimination}$$

The first premise states that $R(w)$ is a type in a context in which w is an object of $P \wedge Q$. Typically, therefore, $R(w)$ is a family of types, one for each such object. More precisely, $R(w)$ is a type expression containing an arbitrary number of occurrences of the dummy variable w such that $R(r)$ is a type whenever r is an object of $P \wedge Q$. Given this premise, the rule is used to prove $R(r)$ as follows. First, establish that r has type $P \wedge Q$. Second, assume that x and y are individual objects of P and Q, respectively, and construct an object $s(x, y)$ of type $R(\langle x, y \rangle)$. Then the exhibited **split** construct is a proof of $R(r)$. Computationally the **split** construct has the effect of splitting a pair, r, into its components, the value of the expression $s(x, y)$ is then computed with the components being bound to the variables x and y occurring free in the expression.

The $\wedge$-elimination rule is one of those that, as yet, we can only illustrate in a limited way. Thus, although the rule permits the type R to depend on the object r, none of the examples in this section illustrate such a dependence.

Example 1.11 $\lambda w.\mathbf{split}(w,(x,y).x) \in A \wedge B \Rightarrow A$

The function, *fst*, which projects a pair onto its first component is the function that splits its argument w into the pair $\langle x,y\rangle$ and returns x.

Derivation

```
0.0     |[  w ∈ A ∧ B
0.1.0   ▷   |[  x ∈ A; y ∈ B
            ▷     % 0.1.0 %
0.1.1           x ∈ A
            ]|
              % 0.0, 0.1, ∧-elim. %
0.2         split(w, (x, y).x) ∈ A
        ]|
```

The function, *snd*, which projects a pair onto its second component is obtained similarly.

(End of example)

Example 1.12 (Uncurrying)

$\lambda f.\lambda w.\mathbf{split}(w,(x,y).fxy) \in (A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)$

Derivation

```
0.0       |[  f ∈ A ⇒ B ⇒ C
0.1.0     ▷   |[  w ∈ A ∧ B
0.1.1.0       ▷   |[  x ∈ A; y ∈ B
                  ▷     % 0.0, 0.1.1.0, ⇒-elim. %
0.1.1.1             fx ∈ B ⇒ C
                      % 0.1.1.0, 0.1.1.1, ⇒-elim. %
0.1.1.2             fxy ∈ C
                  ]|
                    % 0.1.0, 0.1.1, ∧-elim. %
0.1.2             split(w, (x, y).fxy) ∈ C
              ]|
          ]|
```

(End of example)

Exercise 1.3 Verify the following:

(a) $\lambda w.\mathbf{split}(w,(x,y).\langle y,x\rangle) \in A \wedge B \Rightarrow B \wedge A$

(b) $\lambda f.\langle \lambda x.fst(fx), \lambda x.snd(fx)\rangle \in (A \Rightarrow B \wedge C) \Rightarrow (A \Rightarrow B) \wedge (A \Rightarrow C)$

(c) $\lambda w.\mathbf{split}(w,(f,g).\lambda x.\langle fx, gx\rangle) \in (A \Rightarrow B) \wedge (A \Rightarrow C) \Rightarrow (A \Rightarrow B \wedge C)$

(End of exercise)

Exercise 1.4 Parts (b) and (c) of the last exercise can be interpreted as the claim that implication distributes over conjunction; part (a) establishes that conjunction is symmetric. Establish as many other algebraic properties of conjunction and implication as you can.

(End of exercise)

(c) Disjunction

A proof of $P \vee Q$ is a proof of P or a proof of Q together with information as to which has been proved. Thus, $P \vee Q$ is identified with the disjoint sum of P and Q and has two introduction rules.

$$\frac{\langle\text{premises of } \vee\text{-formation}\rangle \quad p \in P}{\mathbf{inl}\ p \in P \vee Q} \qquad \frac{\langle\text{premises of } \vee\text{-formation}\rangle \quad q \in Q}{\mathbf{inr}\ q \in P \vee Q}\ \vee\text{-introduction}$$

The constants **inl** and **inr** are called *injection functions* and stand for inject left and inject right, respectively.

Example 1.13 $\lambda x.\mathbf{inl}\ x \in A \Rightarrow A \vee A$

There are two functions of type $A \Rightarrow A \vee A$ depending on whether the argument x is injected into the first or second operand of $A \vee A$. Our derivation takes the first option.

Derivation

```
0.0    |[  x ∈ A
       ▷      % 0.0, ∨-intro. %
0.1        inl x ∈ A ∨ A
       ]|
```

(End of example)

Example 1.14 $\lambda f.\lambda x.f(\mathbf{inl}\ x) \in (A \vee B \Rightarrow C) \Rightarrow (A \Rightarrow C)$

Given a function mapping elements of the disjoint union $A \vee B$ into C it is possible to construct a function mapping A into C. Alternatively, if either A or B implies C then A implies C.

Derivation

```
0.0      |[  f ∈ A ∨ B ⇒ C
0.1.0    ▷  |[  x ∈ A
            ▷      % 0.1.0, ∨-intro. %
0.1.1           inl x ∈ A ∨ B
                  % 0.0, 0.1.1, ⇒-elim. %
0.1.2           f(inl x) ∈ C
            ]|
         ]|
```

(End of example)

Exercise 1.5 Verify the following :

$$\lambda f.f(\mathbf{inr}(\lambda x.f(\mathbf{inl}\ x))) \in [(A \vee (A \Rightarrow B)) \Rightarrow B] \Rightarrow B$$

(End of exercise)

Example 1.14 and exercise 1.5 are motivated by properties of negation that are considered later in section 1.3(f).

The ∨-elimination rule corresponds to case analysis as in the following argument. Suppose it has been established that just two cases P and Q need to be considered and suppose it is possible to prove R both in the case P and in the case Q. Then R is true in general.

$$\frac{\begin{array}{l}\langle\text{premises of } \vee\text{-formation}\rangle \\ [\!| \; w \in P \vee Q \\ \triangleright \; R(w) \; type \\ |\!] \\ r \in P \vee Q \\ [\!| \; x \in P \\ \triangleright \; s(x) \in R(\mathbf{inl}\; x) \\ |\!] \\ [\!| \; y \in Q \\ \triangleright \; t(y) \in R(\mathbf{inr}\; y) \\ |\!]\end{array}}{\mathbf{when}(r, x.s(x), y.t(y)) \in R(r)} \quad \vee\text{-elimination}$$

The ∨-elimination rule is used to prove a proposition R by a case analysis on P or Q. Although we are unable to illustrate the full generality of the rule at this stage, the first premise indicates that, typically, R is a family of types indexed by the (dummy) variable w of type $P \vee Q$. We prove R at r ("$R(r)$" in the inference) by showing that R is provable at $\mathbf{inl}\; x$ whenever x is an object of P and at $\mathbf{inr}\; y$ whenever y is an object of Q (cf. the last two premises).

Note that the rule discharges two assumptions. Thus within the **when** statement the variable x is bound in $s(x)$ and the variable y is bound in $t(y)$.

A **when** statement is similar to an **if** statement in conventional languages. Computationally the rule states that if $s(x)$ constructs an object of $R(\mathbf{inl}\; x)$ when supplied with an argument x of type P and if $t(y)$ constructs an object of $R(\mathbf{inr}\; y)$ when supplied with an argument y of type Q then it is always possible to construct an object of $R(r)$ from an object r of $P \vee Q$. To do so, examine the form of r and if it is $\mathbf{inl}\; p$ bind x to p and apply $s(x)$, if it is $\mathbf{inr}\; q$ bind y to q and apply $t(y)$.

Example 1.15

$$\lambda d.\mathbf{when}(d, x.x, x.x) \in A \vee A \Rightarrow A$$

An object x of A that is injected into the disjoint sum $A \vee A$ must either be injected into the left or into the right operand. The exhibited function performs the reverse process.

Derivation

0.0 |[$w \in A \vee A$
0.1.0 ▷ |[$x \in A$
0.1.1 ▷ $x \in A$
]|
% 0.0, 0.1, 0.1, ∨-elim. %
0.2 $\mathbf{when}(w, x.x, x.x) \in A$
]|

(End of example)

Example 1.16

$$\lambda w.\mathbf{when}(w, f.\lambda x.f(fst(x)), g.\lambda x.g(snd(x))) \in [(A \Rightarrow C) \vee (B \Rightarrow C)] \Rightarrow [(A \wedge B) \Rightarrow C]$$

Derivation

0.0 |[$w \in (A \Rightarrow C) \vee (B \Rightarrow C)$
0.1.0 ▷ |[$f \in A \Rightarrow C$
0.1.1.0 ▷ |[$x \in A \wedge B$
▷ % 0.1.1.0, Example 1.11 %
0.1.1.1 $fst(x) \in A$
% 0.1.0, 0.1.1.1, ⇒-elim. %
0.1.1.2 $f(fst(x)) \in C$
]|
% 0.1.1, ⇒-intro. %
0.1.2 $\lambda x.f(fst(x)) \in A \wedge B \Rightarrow C$
]|
0.2.0 |[$g \in B \Rightarrow C$
▷ % similarly %
0.2.1 $\lambda x.g(snd(x)) \in A \wedge B \Rightarrow C$
]|
% 0, 0.1, 0.2, ∨-elim. %
0.3 $\mathbf{when}(w, f.\lambda x.f(fst(x)), g.\lambda x.g(snd(x))) \in A \wedge B \Rightarrow C$
]|

(End of example)

Exercise 1.6 Prove the following:

(a) $A \vee B \Rightarrow B \vee A$
(b) $A \vee (B \vee C) \Rightarrow (A \vee B) \vee C$
(c) $A \vee (B \wedge C) \Rightarrow [(A \vee B) \wedge (A \vee C)]$
(d) $A \vee (B \Rightarrow C) \Rightarrow [(A \vee B) \Rightarrow (A \vee C)]$
(e) $[(A \Rightarrow B) \vee (A \Rightarrow C)] \Rightarrow [A \Rightarrow (B \vee C)]$
(f) $[(A \Rightarrow C) \wedge (B \Rightarrow C)] \Rightarrow [(A \vee B) \Rightarrow C]$

(End of exercise)

Exercise 1.7 Conjecture some more algebraic properties of $\wedge$, $\vee$ and $\Rightarrow$ and try to prove them.

(End of exercise)

(d) Universal Quantification

We recall from our informal account of the semantics of universal quantification that, like an implication, the type $\forall(P, x.Q(x))$ is identified with a function type but that the type of the range, $Q(x)$, may depend on the argument, x. In Martin-Löf's terminology, $Q(x)$ is a family of types, one for each object, x, in A. That is, typically $Q(x)$ will be a type expression containing zero or more free occurrences of x. (Of course such free occurrences become bound in the expression $\forall(P, x.Q(x))$.)

We can infer $\forall(P, x.Q(x))$ if it is always possible to establish Q in a context in which it is assumed that x has type P. This is the $\forall$-introduction rule.

⟨premises of $\forall$-formation⟩
|[$x \in P$
▷ $q(x) \in Q(x)$
]|
———————————— $\forall$-introduction
$\lambda x.q(x) \in \forall(P, x.Q(x))$

Two binders are introduced when the assumption $x \in P$ is discharged. The lambda, λ, binds the first x in $q(x)$, and, as already mentioned, the universal quantification binds the second x in $Q(x)$.

Example 1.17 $\lambda A.\lambda x.x \in \forall(U_1, A.A \Rightarrow A)$

$A \Rightarrow A$ is a family of types, one for each small type A. The function $\lambda A.\lambda x.x$ is the polymorphic identity function.

Derivation

0.0 |[$A \in U_1$
▷ % Example 1.6 %
0.1 $\lambda x.x \in A \Rightarrow A$
]|
% Example 1.4,0, $\forall$-intro. %
1 $\lambda A.\lambda x.x \in \forall(U_1, A.A \Rightarrow A)$

(End of example)

Many examples similar to example 1.17 can now be quoted simply by discharging the assumptions $A, B, C \in U_1$ introduced at the beginning of this section. We give just a few.

Example 1.18

(a) $\lambda A.\lambda B.\lambda x.\mathbf{inl}\ x \in \forall(U_1, A.\forall(U_1, B.A \Rightarrow A \vee B))$

(b) $\lambda A.\lambda B.\lambda w.\mathbf{split}(w, (x, y).x) \in \forall(U_1, A.\forall(U_1, B.A \wedge B \Rightarrow A))$

(c) $\lambda A.\lambda B.\lambda C.\lambda f.\lambda x.f(\mathbf{inl}\ x) \in \forall(U_1, A.\forall(U_1, B.\forall(U_1, C.(A \vee B \Rightarrow C) \Rightarrow A \Rightarrow C)))$

(End of example)

The $\forall$-elimination rule is almost identical to the $\Rightarrow$-elimination rule.

$$\frac{\begin{array}{l}\text{(premises of } \forall\text{-formation)}\\ r \in \forall(P, x.Q(x))\\ p \in P\end{array}}{rp \in Q(p)} \qquad \forall\text{-elimination}$$

If r is a function that given an arbitrary argument x of type P constructs a proof of Q and if p has type P then rp (the result of applying r to p) has type (or proves) $Q(p)$.

Several examples of the properties of universal quantification may be obtained by a straightforward generalisation of properties of implication. For instance:

Example 1.19

$$\begin{array}{ll}[\![& F, G \in A \Rightarrow U_1\\ \triangleright & \lambda f.\lambda g.\lambda z.fz(gz) \in \forall(A, x.Fx \Rightarrow Gx) \Rightarrow \forall(A, y.Fy) \Rightarrow \forall(A, z.Gz)\\ ]\!] & \end{array}$$

This example generalises exercise 1.1(a). (To obtain the latter replace occurrences of "Fx" and "Fy" by "B", "Gx" and "Gz" by "C", and "$\forall(A, x.)$", "$\forall(A, y.)$", "$\forall(A, z.)$" by "$A \Rightarrow$".) We begin by establishing that a number of type expressions are well-formed. Note that judgements 0.2 and 0.4 are prerequisites for the assumptions 0.7.1.0 and 0.7.0, respectively. Judgements 0.3, 0.5 and 0.6 are necessary when using $\Rightarrow$-introduction to discharge the assumptions 0.7.1.1.0, 0.7.1.0 and 0.7.0, respectively.

Derivation

0.0 ||[$F, G \in A \Rightarrow U_1$
0.1.0 ▷ ||[$y \in A$
▷ % 0.0, 0.1.0, $\Rightarrow$-elim. %
0.1.1 $Fy \in U_1$
% 0.1.1, U_1-elim. %
0.1.2 Fy *type*
]||
% A *type*, 0.1, $\forall$-form. %
0.2 $\forall(A, y.Fy)$ *type*
% similarly %
0.3 $\forall(A, z.Gz)$ *type*
% exercise %
0.4 $\forall(A, x.Fx \Rightarrow Gx)$ *type*
% 0.2, 0.3, $\Rightarrow$-form. %
0.5 $\forall(A, y.Fy) \Rightarrow \forall(A, z.Gz)$ *type*
% 0.4, 0.5, $\Rightarrow$-form. %
0.6 $\forall(A, x.Fx \Rightarrow Gx) \Rightarrow \forall(A, y.Fy) \Rightarrow \forall(A, z.Gz)$ *type*
0.7.0 ||[$f \in \forall(A, x.Fx \Rightarrow Gx)$
0.7.1.0 ▷ ||[$g \in \forall(A, y.Fy)$
0.7.1.1.0 ▷ ||[$z \in A$
▷ % 0.7.1.0, 0.7.1.1.0, $\forall$-elim. %
0.7.1.1.1 $gz \in Fz$
% 0.7.0, 0.7.1.1.0, $\forall$-elim. %
0.7.1.1.2 $fz \in Fz \Rightarrow Gz$
% 0.7.1.1.1, 0.7.1.1.2, $\Rightarrow$-elim. %
0.7.1.1.3 $fz(gz) \in Gz$
]||
]||
]||
]||

In this example we have been careful to use distinct bound variables so as to illustrate the process of substitution in $\forall$-elimination. (See steps 0.7.1.1.1 and 0.7.1.1.2.) We shall not be so careful in the future.

(End of example)

A second example will illustrate further the process of generalising implication to universal quantifications. Note particularly in this example the importance of well-formedness of type expressions.

Example 1.20

$$\begin{array}{l} ||[\ F \in A \vee B \Rightarrow U_1 \\ \triangleright\ \lambda f.\lambda x.f(\mathrm{inl}\ x) \in \forall(A \vee B, w.Fw) \Rightarrow \forall(A, x.F(\mathrm{inl}\ x)) \\ ]|| \end{array}$$

In a context in which F is a family of small types, one element of the family for each object of $A \vee B$, the function $\lambda f.\lambda x.f(\mathrm{inl}\ x)$ is a proof of $\forall(A \vee B, w.Fw) \Rightarrow \forall(A, x.F(\mathrm{inl}\ x))$. This result generalises the proof of the proposition $(A \vee B \Rightarrow C) \Rightarrow A \Rightarrow C$ in example 1.14. The well-formedness of the type expressions is a non-trivial part of the derivation and so

we begin by establishing a number of judgements of the form P *type*. Note that judgement 0.2 is a prerequisite to assumption 0.5.0, and judgement 0.4 is a prerequisite to the use of $\forall$-introduction in step 0.6.

Derivation

0.0 ||[$F \in A \vee B \Rightarrow U_1$
0.1.0 ▷ ||[$w \in A \vee B$
▷ % 0.0, 0.1.0, $\Rightarrow$-elim. %
0.1.1 $Fw \in U_1$
% 0.1.1, U_1-elim. %
0.1.2 Fw *type*
]||
% $A \vee B$ *type*, 0.1, $\forall$-form. %
0.2 $\forall(A \vee B, w.Fw)$ *type*
0.3.0 ||[$x \in A$
▷ % 0.3.0, $\vee$-intro. %
0.3.1 $\mathbf{inl}\ x \in A \vee B$
% 0.0, 0.3.1, $\Rightarrow$-elim., U_1-elim. %
0.3.2 $F(\mathbf{inl}\ x)$ *type*
]||
% A *type*, 0.3, $\forall$-form. %
0.4 $\forall(A, x.F(\mathbf{inl}\ x))$ *type*
0.5.0 ||[$f \in \forall(A \vee B, w.Fw)$
0.5.1.0 ▷ ||[$x \in A$
▷ % 0.5.1.0, $\vee$-intro. %
0.5.1.1 $\mathbf{inl}\ x \in A \vee B$
% 0.5.0, 0.5.1.1, $\forall$-elim. %
0.5.1.2 $f(\mathbf{inl}\ x) \in F(\mathbf{inl}\ x)$
]||
% 0.4, 0.5.1, $\forall$-intro. %
0.5.2 $\lambda x.f(\mathbf{inl}\ x) \in \forall(A, x.F(\mathbf{inl}\ x))$
]||
% 0.2,0.4, 0.5.0, $\Rightarrow$-intro. %
0.6 $\lambda f.\lambda x.f(\mathbf{inl}\ x) \in \forall(A \vee B, w.Fw) \Rightarrow \forall(A, x.F(\mathbf{inl}\ x))$
]||

(End of example)

From now on we shall omit the verification of the first set of premises (those relating to the formation rule), except in some cases where it is non-trivial or is especially relevant.

Exercise 1.8 Construct proof objects for the following.

(a) (cf. example 1.10)
||[$H \in A \wedge B \Rightarrow U_1$
▷ $\forall(A \wedge B, w.Hw) \Rightarrow \forall(A, x.\forall(B, y.H\langle x, y\rangle))$
]||

(b) (cf. example 1.12)
||[$H \in A \wedge B \Rightarrow U_1$
▷ $\forall(A, x.\forall(B, y.H\langle x, y\rangle)) \Rightarrow \forall(A \wedge B, w.Hw)$
]||

(End of exercise)

(e) *Existential Quantification*

In the same way that universal quantification generalises implication, existential quantification generalises conjunction. To prove $\exists(P, x.Q(x))$ it is necessary to exhibit an object p of P and an object q of $Q(p)$. Thus the proof object is a pair $\langle p, q\rangle$ where now the second component may depend on the first.

$$\frac{\begin{array}{l}\langle\text{premises of } \exists\text{-formation}\rangle\\ p \in P\\ q \in Q(p)\end{array}}{\langle p, q\rangle \in \exists(P, x.Q(x))} \quad \exists\text{-introduction}$$

Example 1.21 $\langle \mathcal{N}, 0\rangle \in \exists(U_1, A.A)$

This example illustrates the dependence of the second component on the first. (The inspiration for it came from [BL].)

Derivation

0 $\mathcal{N} \in U_1$
1 $0 \in \mathcal{N}$
% 0,1, $\exists$-intro %
2 $\langle \mathcal{N}, 0\rangle \in \exists(U_1, A.A)$

(End of example)

Example 1.22

$$\begin{array}{l}|[\ \ F \in A \Rightarrow U_1\\ \triangleright\ \ \lambda f.\lambda x.\lambda y.f\langle x, y\rangle \in (\exists(A, x.Fx) \Rightarrow C) \Rightarrow \forall(A, x.Fx \Rightarrow C)\\ ]|\end{array}$$

This generalisation of example 1.10 is obtained by substituting "$\exists(A, x.Fx)$" for "$A \wedge B$" and "$\forall(A, x.Fx$" for "$A \Rightarrow (B$".

Derivation

0.0 |[$F \in A \Rightarrow U_1$
0.1.0 $\triangleright$ |[$f \in \exists(A, x.Fx) \Rightarrow C$
0.1.1.0 $\triangleright$ |[$x \in A$
0.1.1.1.0 $\triangleright$ |[$y \in Fx$
$\triangleright$ % 0.1.1.0, 0.1.1.1.0, $\exists$-intro. %
0.1.1.1.1 $\langle x, y\rangle \in \exists(A, x.Fx)$
% 0.1.0, 0.1.1.1.1, $\Rightarrow$-elim. %
0.1.1.1.2 $f\langle x, y\rangle \in C$
]|
]|
]|
]|

Following the convention introduced earlier with regard to the omission of steps involving the use of $\Rightarrow$-elimination, we also omit final steps in a derivation that involve the use of $\forall$-introduction.

(End of example)

Exercise 1.9

Fill in all the details in the last derivation. (I.e. include proofs of judgements of the form P *type*.)

(End of exercise)

The following exercise is a further generalisation of Example 1.10.

Exercise 1.10 Prove the following :

$$\begin{array}{l} [\![\ F \in A \Rightarrow U_1\ ;\ G \in \exists(A, x.Fx) \Rightarrow U_1 \\ \triangleright\ \lambda f.\lambda x.\lambda y.f\langle x, y\rangle \in \forall(\exists(A, x.Fx), w.Gw) \Rightarrow \forall(A, x.\forall(Fx, y.G\langle x, y\rangle)) \\ ]\!] \end{array}$$

(End of exercise)

The elimination rule for existential quantification enables one to assume that a proof r of $\exists(P, x.Q(x))$ can be split into two components.

$$\frac{\begin{array}{l} \langle\text{premises of } \exists\text{-formation}\rangle \\ [\![\ w \in \exists(P, x.Q(x)) \\ \triangleright\ R(w)\ \textit{type} \\ ]\!] \\ r \in \exists(P, x.Q(x)) \\ [\![\ y \in P;\ \ z \in Q(y) \\ \triangleright\ s(y, z) \in R(\langle y, z\rangle) \\ ]\!] \end{array}}{\mathbf{split}(r, (y, z).s(y, z)) \in R(r)} \quad \exists\text{-elimination}$$

The first premise states that $R(w)$ is a type in a context in which w is a proof of $\exists(P, x.Q(x))$. Typically, therefore, $R(w)$ is a family of types, one for each object, w, in the existential quantification. Given this premise, one proves $R(r)$ by first establishing that r proves $\exists(P, x.Q(x))$ and, second, establishing $R(\langle y, z\rangle)$ whenever y is an object of P and z is an object of $Q(y)$.

Example 1.23

$$\begin{array}{l} [\![\ F \in A \vee B \Rightarrow U_1 \\ \triangleright\ \exists(A \vee B, x.Fx) \Rightarrow [\exists(A, a.F(\mathbf{inl}\ a)) \vee \exists(B, b.F(\mathbf{inr}\ b))] \\ ]\!] \end{array}$$

Derivation

```
                % conc abbreviates ∃(A, a.F(inl a)) ∨ ∃(B, b.F(inr b)) %
0.0         |[  p ∈ ∃(A ∨ B, x.Fx)
0.1.0       ▷   |[  x ∈ A ∨ B; f ∈ Fx
0.1.1.0         ▷   |[  a ∈ A
0.1.1.1.0           ▷   |[  g ∈ F(inl a)
                        ▷     % 0.1.1.0, 0.1.1.1.0, ∃-intro, ∨-intro %
0.1.1.1.1               inl ⟨a, g⟩ ∈ conc
                        ]|
                      % ⇒-intro %
0.1.1.2               λg.inl ⟨a, g⟩ ∈ F(inl a) ⇒ conc
                    ]|
                  % similarly %
0.1.2.0           |[  b ∈ B
0.1.2.1           ▷   λg.inr ⟨b, g⟩ ∈ F(inr b) ⇒ conc
                  ]|
                  % 0.1.0, 0.1.1, 0.1.2, ∨-elimination %
0.1.3             when(x, a.λg.inl ⟨a, g⟩, b.λg.inr ⟨b, g⟩) ∈ Fx ⇒ conc
                  % 0.1.0, 0.1.3, ⇒-elimination %
0.1.4             when(x, a.λg.inl ⟨a, g⟩, b.λg.inr ⟨b, g⟩)f ∈ conc
                ]|
              % ∃-elimination %
0.2           split(p, (x, f).when(x, a.λg.inl ⟨a, g⟩, b.λg.inr ⟨b, g⟩)f) ∈ conc
            ]|
```

medskip For some time I believed that this example could only be derived with the aid of the equality rules of section 2. The above proof was however suggested by Erik Saaman. It is an excellent illustration of the use of a family of types in an elimination rule. (Note particularly step 0.1.3; the family of types used here is $Fw \Rightarrow conc$ indexed by $w \in A \vee B$.) Nevertheless I still find the derivation unsatisfactory since the final proof object contains a, to my mind, spurious λ-abstraction and function application. This example has therefore acted as a catalyst for a number of proposed changes to Martin-Löf's rules. For further discussion see [Ba?].

(End of example)

Exercise 1.11

Assuming that F and G have type $A \Rightarrow U_1$, construct proof objects for the following.

(a) $\exists(A, x.Fx \Rightarrow C) \Rightarrow \forall(A, x.Fx) \Rightarrow C$

(b) $\exists(A, x.Fx) \vee \exists(A, x.Gx) \Longleftrightarrow \exists(A, x.Fx \vee Gx)$

(c) $\forall(A, x.Fx \Rightarrow Gx) \Rightarrow \exists(A, x.Fx) \Rightarrow \exists(A, x.Gx)$

Assuming that F has type $A \vee B \Rightarrow U_1$, construct a proof object for the following.

(d) $\exists(A, x.F(\mathbf{inl}\ x)) \vee \exists(B, y.F(\mathbf{inr}\ y)) \Rightarrow \exists(A \vee B, w.Fw)$

Assuming that F has type $A \wedge B \Rightarrow U_1$ and G has type $A \Rightarrow U_1$, construct proof objects for the following.

(e) $\exists(A \wedge B, p.Fp) \Rightarrow \exists(A, a.\exists(B, b.F\langle a, b\rangle))$

(End of exercise)

(f) *Negation*

As mentioned earlier, negation is not a primitive of type theory. Instead $\neg A$ is modelled by $A \Rightarrow \emptyset$, where $\emptyset$ is the empty type. I.e. a proof that A is not true is interpreted as a method of constructing an object of the empty type given an object of A.

Several properties of negation arise immediately from earlier examples by substituting $\emptyset$ for one of the type variables A, B or C. Specifically, the substitution rule we use is the following.

$$\frac{\begin{array}{l} p \in P \\ \;|[\;\; x \in P \\ \;\triangleright\;\; r(x) \in R(x) \\ \;]| \end{array}}{r(p) \in R(p)} \qquad \text{Substitution}$$

Example 1.24 $\quad \lambda x.x \in \neg\emptyset$

The identity function maps the empty type to itself. Equally, the identity function is a method of proving that $\emptyset$ is not true.

Derivation

	% U_1-intro. %	
0	$\emptyset \in U_1$	
	% Example 1.6 %	
1.0	$\|[\quad A \in U_1$	
1.1	$\triangleright\quad \lambda x.x \in A \Rightarrow A$	
	$]\|$	
	% 0, 1, $A := \emptyset$ %	
2	$\lambda x.x \in \emptyset \Rightarrow \emptyset$	

Notes :

(a) There is of course a more direct derivation obtained by replacing A by $\emptyset$ in the derivation (rather than the result of example 1.6).

(b) An assignment statement is used to indicate a substitution. My apologies to ardent functional programmers but, after all, the semantics of assignment is substitution.

(End of Example)

Some examples of familiar properties of negation are as follows. In each case the property is established by substituting $\emptyset$ for the type variable C in the relevant example or exercise.

Example 1.25

(a) $\lambda w.\mathbf{when}(w, f.\lambda x.f(fst(x)), g.\lambda x.g(snd(x))) \in (\neg A \vee \neg B) \Rightarrow \neg(A \wedge B)$
(c.f. Example 1.16)

(b) $\lambda f.\lambda x.\lambda y.f\langle x, y\rangle \in \neg\exists(A, x.Fx) \Rightarrow \forall(A, x.\neg(Fx))$
(c.f. Example 1.22)

(c) $\lambda f.\lambda w.\mathbf{split}(w, (x, y).fxy) \in \forall(A, x.\neg Fx) \Rightarrow \neg\exists(A, x.Fx)$
(c.f. Exercise 1.11(a))

(d) $\lambda w.\mathbf{split}(w,(x,g).\lambda f.g(fx)) \in \exists(A,x.\neg Fx) \Rightarrow \neg\forall(A,x.Fx)$
(c.f. Exercise 1.11(b))

Examples (b), (c) and (d) are valid judgements in a context in which F has type $A \Rightarrow U_1$.
(End of Example)

Note that examples 1.25(b), (c) and (d) establish "three-quarters" of DeMorgan's laws for universal and existential quantification, viz. "$\neg\exists \iff \forall\neg$" and "$\exists\neg \Rightarrow \neg\forall$". The final "quarter", "$\neg\forall \Rightarrow \exists\neg$", is not generally true in constructive mathematics. Similarly, it is straighforward to establish that

$$\neg(A \vee B) \Rightarrow \neg A \wedge \neg B$$

which, together with example 1.25(a), forms "three-quarters" of DeMorgan's laws for conjunction and disjunction. Again, the final "quarter"

$$\neg(A \wedge B) \Rightarrow \neg A \vee \neg B$$

is not generally true.

Exercise 1.12 Construct proof objects for the following.

(a) $\neg A \wedge \neg B \Rightarrow \neg(A \vee B)$
(b) $(B \Rightarrow A) \Rightarrow (\neg A \Rightarrow \neg B)$
(c) $\neg\neg\neg A \Rightarrow \neg A$
(d) $\neg(A \vee B) \Rightarrow \neg A \wedge \neg B$
(e) $\neg(A \wedge \neg A)$
(f) $A \Rightarrow \neg\neg A$
(g) $\neg(A \Rightarrow B) \Rightarrow \neg B$
(h) $[\neg\neg(A \Rightarrow B) \wedge \neg B] \Rightarrow \neg A$

Make use of exercise 1.6(f) in the proof of (a), example 1.8 in the proof of (b) and exercise 1.2(c) in the proof of (c). Note that (f) is a curried version of (e).
(End of exercise).

The empty type, $\emptyset$, is an instance of an enumerated type (see section ??). It is peculiar in that it has no introduction rules. It does, however, have one elimination rule, namely that it is possible to construct an object of any type from an object of the empty type.

$$\frac{\begin{array}{l}|[\ w \in \emptyset \\ \triangleright\ R(w)\ type \\ ]| \\ q \in \emptyset\end{array}}{\mathsf{z}(q) \in R(q)}\ \emptyset\text{-elimination}$$

(For uniformity with the enumerated-type elimination rules discussed later we should write "$\mathbf{case}(q)$" instead of $\mathsf{z}(q)$. The absence of cases in the expression looks odd and so we prefer the abbreviated form.)

A first example of a tautology that relies on the $\emptyset$-elimination rule is the following.

Example 1.26 $\quad \lambda f.\lambda y.z(f(\lambda x.y)) \in \neg(A \Rightarrow B) \Rightarrow B \Rightarrow A$

Derivation

```
0.0      |[  f ∈ (A ⇒ B) ⇒ ∅                          %¬(A ⇒ B)%
0.1.0    ▷  |[  y ∈ B
0.1.1.0     ▷  |[  x ∈ A
               ▷     % 0.1.0 %
0.1.1.1           y ∈ B
               ]|
0.1.2          λx.y ∈ A ⇒ B
                  % 0.0, 0.1.2, ⇒-elim. %
0.1.3          f(λx.y) ∈ ∅
                  % 0.1.3, ∅-elim. %
0.1.4          z(f(λx.y)) ∈ A
            ]|
         ]|
```

(End of example)

Exercise 1.13 Prove the following.

(a) $\neg(A \Rightarrow B) \Rightarrow \neg\neg A$

(b) $A \vee B \Rightarrow \neg A \Rightarrow B$

(End of exercise)

In presentations of the classical propositional calculus it is common to see a list of tautologies describing the algebraic properties of the connectives. It is usual also to provide one or two examples of propositional forms that are not tautologies, a single counterexample being sufficient to refute each. For example, $A \Rightarrow A$ is a tautology and $A \Rightarrow B$ is not. A counterexample for the latter is $A = \mathbf{true}$, $B = \mathbf{false}$.

In these notes, a *tautology* is a family of types $P(A, B, C, \ldots)$, dependent on the type variables $A, B, C, \ldots$ assumed to be objects of U_1, such that

$$\forall(U_1, A.\forall(U_1, B.\forall(U_1, C.\ldots.P(A, B, C, \ldots))))$$

is provable. Given such a family of types P there are now three possibilities :

(a) P can be proved to be a tautology.

(b) A counterexample refuting the universal validity of P can be exhibited.

(c) Neither of (a) or (b) is the case.

Example 1.27 $\quad \exists(U_1, A.\neg A)$

A is not a tautology.

Derivation

```
0    ∅ ∈ U1
        % example 1.24 %
1    λx.x ∈ ¬∅
        % 0, 1, ∃-intro %
2    ⟨∅, λx.x⟩ ∈ ∃(U1, A.¬A)
```

(End of example)

Example 1.28 $\exists(U_1, A.\exists(U_1, B.\neg(A \Rightarrow B)))$

Here we use $\emptyset$ to represent **false** and $\mathcal{N}$ to represent **true**. (Any non-empty type will do in place of $\mathcal{N}$.)

Derivation

0 $\mathcal{N} \in U_1$
1 $\emptyset \in U_1$
2.0 |[$f \in \mathcal{N} \Rightarrow \emptyset$
2.1 ▷ $0 \in \mathcal{N}$
% 2.0, 2.1, $\Rightarrow$-elim %
2.2 $f0 \in \emptyset$
]|
3 $\lambda f.f0 \in \neg(\mathcal{N} \Rightarrow \emptyset)$
% 0, 1, 3, $\exists$-intro %
4 $\langle \mathcal{N}, \langle \emptyset, \lambda f.f0 \rangle \rangle \in \exists(U_1, A.\exists(U_1, B.\neg(A \Rightarrow B)))$

(End of example)

Exercise 1.14 Construct counterexamples to the following propositions:

(a) $A \vee \neg B \Rightarrow \neg(A \vee B)$
(b) $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow C)$
(c) $((A \wedge B) \Rightarrow (A \wedge C)) \Rightarrow A \wedge (B \Rightarrow C)$

(End of exercise)

The law of the excluded middle is an example of a propositional form that can neither be proved to be a tautology, not can it be refuted by a counterexample. Specifically, by substituting $\emptyset$ for B in exercise 1.5 we obtain the tautology

$$\neg\neg(A \vee \neg A).$$

Quantifying over A we obtain

$$\forall(U_1, A.\neg\neg(A \vee \neg A))$$

and applying the result that "$\forall\neg \Rightarrow \neg\exists$" we obtain

$$\neg\exists(U_1, A.\neg(A \vee \neg A)).$$

We interpret the last proposition as the statement that it is impossible to exhibit a type, A, for which the law of the excluded middle does not hold. (The last two steps can be proved formally within the theory. We have not done so because the mechanism for applying "$\forall\neg \Rightarrow \neg\exists$" has not yet been discussed. As an exercise the reader may like to prove $\neg\exists(U_1, A.\neg(A \vee \neg A)$ directly. Refer to exercise 1.5 for hints.)

The form $\neg\neg P$ is of interest because it asserts that P cannot be refuted. Later (section on universes) we prove formally that any classical tautology cannot be refuted. For the moment we just give one more particular example.

Example 1.29 $\neg\neg((A \Rightarrow B) \vee (B \Rightarrow A))$

A classical proof of $(A \Rightarrow B) \vee (B \Rightarrow A)$ when translated into type theory would take the following form. Consider three cases, A non-empty, B non-empty and both A and B empty. In the first case choose an arbitrary element x, say, of A. Then the constant function $\lambda y.x$ is a proof of $B \Rightarrow A$ and so $\mathbf{inr}(\lambda y.x)$ is a proof of $(A \Rightarrow B) \vee (B \Rightarrow A)$. Similarly, $\mathbf{inl}(\lambda x.y)$ is a proof of $(A \Rightarrow B) \vee (B \Rightarrow A)$, where y is an arbitrary element of B. Finally, if A and B are both empty the identity function proves $A \Rightarrow B$ (or $B \Rightarrow A$).

This proof is not valid in type theory because it relies on the law of the excluded middle — that it is decidable whether A and/or B is nonempty. The statement that it is impossible to refute $(A \Rightarrow B) \vee (B \Rightarrow A)$ is therefore the best we can hope for.

Derivation

0.0 |[$f \in \neg((A \Rightarrow B) \vee (B \Rightarrow A))$
0.1.0 ▷ |[$x \in A$
0.1.1 ▷ $\lambda y.x \in B \Rightarrow A$
0.1.2 $\mathbf{inr}(\lambda y.x) \in (A \Rightarrow B) \vee (B \Rightarrow A)$
% 0.0, 0.1.2 ⇒-elim. %
0.1.3 $f(\mathbf{inr}(\lambda y.x)) \in \emptyset$
% 0.1.3, ∅-elim %
0.1.4 $z(f(\mathbf{inr}(\lambda y.x))) \in B$
]|
% 0.1.0, 0.1.4, ⇒-intro %
0.2 $\lambda x.z(f(\mathbf{inr}(\lambda x.y))) \in A \Rightarrow B$
% 0.2, ∨-intro %
0.3 $\mathbf{inl}(\lambda x.z(f(\mathbf{inr}(\lambda x.y)))) \in (A \Rightarrow B) \vee (B \Rightarrow A)$
% 0.0, 0.3, ⇒-elim %
0.4 $f(\mathbf{inl}(\lambda x.z(f(\mathbf{inr}(\lambda x.y))))) \in \emptyset$
]|

(End of example)

Exercise 1.15

(Hard) Prove :

$$\neg\neg[(A \Rightarrow B \vee C) \Rightarrow ((A \Rightarrow B) \vee (A \Rightarrow C))]$$

(Easy) Explain why the classical tautology

$$(A \Rightarrow B \vee C) \Rightarrow ((A \Rightarrow B) \vee (A \Rightarrow C))$$

is not constructively true.

(End of exercise)

To conclude this discussion some comments on "indirect proof" in classical mathematics may be of value. Indirect proof (as I understand it, anyway) takes one of two forms. The first form is that to prove that a proposition A is true one proves that $\neg A$ is false. In other words one proves $\neg\neg A$ which, in constructive mathematics is not the same as A. This is the form used by Courant and Robbins in their proof of the infinitude of primes quoted earlier. Another form of indirect proof is used to establish a proposition like $A \Rightarrow B$. The strategy is to assume $\neg B$ and use it to establish $\neg A$. This is based on the classical tautology $(\neg B \Rightarrow \neg A) \Rightarrow A \Rightarrow B$. However, this is also not a valid argument form in constructive mathematics. Indeed, it *is* a tautology of constructive mathematics that

$$(\neg B \Rightarrow \neg A) \Rightarrow A \Rightarrow \neg\neg B$$

(this is easy to prove) and, also,

$$\neg\neg[(\neg B \Rightarrow \neg A) \Rightarrow A \Rightarrow B]$$

(this is harder to prove, but see the section on universes).

Thus a proof that $\neg B$ implies $\neg A$ establishes that in a context in which A is true B cannot be refuted. Moreover the classical basis for such a proof of $A \Rightarrow B$ can itself not be refuted in constructive mathematics, but it is not constructively valid because a proof of $\neg B \Rightarrow \neg A$ does not provide a method of constructing an element of B given an element of A.

Exercise 1.16 (See [CH, section 3.3])
Make the following definitions (where P and Q are small types)

$$[P] \equiv \neg\neg P$$
$$P?Q \equiv \neg P \Rightarrow [Q]$$

($[P]$ is interpreted as the classical truth of P, and ? is interpreted as classical disjunction.)

Prove the following :
(i) $A?\neg A$
(ii) $[A]?[B] \Rightarrow A?B$
(iii) $[A?B] \Rightarrow A?B$
(iv) $A?\emptyset \Longleftrightarrow [A]$

(End of exercise)

The following sclose is the one promised at the beginning of this section. It terminates the scope of the assumptions $A, B, C \in U_1$.

]|

1.5 Conclusions

What we have presented so far can be viewed in two lights. In one light we have described a theory of the primitive type constructors — disjoint sum, Cartesian product, dependent product, function spaces and dependent function spaces. In this light the theorems we have established can be regarded as instances of *type-coercions* — mappings from one type to another that are independent of the primitive types. In another light we have presented a constructive theory of the predicate calculus and discussed its relationship to the classical predicate calculus.

There are cogent arguments for discussing the theory only in respect of its use as a flexible and elegant type discipline. Such a discussion would use Martin-Löf's notation of "+" for disjoint sum, "×" for Cartesian product, "Σ" for dependent product, "→" for function spaces and "Π" for dependent function space. I have chosen not to do so not for any philosphical motives but simply to appeal to the reader's existing familiarity with classical reasoning. (In all honesty I have to admit that writing these notes is a — continuing — learning exercise for myself. When I began I could claim to have an adequate knowledge of classical reasoning but zero knowledge of intuitionistic reasoning. For my own peace of mind I have therefore been obliged to delve deeply into both. These notes reflect the way I have set about the task.)

This section has had several objectives, some of them more explicit than others. Among the explicit objectives have been those of introducing the basic form and structure of the rules of inference in Martin-Löf's theory and of explicating the notion of propositions as types. Among the implicit objectives has been my intention to present a personal view on the form that an introduction to advanced programming techniques should take.

My view is that programming languages should be regarded first and foremost as algebras and programming as a calculational activity within such an algebra. (Of course I am not alone in this view but I can claim to have proposed it as long ago as 1973.) Any presentation of a programming language should therefore emphasise important algebraic properties like associativity, symmetry and distributivity. It is appropriate therefore that this first section should contain a catalogue of many such properties of the type constructors.

2 Equalities and the Equality Type

Normally when we write a program it is for the purpose of computing specific values. Although we have claimed an identity between proof objects and programs we haven't yet shown how to compute with proof objects. This is done with the aid of *equality judgements* of the form

$$a = b \in A$$

Such a judgement states that a and b are equal objects within the type A.

Closely related to the equality judgements is the *equality type*. The type $a =_A b$ is identified with the proposition "a and b are equal objects of type A." In essence the two different judgement forms

$$a = b \in A$$

and

$$r \in a =_A b$$

mean the same thing; moving from one form to the other is a trick that plays a central role in the theory. (I call it the Martin-Löf bottleneck!)

The fourth judgement form is $A = B$ and asserts that A and B are equal types.

Most mathematical work is done within a single type structure (for example, number theory is about natural numbers and analysis is about real numbers). The notion that equality is typed, and that objects can be equal in one type and unequal in another, is therefore an unfamiliar one which can create surprises and also difficulties. For example, we see later (exercise 2.6) that $\lambda x.0 = \lambda x.x \in \emptyset \Rightarrow \emptyset$ but it is clearly not the case that $\lambda x.0 = \lambda x.x \in \mathcal{N} \Rightarrow \mathcal{N}$. The difficulties caused by typed equality are therefore to do with substitution of equals for equals, such rules being less easy to apply than the rule to which most of us are used. Another disconcerting feature of the theory is the bewilderingly large number of equality rules. However the rules fall into a pattern which, once observed, makes them easier to comprehend. We begin by considering general equality rules (reflexivity, symmetry etc.) after which we consider the equality rules that are specific to individual types. The latter can be divided into two categories: *computation rules* which say how to compute the values of noncanonical objects in a type, and *congruence* rules which express the fact that equality is preserved by substitution in object expressions.

Throughout this section we assume once again that A and B are objects of the first universe.

$[\![\ A \in U_1; B \in U_1$

2.1 Type Formation Rule

We begin our account of equalities with the type formation rule:

$$\frac{\begin{array}{l} p \in P \\ q \in P \end{array}}{p \underset{P}{=} q \ \ type} \qquad =\text{-formation}$$

Note how the =-formation rule relies on the ability to make judgements of the form $p \in P$. This is a significant aspect of the formal system with the implication that it is no longer possible to claim that the judgement P *type* means that P is a "well-formed type expression". The two judgement forms are inextricably bound together!

Our first two examples are in preparation for example 2.2; the reader may wish to scan them quickly before moving on to the next section.

Example 2.0

$$\begin{array}{l} [\![\ a \in A; b \in B \\ \triangleright\ [\![\ y \in B \\ \quad \triangleright\ \langle a, y\rangle \ \underset{A \wedge B}{=} \langle a, b\rangle \ type \\ \quad]\!] \\ ]\!] \end{array}$$

Derivation

0.0 $[\![\ a \in A; b \in B$
0.1 $\triangleright\ \langle a, b\rangle \in A \wedge B$
0.2.0 $[\![\ y \in B$
0.2.1 $\triangleright\ \langle a, y\rangle \in A \wedge B$
% 0.1, 0.2.1, =-form. %
0.2.2 $\langle a, y\rangle \ \underset{A \wedge B}{=} \langle a, b\rangle \ type$
$]\!]$
$]\!]$

(End of example)

Example 2.1

$$\begin{array}{l} [\![\ a \in A; b \in B \\ \triangleright\ [\![\ x \in A \\ \quad \triangleright\ \exists(B, y.\langle x, y\rangle \ \underset{A \wedge B}{=} \langle a, b\rangle) \ \ type \\ \quad]\!] \\ ]\!] \end{array}$$

Derivation

0.0 |[$a \in A; b \in B$

0.1 ▷ $\langle a, b \rangle \in A \wedge B$

0.2.0 |[$x \in A$

0.2.1.0 ▷ |[$y \in B$

0.2.1.1 ▷ $\langle x, y \rangle \in A \wedge B$

% 0.1, 0.2.1.1, =-form %

0.2.1.2 $\langle x, y \rangle \underset{A \wedge B}{=} \langle a, b \rangle$ *type*

]|

% B *type*, 0.2.1, ∃-form %

0.2.1 $\exists(B, y.\langle x, y \rangle \underset{A \wedge B}{=} \langle a, b \rangle)$ *type*

]|

]|

(End of example)

2.2 Elementary rules

Equality obeys the usual rules of reflexivity, symmetry and transitivity.

$$\frac{p \in P}{p = p \in P} \qquad \frac{}{P = P} \qquad \text{reflexivity}$$

$$\frac{p = q \in P}{q = p \in P} \qquad \frac{P = Q}{Q = P} \qquad \text{symmetry}$$

$$\frac{p = q \in P \quad q = r \in P}{p = r \in P} \qquad \frac{P = Q \quad Q = R}{P = R} \qquad \text{transitivity}$$

An object of the equality type is introduced by making equality judgements.

$$\frac{p = q \in P}{\mathrm{e} \in p \underset{P}{=} q} \qquad \text{=-introduction}$$

Conversely, if we are able to construct an object of an equality type then we can also make the corresponding equality judgement.

$$\frac{r \in p \underset{P}{=} q}{p = q \in P} \qquad \text{=-elimination}$$

Note that by applying =-introduction immediately after =-elimination it is always possible to replace any object r in an equality type by the object e. I.e. we have the derived rule

$$\frac{r \in p \underset{P}{=} q}{\mathrm{e} \in p \underset{P}{=} q}$$

In practice this means that e is the only object of the equality type $p \underset{P}{=} q$ (if such an object exists). Thus the object e has no computational significance and is there only to maintain uniformity of the judgements. Later we abandon it.

Example 2.2

$$\lambda p.\ \mathbf{split}(p, (a, b).\langle a, \langle b, \mathrm{e}\rangle\rangle)$$
$$\in \forall(A \wedge B, p.\exists(A, x.\exists(B, y.\langle x, y\rangle \underset{A \wedge B}{=} p)))$$

From a theoretical viewpoint this example is truly remarkable. It asserts that every element of a conjunction is a pair — in other formal systems such a judgement would be implicit or stated as an axiom. From a practical viewpoint this and similar properties are

indispensable since we use them often to simplify complex objects. Thus the following derivation is well worthwhile studying.

Derivation

0.0 |[$p \in A \wedge B$
0.1.0 ▷ |[$a \in A; b \in B$
▷ % 0.1.0, $\wedge$-intro %
0.1.1 $\langle a, b\rangle \in A \wedge B$
% 0.1.1, refl %
0.1.2 $\langle a, b\rangle \underset{A\wedge B}{=} \langle a, b\rangle$
% 0.1.2, =-intro %
0.1.3 $\mathrm{e} \in \langle a, b\rangle \underset{A\wedge B}{=} \langle a, b\rangle$
% example 2.0, 0.1.0, 0.1.3, $\exists$-intro %
0.1.4 $\langle b, \mathrm{e}\rangle \in \exists(B, y.\langle a, y\rangle \underset{A\wedge B}{=} \langle a, b\rangle)$
% example 2.1, 0.1.0, 0.1.4, $\exists$-intro %
0.1.5 $\langle a, \langle b, \mathrm{e}\rangle\rangle \in \exists(A, x.\exists(B, y.\langle x, y\rangle \underset{A\wedge B}{=} \langle a, b\rangle))$
]|
% 0.0, 0.1, $\wedge$-elim %
0.1 $\mathrm{split}(p, (a, b).\langle a, \langle b, \mathrm{e}\rangle\rangle) \in \exists(A, x.\exists(B, y.\langle x, y\rangle \underset{A\wedge B}{=} p))$
]|

The steps worth studying are the final three. Steps 0.1.4 and 10..5 are the first real examples we have been able to give of the use of families of types. Specifically, example 2.0 states that $\langle a, y\rangle \underset{A\wedge B}{=} \langle a, b\rangle$ is to be regarded as a family of types where y ranges over the type B. A particular instance of this family is the type $\langle a, b\rangle \underset{A\wedge B}{=} \langle a, b\rangle$ about which a judgement is made in step 0.1.3. Thus 0.1.4 follows by the $\exists$-introduction rule. Similarly, example 2.1 states that $\exists(B, y.\langle x, y\rangle \underset{A\wedge B}{=} \langle a, b\rangle)$ is to be regarded as a family of types where x ranges over the type A. A particular instance of this family is the type $\exists(B, y.\langle a, y\rangle \underset{A\wedge B}{=} \langle a, b\rangle)$ about which a judgement has been made in step 0.1.4. Thus 0.1.5 follows by the $\exists$-introduction rule. Step 0.1 illustrates the use of substitution in the type expression when using $\wedge$-elimination — again something we have not illustrated earlier.

(End of example)

Exercise 2.0 Prove : $\forall(A \vee B, d.\exists(A, x.\mathbf{inl}\ x \underset{A\vee B}{=} d) \vee \exists(B, y.\mathbf{inr}\ y \underset{A\vee B}{=} d))$

Formulate a property similar to that in example 2.2 relating to existential quantification. Establish the property.

(End of exercise)

2.3 Substitutivity Properties

Two sets of rules express substitutivity of equals for equals. The first set expresses the substitutivity of equal objects for equal objects in a type.

$$\frac{\begin{array}{l}[\![\ x \in P \\ \triangleright\ r(x) = s(x) \in R(x) \\ ]\!] \\ p = q \in P\end{array}}{r(p) = s(q) \in R(p)} \qquad \frac{\begin{array}{l}[\![\ x \in P \\ \triangleright\ Q(x) = R(x) \\ ]\!] \\ p = q \in P\end{array}}{Q(p) = R(q)} \qquad \text{substitution}$$

Example 2.3 $\quad \forall(A,\ a, a'.\forall(A \Rightarrow B, f.(a \underset{A}{=} a') \Rightarrow (fa \underset{B}{=} fa')))$

Derivation

```
0.0     [[  a, a' ∈ A; f ∈ A ⇒ B
0.1.0   ▷   [[  r ∈ a =_A a'
            ▷     % 0.1.0, =-elim %
0.1.1           a = a' ∈ A
0.1.2.0         [[  x ∈ A
                ▷     % 0.0,0.1.2.0, ⇒-elim,refl %
0.1.2.1           fx = fx ∈ B
                ]]
                  % 0.1.1,0.1.2,subst %
0.1.3           fa = fa' ∈ B
                  % 0.1.3, =-intro %
0.1.4           e ∈ fa =_B fa'
            ]]
        ]]
```

End of example

Exercise 2.1 Prove the following :

(a) $\quad \forall(A,\ a, a'.\forall(A \Rightarrow B,\ f, g.(a \underset{A}{=} a') \Rightarrow (f \underset{A \Rightarrow B}{=} g) \Rightarrow (fa \underset{B}{=} ga')))$

(b) $\quad \forall(A \Rightarrow B,\ f, g.(f \underset{A \Rightarrow B}{=} g) \Rightarrow (\lambda x.fx \underset{A \Rightarrow B}{=} \lambda x.gx))$

(c) $\quad \forall(A,\ a, a'.\forall(B,\ b, b'.(a \underset{A}{=} a') \Rightarrow (b \underset{A}{=} b') \Rightarrow (\langle a, b\rangle \underset{A \wedge B}{=} \langle a', b'\rangle)))$

(d) $\quad \forall(A,\ a, a'.(a \underset{A}{=} a') \Rightarrow (\text{inl } a \underset{A \vee B}{=} \text{inl } a'))$

End of exercise

The second set of rules expresses the substitutivity of equal types.

$$\frac{\begin{array}{l}P = Q \\ p \in P\end{array}}{p \in Q} \qquad \frac{\begin{array}{l}P = Q \\ p = q \in P\end{array}}{p = q \in Q} \qquad \text{type equality}$$

Example 2.4 This example has been removed.

Exercise 2.2 This exercise has been removed.

Example 2.5 (Arguments for absurdity)

$$\forall(U_1,\ \ A, B, C.\forall(A, x.\forall(B, y.\emptyset \Rightarrow zx \underset{C}{=} zy)))$$

The constant z introduced by the $\emptyset$-elimination rule has a single argument but this argument has no significance and may be replaced by any object of any type.

Derivation

		% U_1-form, U_1-elim, refl %
0.0	][	$X \in U_1$
0.1	▷	$X = X$
	]]	
1.0	][	$A, B, C \in U_1$
1.1.0	▷	][$x \in A; y \in B$
1.1.1.0		▷][$a \in \emptyset$
		▷ % $\emptyset \in U_1$, 1.0, =-form %
1.1.1.1		$\emptyset \underset{U_1}{=} A$ *type*
		% 1.1.1.0, 1.1.1.1, $\emptyset$-elim %
1.1.1.2		$za \in \emptyset \underset{U_1}{=} A$
		% 1.1.1.2, =-elim %
1.1.1.3		$\emptyset = A \in U_1$
		% 0, 1.1.1.3, subst %
1.1.1.4		$\emptyset = A$
		% 1.1.0, 1.1.1.4, type equality %
1.1.1.5		$x \in \emptyset$
		% 1.1.1.5, 1.0, U_1-elim, $\emptyset$-elim %
1.1.1.6		$zx \in C$
		% similarly %
1.1.1.7		$zy \in C$
		% 1.1.1.6, 1.1.1.7, =-form %
1.1.1.8		$zx \underset{C}{=} zy$ *type*
		% 1.1.1.0, 1.1.1.8, $\emptyset$-elim %
1.1.1.9		$za \in zx \underset{C}{=} zy$
		]]
		]]
		]]

Notes

(a) The step from 1.1.1.3 to 1.1.1.4 can be made directly using an inference rule to be presented in section 3. Our use of step 0 was a simple trick to avoid introducing the rule at this stage.

(b) I do not like the use of "similarly" in step 1.1.1.7. However, the substitution rule that I have in mind does not appear to be included in Martin-Löf's rules. This may or may not be significant.

(c) In spite of its apparent uselessness, this example is important to us when we consider subset types. The property was pointed out to me (in a slightly different guise) by Stuart Anderson.

(End of example)

Exercise 2.3 Show that $\lambda x.x \in \emptyset \Rightarrow A$

(End of exercise)

2.4 Congruence and Extensionality Properties

A number of rules in the theory express congruence properties of equality with respect to the various operators. Some of these rules, for instance the congruence of pairing, can be established using the simple properties of substitution already stated. Where an operator involves abstraction (e.g. λ and **when**), however, the properties we require cannot be established using simple substitution, and they make significant use of the rules in this section.

$$\frac{\begin{array}{l}[\![\ x \in P \\ \triangleright\ p(x) = q(x) \in Q \\ ]\!]\end{array}}{\lambda x.p(x) = \lambda x.q(x) \in P \Rightarrow Q} \qquad \lambda\text{-congruence}$$

Exercise 2.4 Show that $\lambda x.x = \lambda x.zx \in \emptyset \Rightarrow A$
(End of exercise)

The rule known as "eta-reduction" in the Lambda calculus is an ad-hoc addition to the system (i.e. it does not fall into the pattern of the other rules). It and λ-congruence guarantee extensionality of functions (see example 2.6).

$$\frac{f \in P \Rightarrow Q}{\lambda x.fx = f \in P \Rightarrow Q} \qquad \eta\text{-reduction}$$

(where x does not occur free in f)
(Note that an immediate consequence of this rule is the property $\forall(A \Rightarrow B, f.f \underset{A \Rightarrow B}{=} \lambda x.fx)$. This is the closure property of implication akin to the closure properties stated in example 2.2 and exercise 2.0. The fact that it is not derivable except by η-reduction emphasises the ad-hoc nature of the rule. Martin-Löf has now eliminated the anomaly by "the adoption of a systematic higher level notation" [Sa]. (We will discuss the new notation and the revised rules for implication and universal quantification later.)

Example 2.6 (Extensionality of functions)

$$\forall(A \Rightarrow B,\ f,g.\forall(A, a.fa \underset{B}{=} ga) \Rightarrow (f \underset{A \Rightarrow B}{=} g))$$

Derivation

0.0 ||[$f, g \in A \Rightarrow B$

0.1.0 ▷ ||[$h \in \forall(A, a.fa \underset{B}{=} ga)$

0.1.1.0 ▷ ||[$a \in A$

▷ % 0.1.0, 0.1.1.0, $\forall$-elim %

0.1.1.1 $ha \in fa \underset{B}{=} ga$

% 0.1.1.1, =-elim %

0.1.1.2 $fa = ga \in B$

]|

% 0.1.1, λ-congr %

0.1.2 $\lambda a.fa = \lambda a.ga \in A \Rightarrow B$

% 0.0, η-red %

0.1.3 $\lambda a.fa = f \in A \Rightarrow B$

% 0.0, η-red %

0.1.4 $\lambda a.ga = g \in A \Rightarrow B$

% 0.1.2, 0.1.3, 0.1.4, sym, trans %

0.1.5 $f = g \in A \Rightarrow B$

]|

]|

(End of Example)

Exercise 2.5 (The most elementary instance of extensionality)
Show that $\forall(\emptyset \Rightarrow A, f.f \underset{\emptyset \Rightarrow A}{=} \lambda x.z(x))$

(End of exercise)

Take careful note of the premise in the η-reduction rule. It is not possible to derive — for very good reasons — the judgements $z \in \emptyset \Rightarrow A$ or $\mathsf{inl} \in A \Rightarrow A \vee B$ within the theory. Thus, for example, the consequent in exercise 2.5 cannot be simplified to $f \underset{\emptyset \Rightarrow A}{=} z$.

Example 2.8 (Proof objects of negations)

$$\forall(\neg A, f.f \underset{\neg A}{=} \lambda x.x)$$

The identity function is the unique proof object of $\neg A$, if such an object exists. Thus there is no computational content in any proof object of $\neg A$.

Derivation

0.0 |[$f \in \neg A$
0.1.0 ▷ |[$x \in A$
▷ % 0.0, 0.1.0, $\Rightarrow$-elim %
0.1.1 $fx \in \emptyset$
% 0.1.1, $\emptyset$-elim %
0.1.2 $z(fx) \in fx \underset{\emptyset}{=} x$
% 0.1.2, =-elim %
0.1.3 $fx = x \in \emptyset$
]|
% 0.1, λ -congr %
0.2 $\lambda x.fx = \lambda x.x \in \neg A$
% 0.0, η-red %
0.3 $\lambda x.fx = f \in \neg A$
% 0.2, 0.3, sym, trans %
0.4 $f = \lambda x.x \in \neg A$
]|

Several steps have been omitted in the deduction of 0.1.2. We have to show that $fx \underset{\emptyset}{=} x$ is a type which, in turn, requires a proof of $x \in \emptyset$. The latter judgement can be derived as in example 2.5.

(End of example)

Exercise 2.6 Show that $\lambda x.0 = \lambda x.x \in \emptyset \Rightarrow \emptyset$

(End of exercise)

The congruence rules for function application, pairing and injection add nothing new to the system. Following Martin-Löf we include them for uniformity.

$$\frac{p = q \in P \qquad r = s \in P \Rightarrow Q}{rp = sq \in Q}$$ congruence of function application

$$\frac{p = p' \in P \qquad q = q' \in Q}{\langle p, q\rangle = \langle p', q'\rangle \in P \wedge Q}$$ $\langle\rangle$-congruence

$$\frac{p = q \in P}{\mathrm{inl}\ p = \mathrm{inl}\ q \in P \vee Q}$$ inl-congruence

$$\frac{p = q \in Q}{\mathrm{inr}\ p = \mathrm{inr}\ q \in P \vee Q}$$ inr-congruence

There are similar rules for universal quantification and existential quantification.

Exercise 2.7 Show that $\forall(A,\ a,b.\neg(\text{inl } a \underset{A\vee B}{=} \text{inl } b) \Rightarrow \neg(a \underset{A}{=} b))$.
Complete the following proposition and prove it.

$$\forall(A,\ a,a'.\forall(B,\ b,b'.\neg(\langle a,b\rangle \underset{A\wedge B}{=} \langle a',b'\rangle) \Rightarrow ???))$$

(End of exercise)

The abstractions **when** and **split** also preserve equality; like λ-congruence the following rules appear to be independent of other rules.

$$\frac{\begin{array}{l}|[\ x \in P \wedge Q \\ \triangleright\ R(x)\ \ type \\ ]|\\ p = q \in P \wedge Q \\ |[\ y \in P; z \in Q \\ \triangleright\ r(y,z) = s(y,z) \in R(\langle y,z\rangle) \\ ]|\end{array}}{\text{split}(p,(y,z).r(y,z)) = \text{split}(q,(y,z).s(y,z)) \in R(p)}\ \textbf{split}\text{-congruence}$$

$$\frac{\begin{array}{l}|[\ x \in P \vee Q \\ \triangleright\ R(x)\ type \\ ]|\\ p = q \in P \vee Q \\ |[\ y \in P \\ \triangleright\ r(y) = r'(y) \in R(\text{inl } y) \\ ]|\\ |[\ z \in Q \\ \triangleright\ s(z) = s'(z) \in R(\text{inr } z) \\ ]|\end{array}}{\text{when}(p,y.r(y),z.s(z)) = \text{when}(q,y.r'(y),z.s'(z)) \in R(p)}\ \textbf{when}\text{-congruence}$$

*** Examples and exercises exploiting the above rules appear in the next section. Any suggestions for further examples that can be included at this point in the text would be welcome.

2.5 Computation Rules

Martin-Löf calls the rules in this section the "equality rules". However, their function is to say how to compute the value of non-canonical objects; we therefore prefer to refer to them as "computation rules".

We recall that the canonical objects of a type are created by the introduction rules, and non-canonical objects are created by the elimination rules. Thus the non-canonical objects associated with implication are function applications, those associated with disjunctions are when expressions etc. It is an important attribute of the theory that it suffices to specify how to compute the value of a non-canonical object when applied to a canonical object of the associated type. Thus, it suffices to define how to evaluate the application of λ-abstraction (a canonical object of an implication-type) to a given argument; it is not necessary to say how to evaluate the application of a non-canonical functional expression to a given argument. The closure properties discussed in example 2.2 and exercise 2.0 are thus both vital to our understanding of the theory and to its practical application.

In the jargon of computing science, expression evaluation is "lazy", i.e., an expression is fully evaluated if its principal or outermost connective is a canonical-object constructor. Thus, for example, in section 1.3(f) we saw lots of expressions involving the (non-canonical) constructor z. This does not give rise to any problems even though the latter may be regarded as a non-terminating computation. However, were evaluation non-lazy, say call-by-value, then many of the objects we construct would be undefined (in the sense of non-terminating).

(a) Function application

The computation rule for function application is the familiar rule of β-reduction in the Lambda calculus.

$$\frac{\begin{array}{l}[\![\ x \in P \\ \triangleright\ q(x) \in Q \\ ]\!] \\ p \in P\end{array}}{(\lambda x.q(x))p = q(p) \in Q} \quad \Rightarrow\text{-computation}$$

Example 2.8 $\quad II = I$

The examples and exercises that follow (up to and including exercise 2.10) are all well-known properties of the combinators, S, K, I and B [CF,St, Tu]. In a typed system the results are slightly more complex, but also a little clearer. In this example we prove that $I_1 I_0 = I_0$ where $I_1 = \lambda x.x \in (A \Rightarrow A) \Rightarrow (A \Rightarrow A)$ and $I_0 = \lambda y.y \in A \Rightarrow A$. Since this is a very elementary example we also use it to illustrate our technique for introducing definitions.

Derivation

0.0 |[$I_1 \in (A \Rightarrow A) \Rightarrow (A \Rightarrow A)$
0.1.0 ▷ |[$r \in I_1 = \lambda x.x$
0.1.1.0 ▷ |[$I_0 \in A \Rightarrow A$
0.1.1.1.0 ▷ |[$s \in I_0 = \lambda y.y$
0.1.1.1.1.0 ▷ |[$x \in A \Rightarrow A$
0.1.1.1.1.1 ▷ $x \in A \Rightarrow A$
]|
% 0.1.1.1.1, 0.1.1.1.0, ⇒-comp %
0.1.1.1.2 $(\lambda x.x)I_0 = I_0 \in A \Rightarrow A$
% 0.1.0, 0.1.1.1.2, exercise 2.1(a) %
0.1.1.1.3 $I_1 I_0 = I_0 \in A \Rightarrow A$
]|
]|
]|
]|

Two steps are used to introduce the abbreviation I_1 for $\lambda x.x$. First we say what type I_1 has (step 0.0) and then we assume (step 0.1.0) that we have an object of the equality type $I_1 = \lambda x.x$. (The type $(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$ should be made explicit in this second step but that would be overkill.) The same two steps have been used to introduce I_0. Note that the definition of I_0 is never used.

(End of example)

Example 2.9 $BI = I$

In this case the property we prove is $BI_0 = I_1$ where $B = \lambda x.\lambda y.\lambda z.x(yz) \in (A \Rightarrow A) \Rightarrow (A \Rightarrow A) \Rightarrow A \Rightarrow A$, and I_0 and I_1 are as above.

Derivation

0.0 |[$B = \lambda x.\lambda y.\lambda z.x(yz) \in (A \Rightarrow A) \Rightarrow (A \Rightarrow A) \Rightarrow A \Rightarrow A$
0.1 ; $I_0 = \lambda x.x \in A \Rightarrow A; I_1 = \lambda y.y \in (A \Rightarrow A) \Rightarrow A \Rightarrow A$
0.2.0 ▷ |[$x \in A \Rightarrow A$
▷ % 0.1.0, ⇒-intro, ⇒-elim %
0.2.1 $\lambda y.\lambda z.x(yz) \in (A \Rightarrow A) \Rightarrow A \Rightarrow A$
]|
% 0.0, 0.1, ⇒-comp %
0.3 $(\lambda x.\lambda y.\lambda z.x(yz))\ I_0 = \lambda y.\lambda z.I_0(yz) \in (A \Rightarrow A) \Rightarrow A \Rightarrow A$
% 0.0, 0.2, exercise 2.1(a) %
0.4 $BI_0 = \lambda y.\lambda z.I_0(yz) \in (A \Rightarrow A) \Rightarrow A \Rightarrow A$
0.5.0 |[$y \in A \Rightarrow A$
0.5.1.0 ▷ |[$z \in A$
▷ % 0.4.0, 0.4.1.0, ⇒-elim %
0.5.1.1 $yz \in A$
0.5.1.2.0 |[$x \in A$
0.5.1.2.1 ▷ $x \in A$
]|
% 0.4.1.1, 0.4.1.2, ⇒-comp %
0.5.1.3 $(\lambda x.x)(yz) = yz \in A$
% 0.0, 0.4.1.3, exercise 2.1(a) %
0.5.1.4 $I_0(yz) = yz \in A$
]|
% 0.4.1, λ-congr %
0.5.2 $\lambda z.I_0(yz) = \lambda z.yz \in A$
% 0.4.0, η-red %
0.5.3 $\lambda z.yz = y \in A \Rightarrow A$
% 0.4.2, 0.4.3, trans %
0.5.4 $\lambda z.I_0(yz) = y \in A \Rightarrow A$
]|
% 0.4, λ-congr %
0.6 $\lambda y.\lambda z.I_0(yz) = \lambda y.y \in (A \Rightarrow A) \Rightarrow A \Rightarrow A$
% 0.3, 0.5, 0.0, trans %
0.7 $BI_0 = I_1 \in (A \Rightarrow A) \Rightarrow A \Rightarrow A$
]|

Note that step 0 abbreviates the method used in example 2.8 for the introduction of abbreviations.

(End of example)

Exercise 2.8 Establish the following.

$$S(KI_0) = I_1 \in (A \Rightarrow A) \Rightarrow A \Rightarrow A$$

where $S = \lambda f.\lambda g.\lambda x.fx(gx) \in (A \Rightarrow A \Rightarrow A) \Rightarrow ((A \Rightarrow A) \Rightarrow A \Rightarrow A)$

and $K = \lambda x.\lambda y.x \in (A \Rightarrow A) \Rightarrow (A \Rightarrow A \Rightarrow A)$

(End of exercise)

Exercise 2.9 Assuming that $S \equiv \lambda f.\lambda g.\lambda x.fx(gx)$, $K \equiv \lambda x.\lambda y.x$, $B \equiv \lambda x.\lambda y.\lambda z.x(yz)$ and $I \equiv \lambda x.x$, what type assignments (to each occurrence of each variable) are sufficient to make sense of the following equalities :

(a) $S(KS) = B$
(b) $SKK = I$

(End of exercise)

The rule for $\forall$-computation is the obvious extension of the $\Rightarrow$-computation rule to dependent types.

$$\frac{\begin{array}{l}[\!| \ x \in P \\ \triangleright \ q(x) \in Q(x) \\ |\!] \\ p \in P\end{array}}{(\lambda x.q(x))p = q(p) \in Q(p)} \quad \forall\text{-computation}$$

Exercise 2.10 (For the very assiduous)
Establish the following :

$$\begin{array}{l}[\!| \ F \in A \Rightarrow U_1; G \in \forall(A, a.Fa \Rightarrow U_1) \\ \triangleright \ [\!| \ S = \lambda f.\lambda g.fx(gx) \in \forall(A, x.\forall(Fx, y.Gxy)) \Rightarrow \forall(\forall(A, x.Fx), h.\forall(A, x.Gx(hx))) \\ \quad \triangleright \ [\!| \ K_1 = \lambda x.\lambda y.x \in \forall(A, x.\forall(\forall(Fx, z.A), y.A)) \\ \quad\quad \triangleright \ [\!| \ K_2 = \lambda x.\lambda y.x \in \forall(A, x.\forall(Fx, y.A)) \\ \quad\quad\quad \triangleright \ SK_1K_2 = \lambda x.x \in \forall(A, x.A) \\ \quad\quad\quad |\!] \\ \quad\quad |\!] \\ \quad |\!] \\ |\!]\end{array}$$

(End of exercise)

(b) Splitting Pairs

The non-canonical objects associated with conjunction and existential quantification are objects of the form $\mathbf{split}(p, (x, y).q(x, y))$. Viewed operationally such a construct splits p into its two components; these are then bound to x and y in the computation of q. The formal statement of the computation rule is very straightforward because we need only say what it means to split a pair.

$$\frac{\begin{array}{l}[\!| \ x \in P \wedge Q \\ \triangleright \ R(x) \ \ type \\ |\!] \\ p \in P \\ q \in Q \\ [\!| \ y \in P; z \in Q \\ \triangleright \ r \in R(\langle y, z\rangle) \\ |\!]\end{array}}{\mathbf{split}(\langle p, q\rangle, (y, z).r(y, z)) = r(p, q) \in R(\langle p, q\rangle)} \quad \wedge\text{-computation}$$

Example 2.10

$$
\begin{array}{l}
[\![\ p \in A \wedge B \\
\triangleright\ \mathbf{split}(p, (a,b).\langle a,b\rangle) = p \in A \wedge B \\
]\!]
\end{array}
$$

Derivation

$$
\begin{array}{lll}
0.0 & [\![& p \in A \wedge B \\
0.1.0 & \triangleright & [\![\quad a' \in A; b' \in B \\
0.1.1.0 & & \triangleright \quad [\![\quad a \in A; b \in B \\
0.1.1.1 & & \qquad \triangleright \quad \langle a,b\rangle \in A \wedge B \\
 & & \qquad]\!] \\
 & & \qquad\quad \%\ 0.1.0,\ \wedge\text{-intro},\ 0.1.1,\ \wedge\text{-comp}\ \% \\
0.1.2 & & \qquad \mathbf{split}(\langle a',b'\rangle, (a,b).\langle a,b\rangle) = \langle a',b'\rangle \in A \wedge B \\
 & & \qquad\quad \%\ 0.1.2,\ =\text{-intro},\ =\text{-elim}\ \% \\
0.1.3 & & \qquad \mathrm{e} \in \mathbf{split}(\langle a',b'\rangle, (a,b).\langle a,b\rangle) \underset{A\wedge B}{=} \langle a',b'\rangle \\
 & &]\!] \\
 & & \quad\ \% \ 0.0,\ 0.1,\ \wedge\text{-elim}\ \% \\
0.2 & & \quad \mathrm{e} \in \mathbf{split}(p, (a,b).\langle a,b\rangle) \underset{A\wedge B}{=} p \\
 &]\!] &
\end{array}
$$

(End of example)

Exercise 2.11 (Cancellation)

Establish the following :

(a) $\forall(A, a.\forall(A, a'.\forall(B, b.\forall(B, b'.[(\langle a,b\rangle \underset{A\wedge B}{=} \langle a',b'\rangle) \Rightarrow (a \underset{A}{=} a')])))$

(a) $\forall(A, a.\forall(A, a'.\forall(B, b.\forall(B, b'.[\neg(\langle a,b\rangle \underset{A\wedge B}{=} \langle a',b'\rangle) \Rightarrow \neg(a \underset{A}{=} a' \wedge b \underset{B}{=} b')])))$

Hint: See exercise 2.7 for one half of this equivalence. For the other half make use of example 2.10, split-congruence and split-computation.

(End of exercise)

The $\exists$-computation rule generalises the $\wedge$-computation rule to dependent types.

$$
\begin{array}{l}
[\![\ w \in \exists(P, x.Q(x)) \\
\triangleright\ R(w)\ \ type \\
]\!] \\
p \in P \\
q \in Q(p) \\
[\![\ y \in P; z \in Q(y) \\
\triangleright\ s(y,z) \in R(\langle y,z\rangle) \\
]\!] \\
\hline
\mathbf{split}(\langle p,q\rangle, (y,z).s(y,z)) = s(p,q) \in R(\langle p,q\rangle)
\end{array}
\qquad \exists\text{-computation}
$$

Example 2.11

$$
\begin{array}{l}
[\![\ F \in A \Rightarrow U_1 \\
\triangleright\ \lambda p.\mathbf{split}(p, (a',b').b') \in \forall(\exists(A, a.Fa), p.F(\mathbf{split}(p, (a,b).a))) \\
]\!]
\end{array}
$$

The second component of a pair p in $\exists(A, a.Fa)$ is a proof that the first component has the property F.

Derivation

0.0 |[$F \in A \Rightarrow U_1$
0.1.0 ▷ |[$p \in \exists(A, a.Fa)$
0.1.1.0 ▷ |[$a' \in A; b' \in Fa'$
0.1.1.1.0 ▷ |[$a \in A; b \in Fa$
▷ % 0.1.1.1.0 %
0.1.1.1.1 $a \in A$
]|
% 0.1.1.0, ∃-intro, 0.1.1.1, ∃-comp %
0.1.1.2 $\mathbf{split}(\langle a', b'\rangle, (a, b).a) = a' \in A$
% 0.1.1.0, 0.1.1.2 refl, subst, type eq %
0.1.1.3 $b' \in F(\mathbf{split}(\langle a', b'\rangle, (a, b).a))$
]|
% 0.1.0, 0.1.1, ∃-elim %
0.1.2 $\mathbf{split}(p, (a', b').b') \in F(\mathbf{split}(p, (a, b).a))$
]|
]|

(End of example)

Example 2.12 (A conditional implication)

$$|[\ F \in A \Rightarrow U_1$$
$$\triangleright\ \forall(A, a.\forall(Fa, b.\forall(A, a'.\forall(Fa', b'.(\langle a, b\rangle \underset{\exists(A,x.Fx)}{=} \langle a', b'\rangle) \Rightarrow (b \underset{Fa}{=} b')))))$$
$$]|$$

As the reader will have realised, we have been omitting type judgements for some time. We give the full details of this example because it is the first example of a *conditional* implication, and hence the first example that makes full use of the $\Rightarrow$-formation rule in section 1.2(a).

Derivation

0.0 ||[$F \in A \Rightarrow U_1$

0.1.0 ▷ ||[$a \in A; b \in Fa; a' \in A; b' \in Fa'$

▷ % 0.1.0, $\exists$-intro %

0.1.1 $\langle a, b\rangle \in \exists(A, x.Fx)$

% 0.1.0, $\exists$-intro %

0.1.2 $\langle a', b'\rangle \in \exists(A, x.Fx)$

% 0.1.1, 0.1.2, =-form %

0.1.3 $\langle a, b\rangle \underset{\exists(A,x.Fx)}{=} \langle a', b'\rangle$ *type*

0.1.4.0 ||[$r \in \langle a, b\rangle \underset{\exists(A,x.Fx)}{=} \langle a', b'\rangle$

▷ % 0.1.4.0, exercise 2.8 %

0.1.4.1 $a = a' \in A$

% 0.0, 0.1.4.1, $\Rightarrow$-elim, U_1-elim, refl, 0.1.0, subst, type eq %

0.1.4.2 $b' \in Fa$

% 0.1.0, 0.1.4.2, =-form %

0.1.4.3 $b \underset{Fa}{=} b'$ *type*

]|

% 0.1.3, 0.1.4, $\Rightarrow$-form %

0.1.5 $(\langle a, b\rangle \underset{\exists(A,x.Fx)}{=} \langle a', b'\rangle) \Rightarrow (b \underset{Fa'}{=} b')$ *type*

0.1.6.0 ||[$r \in \langle a, b\rangle \underset{\exists(A,x.Fx)}{=} \langle a', b'\rangle$

▷ % 0.1.6.0, =-elim %

0.1.6.1 $\langle a, b\rangle = \langle a', b'\rangle \in \exists(A, x.Fx)$

0.1.6.2.0 ||[$c \in A; d \in Fc$

▷ % 0.1.6.2.0, example 2.11 %

0.1.6.2.1 $d \in F(\mathbf{split}(\langle c, d\rangle, (c', d').c'))$

]|

% 0.1.6.1, 0.1.6.2, split-congr %

0.1.6.3 $\mathbf{split}(\langle a, b\rangle, (c, d).d) = \mathbf{split}(\langle a', b'\rangle, (c, d).d) \in F(\mathbf{split}(\langle a, b\rangle, (c', d').c'))$

% 0.1.1, 0.1.6.2, $\exists$-comp %

0.1.6.4 $\mathbf{split}(\langle a, b\rangle, (c, d).d) = b \in F(\mathbf{split}(\langle a, b\rangle, (c', d').c'))$

% 0.1.2, 0.1.6.2, $\exists$-comp %

0.1.6.5 $\mathbf{split}(\langle a', b'\rangle, (c, d).d) = b' \in F(\mathbf{split}(\langle a, b\rangle, (c', d').c'))$

% similarly %

0.1.6.6 $\mathbf{split}(\langle a, b\rangle, (c', d').c') = a \in A$

% 0.1.6.3, 0.1.6.4, 0.1.6.5, 0.1.6.6, subst, type eq, trans, sym %

0.1.6.7 $b = b' \in Fa$

% 0.1.6.7, =-intro %

0.1.6.8 $\mathrm{e} \in b \underset{Fa}{=} b'$

]|

% 0.1.5, 0.1.6, $\Rightarrow$-intro %

0.1.7 $\lambda r.\mathrm{e} \in (\langle a, b\rangle \underset{\exists(A,x.Fx)}{=} \langle a', b'\rangle) \Rightarrow (b \underset{Fa}{=} b')$

]|

]|

The most interesting aspect of this derivation is step 0.1.4. It is impossible to make the judgement that $b \underset{Fa}{=} b'$ is a type except in a context in which $a \underset{A}{=} a'$ is provable. But to prove the latter given the assumption that $\langle a, b\rangle = \langle a', b'\rangle$ we are obliged to make use of

the computation rules. Hence our earlier remark about each of the judgement forms being inextricably bound together.

(End of example)

Exercise 2.12 Prove the following :

$$[\![\ F \in A \Rightarrow U_1$$
$$\triangleright\ \forall(\exists(A, a.Fa), p.snd\ p\ \underset{F(\overset{=}{fst\ p})}{}\ snd\ p)$$
$$]\!]$$

where *fst p* denotes $\mathbf{split}(p, (a, b).a)$ and *snd p* denotes $\mathbf{split}(p, (a, b).b)$.

This curious exercise (which may appear trivial at first sight) illustrates the way that type membership is expressed in the theory. A paraphrase of the theorem is that *snd p* has type $F(fst\ p)$ whenever p has type $\exists(A, a.Fa)$.

(End of exercise)

At last we come to the first and only example in Martin-Löf's celebrated paper [M-L].

Example 2.13 (Axiom of choice)

$$[\![\ F \in A \Rightarrow U_1; G \in \forall(A, x.Fx \Rightarrow U_1)$$
$$\triangleright\ \lambda f.\langle \lambda x.fst(fx), \lambda x.snd(fx)\rangle \in \forall(A, x.\exists(Fx, y.Gxy)) \Rightarrow \exists(\forall(A, x.Fx), h.\forall(A, x.Gx(hx)))$$
$$]\!]$$

(Cf., also, exercise 2.2(b).)

The two assumptions state that Fa is a (small) type whenever a is in A and Gab is a (small) type whenever a is in A and b is in Fa. In such a context the axiom of choice is valid. That is, if for all x in A it is known that there is a y in Fx such that Gxy, then there is a choice function h that given an object x of A constructs such an object. (Omitting types in the quantifiers the axiom reads : $\forall(x.\exists(y.Gxy)) \Rightarrow \exists(h.\forall(x.Gx(hx)))$.)

Derivation

0.0 |[$F \in A \Rightarrow U_1; G \in \forall(A, x.Fx \Rightarrow U_1)$

0.1.0 ▷ |[$f \in \forall(A, x.\exists(Fx, y.Gxy))$

0.1.1.0 ▷ |[$x \in A$

▷ % 0.1.0, 0.1.1.0, ∀-elim %

0.1.1.1 $fx \in \exists(Fx, y.Gxy)$

% 0.1.1.1, ∃-elim %

0.1.1.2 $fst(fx) \in Fx$

]|

% 0.1.1, ∀-intro %

0.1.2 $\lambda x.fst(fx) \in \forall(A, x.Fx)$

0.1.3.0 |[$h = \lambda x.fst(fx) \in \forall(A, x.Fx)$

0.1.3.1.0 ▷ |[$x \in A$

▷ % 0.1.1, 0.1.3.1.0, ∀-comp, subst %

0.1.3.1.1 $hx = fst(fx) \in Fx$

% 0.1.0, 0.1.3.1.0, ∀-elim %

0.1.3.1.2 $fx \in \exists(Fx, y.Gxy)$

% 0.0, 0.1.3.1.1, 0.1.3.1.2, example 2.11, subst %

0.1.3.1.3 $snd(fx) \in Gx(hx)$

]|

% 0.1.3.1, ∀-intro %

0.1.3.2 $\lambda x.snd(fx) \in \forall(A, x.Gx(hx))$

]|

% 0.1.3, ∀-intro(twice) %

0.1.4 $\lambda h.\lambda r.\lambda x.snd(fx)$
$\in \forall(\forall(A, x.Fx), h.\forall(h =_{\forall(A,x.Fx)} \lambda x.\mathbf{split}(fx, (y, z)y), r.\forall(A, x.Gx(hx))))$

% 0.1.2, 0.1.4, refl, ∀-comp %

0.1.5 $\lambda x.snd(fx) \in \forall(A, x.Gx((\lambda x.snd(fx))x))$

% 0.1.2, 0.1.5, ∃-intro %

0.1.6 $\langle \lambda x.fst(fx), \lambda x.snd(fx) \rangle \in \exists(\forall(A, x.Fx), h.\forall(A, x.Gx(hx)))$

]|

]|

Note (to myself) the introduction of the abbreviation h for the choice function $\lambda x.fst(fx)$. Such techniques should be encouraged in formal derivations but once again I find that the substitution rule in Martin-Löf's theory is not powerful enough for my purposes - see steps 0.1.4, 0.1.5.

(End of example)

(c) ∨-computation

There are two computation rules for **when** objects, one for each sort of canonical object.

$$\frac{\begin{array}{l}[\![\ x \in P \vee Q \\ \triangleright\ R(x)\ \ type \\ ]\!] \\ [\![\ y \in P \\ \triangleright\ r \in R(\mathbf{inl}\ y) \\ ]\!] \\ [\![\ z \in Q \\ \triangleright\ s \in R(\mathbf{inr}\ z) \\ ]\!] \\ p \in P \end{array}}{\mathbf{when}(\mathbf{inl}\ p, y.r(y), z.s(z)) = r(p) \in R(\mathbf{inl}\ p)} \quad \vee\text{-computation}$$

$$\frac{\begin{array}{l}[\![\ x \in P \vee Q \\ \triangleright\ R(x)\ \ type \\ ]\!] \\ [\![\ y \in P \\ \triangleright\ r \in R(\mathbf{inl}\ y) \\ ]\!] \\ [\![\ z \in Q \\ \triangleright\ s \in R(\mathbf{inr}\ z) \\ ]\!] \\ q \in Q \end{array}}{\mathbf{when}(\mathbf{inr}\ q, y.r(y), z.s(z)) = s(q) \in R(\mathbf{inr}\ q)} \quad \vee\text{-computation}$$

The operational understanding of **when** is that **when**(t,$y.r(y)$,$z.s(z)$) picks out either $r(y)$ or $s(z)$ depending on the form taken by t. If it has the form **inl** p then r is evaluated with the parameter y bound to p. On the other hand if it has the form **inr** q then s is evaluated with the parameter z bound to q.

Example 2.14

$$\forall(A \vee B, d.\mathbf{when}(d, a.\mathbf{inl}\ a, b.\mathbf{inr}\ b) \underset{A \vee B}{=} d)$$

Derivation

0.0 ||[$d \in A \vee B$

0.1.0 ▷ ||[$a' \in A$

0.1.1.0 ▷ ||[$a \in A$

▷ % 0.1.1.0, ∨-intro %

0.1.1.1 $\mathbf{inl}\ a \in A \vee B$

]||

0.1.2.0 ||[$b \in B$

▷ % 0.1.2.0, ∨-intro %

0.1.2.1 $\mathbf{inr}\ b \in A \vee B$

]||

% 0.1.0, ∨-intro, 0.1.1, 0.1.2, ∨-comp, =-intro %

0.1.3 $e \in \mathbf{when}(\mathbf{inl}\ a', a.\mathbf{inl}\ a, b.\mathbf{inr}\ b) \underset{A \vee B}{=} \mathbf{inl}\ a'$

]||

% similarly %

0.2.0 ||[$b' \in B$

0.2.1 ▷ $e \in \mathbf{when}(\mathbf{inr}\ b', a.\mathbf{inl}\ a, b.\mathbf{inr}\ b) \underset{A \vee B}{=} \mathbf{inr}\ b'$

]||

% 0.0, 0.1, 0.2, -comp, =-intro, =-elim %

0.3 $e \in \mathbf{when}(d, a.\mathbf{inl}\ a, b.\mathbf{inr}\ b) \underset{A \vee B}{=} d$

]||

Strictly it is necessary to prove that $\mathbf{when}(x, a.\mathbf{inl}\ a, b.\mathbf{inr}\ b) \underset{A \vee B}{=} x$ is a family of types indexed by $x \in A \vee B$. This step has been omitted.

(End of example)

Exercise 2.13 (Cancellation)

Establish the following :

(a) $\forall(A, a.\forall(A, a'.(\mathbf{inl}\ a \underset{A \vee B}{=} \mathbf{inl}\ a') \Longleftrightarrow (a \underset{A}{=} a')))$

(b) $\forall(A, a.\forall(A, a'.\neg(\mathbf{inl}\ a \underset{A \vee B}{=} \mathbf{inl}\ a') \Longleftrightarrow \neg(a \underset{A}{=} a')))$

(End of exercise)

At the end of section 1 I argued the case for a detailed study of algebraic properties. It seems appropriate to remark at this point that we are still unable to establish the judgement $\neg(\mathbf{inl}\ a \underset{A \vee B}{=} \mathbf{inr}\ b)$ whenever $a \in A$ and $b \in B$. Its derivation must wait until the universes have been introduced in section 3.

]||

2.6 In Retrospect

By now I expect that some readers will have been tempted to rush past the seemingly endless series of "trivial" examples and exercises. However, in spite of their simplicity, many are extremely important and worthy of deep reflection. I propose, therefore, to briefly review those that I find especially significant.

Example 2.2 and exercise 2.0 describe closure properties of conjunction and disjunction. (Roughly speaking, every conjunct is a pair and every disjunct is of the form **inl** a or **inl** b for some a or b.) The remarkable aspect of these examples is their use of the notion of a family of types. The reader will also have noted a pattern in their derivation which reflects a deeper pattern in the construction of the rules. This aspect of the theory is yet another that we postpone for later discussion.

The notion of a conditional implication was illustrated by example 2.12. This example is important because it shows that conditional expressions can be discussed without introducing a notion of undefined values. (Commonly, in computing and logic texts, the "undefined" value is a perfectly well-defined value that is confusingly given the name "undefined".) The example may well seem to be at variance with some of the properties discussed in section 1, for example the property (see exercise 1.2(a))

$$
\begin{array}{l}
[\![\ A, B, C \in U_1 \\
\triangleright\ [A \Rightarrow (B \Rightarrow C)] \Longleftrightarrow [B \Rightarrow (A \Rightarrow C)] \\
]\!]
\end{array}
$$

Is it a corollary of the above, for instance, that

$$
\begin{array}{l}
[\![\ F \in A \Rightarrow U_1; a, a' \in A; b \in Fa; b' \in Fa' \\
\triangleright\ [(a \underset{A}{=} a') \Rightarrow (b \underset{Fa}{=} b') \Rightarrow (\langle a, b\rangle \underset{\exists(A,x.Fx)}{=} \langle a', b'\rangle)] \\
\quad \Rightarrow [(b \underset{Fa}{=} b') \Rightarrow (a \underset{A}{=} a') \Rightarrow (\langle a, b\rangle \underset{\exists(A,x.Fx)}{=} \langle a', b'\rangle)] \\
]\!]
\end{array}
$$

The answer is no, because the context in exercise 1.2(a) specifies A, B and C to be variables in U_1, and there can thus be no dependance among them. Since we cannot establish the judgement $b \underset{Fa}{=} b' \in U_1$, exercise 1.2(a) is inapplicable.

Example 2.13 is Martin-Löf's favourite. But I have chosen to be much more formal in my presentation — not only in respect of providing much greater detail than Martin-Löf. In particular, I have proved the axiom of choice in a context in which $A \in U_1, F \in A \Rightarrow U_1$ and $G \in \forall(A, x.Fx \Rightarrow U_1)$. In his presentation, Martin-Löf makes the following assumptions

$$
\begin{array}{l}
A \ \ type \\
F(x) \ \ type\ [x \in A] \\
G(x, y) \ \ type\ [x \in A; y \in F(x)]
\end{array}
$$

But there is no mechanism within his formal system to make such assumptions! This points to a deficiency in his theory about which he has not been completely frank, but which we will return to in the section on universes.

Several examples, all involving the empty type, have been concerned with the (lack of) computational content in proof objects, and we have alluded to the subset type to be introduced later. Originally, I had intended to introduce the subset type at this point in the dicussion. However, I am currently of the opinion that the intrusion of proof objects with no computational content is a serious flaw in the theory that is only compensated to a limited extent by the subset type. I am therefore delaying the latter's introduction until I have better understood the problems myself.

References

[Ba] R.C. Backhouse "Overcoming the mismatch between programs and proofs".
Proceedings of workshop on Programming Logic, Marstrand, June 1-4 1987,
Chalmers Univ. of Technology, Dept. of Computer Sciences.

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

[Bl] E. Bishop *Foundations of Constructive Analysis*
McGraw-Hill, New York (1967).

[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,
Lecture Notes in Computer Science 173, 1-50 (1984).

[CH] Th. Coquand and G. Huet
"Constructions: A higher order proof system for mechanizing mathematics"
EUROCAL 85 (April 85) Linz, Austria.

[CR] R. Courant and H. Robbins
What is mathematics?
Oxford Univ. Press, London (1941).

[Ge] G. Gentzen "Investigations into logical deduction"
In *The Collected Papers of Gerhard Gentzen*, pp. 68-213, M.E. Szabo (Ed.),
North-Holland, Amsterdam (1969).

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

[Ho] C.A.R. Hoare "Notes on data structuring"
In *Structured Programming*, O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare (Eds.),
Academic Press (1972).

[Ja] M.A. Jackson *Principles of Program Design*
Academic Press (1975).

[M-L1] P. Martin-Löf "Constructive mathematics and computer programming"
Proc. 6th Int. Congress for Logic, Methodology and Philosophy of Science
(Eds. L.J. Cohen, J. Los, H. Pfeiffer and K.-P. Podewski) pp. 153-175,
North-Holland Publ. Co. (1982).

[M-L2] P. Martin-Löf "Intuitionistic Type Theory"
Notes by Giovanni Sambin of a series of lectures given in Padova, Juni 1980.

[Mi1] R. Milner "A theory of type polymorphism in programming"
J. Comp. Syst. Scs. *17*, pp. 348-375 (1977).

[Mi2] R. Milner "The standard ML core language"
Polymorphism II, 2 (October 1985).

[NPS] B. Nordström, K. Petersson and J. Smith
"An introduction to Martin-Löf's Type Theory"
Chalmers Inst. of Technology, Dept. of Computer Sciences, Midsummer 1986.

[Pe] K. Petersson "A programming system for type theory"
Memo 21, Programming Methodology Group, Chalmers Inst. of Technology,
Göteborg, Sweden (1982).

[PRL] The PRL Staff "Implementing Mathematics with the NuPRL Proof Development System"
Prentice-Hall Inc., Englewood Cliffs, New Jersey (1986)

[St] J. Stoy *Denotational Semantics*
The MIT Press, Cambride, Mass. (1977).

[Tu] D. Turner "A new implementation technique for applicative languages"
Software-Practice and Experience, *9*, 31-49 (1979).