Constructive Type Theory
A Perspective from Computing Science

Roland Backhouse *

12th November 1987

1 Introduction

Renewed interest in the formal connection between programs and proofs has recently been stimulated by Per Martin-Löf's formalisation of constructive mathematics. Although Martin-Löf is himself a philosopher rather than a computing scientist his theory has attracted considerable attention among theoretical computing scientists (at least in Europe!). My own contribution to this institute is to try to explain his theory from my own perspective as a computing scientist. I have two specific objectives. The first is to demonstrate how the theory increases our understanding of constructive mathematics and the relation between programs and proofs. The second is to convey to you some of my exuberance for his system as a formal system for performing program construction.

In order to put his work into perspective 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

*Subfaculteit Wiskunde en Informatica, Rijksuniversiteit Groningen, Postbus 800, 9700 AV GRONINGEN, The Netherlands

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 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 $* \to *$, 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 $int \times int \to int$ 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 $S \times S \to S$ 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 the time that I have available I shall try to provide an account of the contribution that the theory might make to the very practical task of program construction.

1.1 Propositions As Types

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. For the moment, however, I shall consider only one of these, the form

$$p \in P$$

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". Examples of such judgements (introduced now so that I can use them very shortly) are

$$0 \in \mathbb{N}$$

Proposition	*Type*	*Type Name*	*Example*
$P \Rightarrow Q$	$P \longrightarrow Q$	function space	$\lambda x.x \in A \Rightarrow A$ $\lambda x.\lambda y.x \in A \Rightarrow (B \Rightarrow A)$
$P \wedge Q$	$P \times Q$	cartesian product	$\lambda x.\langle x, x\rangle \in A \Rightarrow (A \wedge A)$ $\lambda y.\mathbf{fst}\ y \in (A \wedge B) \Rightarrow A$
$P \vee Q$	$P + Q$	disjoint sum	$\lambda x.\mathbf{inl}\ x \in A \Rightarrow (A \vee B)$
$\exists(P, x.Q(x))$	$\Sigma(P, x.Q(x))$	dependent product	$\langle \mathbb{N}, 0\rangle \in \exists(U1, A.A)$ $\langle \mathbb{N}, \lambda x.x\rangle \in \exists(U1, A.A \Rightarrow A)$
$\forall(P, x.Q(x))$	$\Pi(P, x.Q(x))$	dependent function space	$\lambda A.\lambda x.x \in \forall(U1, A.A \Rightarrow A)$
$\neg P$	$P \longrightarrow \emptyset$		$\lambda f.f\emptyset \in \neg\forall(U1, A.A)$

Table 1: Propositions as types.

meaning "0 has the type natural number"

$$red \in \{red, white, blue\}$$

meaning "red is an element of the enumerated type $\{red, white, blue\}$"

$$\mathbb{N} \in U_1,$$

and

$$\emptyset \in U_1.$$

Here U_1 stands for a universe of types, the first in a hierarchy of universes. Thus the judgement $\mathbb{N} \in U_1$ reads that the set of natural numbers is an element of the first universe, and the judgement $\emptyset \in U_1$ reads that the empty type is also such an element.

In "intuitionistic" or "constructive" logic the judgement form $p \in P$ 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. In other words proposition P is identified with the set (or "type") of its proofs. This is the idea generally attributed to Curry and Howard and called the principle of propositions-as-types. Table 1 illustrates the principle.[1]

In constructive mathematics, a proof of $P \Rightarrow Q$ is a method of proving Q given a proof of P. Thus $P \Rightarrow Q$ is identified with the type $P \longrightarrow Q$ of (total)

[1]The notation we are using for λ-expressions and function application is the conventional one [Chu,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 to the right 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 $P \Rightarrow (Q \Rightarrow R)$.

functions from the type P into the type Q. Assuming that A is a proposition, an elementary example would be the proposition $A \Rightarrow A$. A proof of $A \Rightarrow A$ is a method of constructing a proof of A given a proof A. Such a method would be the identity function of A, $\lambda x.x$, since this is a function that, given an object of A, returns the same object of A. 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 propositions, 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 $A \Rightarrow (B \Rightarrow A)$.

To prove $P \wedge Q$ constructively it is necessary to exhibit a proof of P and to exhibit a proof of Q. Thus the proposition $P \wedge Q$ is identified with the cartesian product, $P \times Q$, of the types P and Q. That is, $P \times Q$ is the type of all pairs $\langle x, y \rangle$ where x has type P and y has type Q. For example, assuming that A and B are propositions, the proposition $(A \wedge B) \Rightarrow A$ is proved constructively as follows. 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 **fst** that projects an object of $A \wedge B$ onto its first component. (The function **fst** is not a primitive of type theory but is expressed as $\lambda p.\mathbf{split}(p, (x, y).x)$. In general $\mathbf{split}(p, (x, y).e)$ splits a pair p into its two components and evaluates the expression e with the variables x and y bound to the respective components. Thus $\mathbf{split}(p, (x, y).x)$ splits p into its two components and then evaluates the expression x with x bound to the first component, i.e. it evaluates the first component.)

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 **inl** x or **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 $A \Rightarrow A \vee B$ and $A \vee B \Rightarrow B \vee A$. The proposition $A \Rightarrow A \vee B$ is proved by the function $\lambda x.\mathbf{inl}\ x$ that injects an argument x of type A into the left operand of $A \vee B$. The proposition $A \vee B \Rightarrow B \vee A$ is proved by the function $\lambda x.\mathbf{when}(x, y.\mathbf{inr}\ y, z.\mathbf{inl}\ z)$.[2] In general the construct $\mathbf{when}(x, y.e, z.f)$ is evaluated as follows. The argument x is evaluated; if it takes the form **inl** a then the expression e is evaluated with the variable y bound to a; if x takes the form **inr** b then the expression f is evaluated with the variable z bound to b. Thus $\mathbf{when}(x, y.\mathbf{inr}\ y, z.\mathbf{inl}\ z)$ has the effect of transforming a value of the form **inr** b into **inl** b and vice-versa.

The notation $\forall(P, x.Q(x))$ denotes a universal quantification. We prefer

[2]Later the name "$\vee$-*elim*" is used instead of "**when**". The latter is used for the moment in order to suggest its operational meaning. Similarly, "$\wedge$-*elim*" should have been used instead of "**split**".

this notation to the more conventional $(\forall x \in P)Q(x)$ because it makes clear the scope of the binding of the variable 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 a proof of $Q(p)$. 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. As an example the polymorphic identity function $\lambda A.\lambda x.x$ is a proof of the proposition $\forall(U_1, A.A \Rightarrow A)$.

The notion of *dependent* function space is often severely restricted if not completely unknown in conventional programming languages even though the idea is commonplace in the space of real world problems. Examples would include the type of functions that input a number n and then return a number that is at least n, the type of functions that input a number n and then return a function that inputs an array of size n and outputs its length, or a function that inputs the details of a person and then, depending on whether the person is living or dead, outputs that person's employment status or details of the person's estate.

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* product because the type of the second component, q, in a pair $\langle p, q\rangle$ in the type depends on the first component, p. For 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 type and a is an object of that type. (Thus the proposition is interpreted as the statement "there is a type that is provable", or "there is a type that is non-empty".) The pair $\langle \mathbb{N}, 0\rangle$ is an object of $\exists(U_1, A.A)$ since $\mathbb{N}$ is an element of U_1 and 0 is an element of $\mathbb{N}$. Two further examples are $\langle \{red, white, blue\}, red\rangle$ and $\langle \mathbb{N} \Rightarrow \mathbb{N}, \lambda x.x\rangle$.

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. Indeed algebras are good examples of the need for dependent types. A semigroup, for example, is a set S together with an associative binary operation on S. Thus a semigroup is a pair in which the type of the second component depends on the value of the first component.

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. 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 proposition (in U_1) is provable, or not every 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 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 $\lambda f.f\emptyset$.

Some further examples of provable propositions may help to clarify the nature of constructive proof.

(1) $$\lambda f.\lambda g.\lambda x.g(fx) \in (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$$

Functional composition proves the transitivity of implication.

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

The propositional equivalent of currying.

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

Uncurrying.

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

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

1.2 An example derivation

Martin-Löf's theory is defined by a number of natural deduction style [Ge] inference rules. For the purposes of illustration we consider just four rules for the moment. These are (simplified forms of) the rules for function introduction and elimination, and two rules for $\vee$-introduction.

$$\frac{\begin{array}{l}|[\ \ x \in A \\ \triangleright\ \ f(x) \in B \\ ]|\end{array}}{\lambda x.f(x) \in A \Rightarrow B} \qquad \Rightarrow\text{-introduction}$$

$$\frac{\begin{array}{l}a \in A \\ f \in A \Rightarrow B\end{array}}{fa \in B} \qquad \Rightarrow\text{-elimination}$$

$$\frac{a \in A}{\mathbf{inl}\ a \in A \vee B} \qquad \frac{b \in B}{\mathbf{inr}\ b \in A \vee B} \qquad \vee\text{-introduction}$$

The second of these rules ($\Rightarrow$-elimination) introduces the least amount of new notation and so is the easiest to begin with. It can be read in two senses — in a logical sense and in a computational sense. In a logical sense the rule states that if a is a proof of A and f is a proof of $A \Rightarrow B$, i.e. a method of going from a proof of A to a proof of B, then fa — the result of applying the method f to the given proof a — is a proof of B. In a computational sense it states that if a has type A and f is a function from A to B then fa, the result of applying the function f to a, has type B.

The first rule ($\Rightarrow$-introduction) says how functions can be constructed. It has one premise — a so-called "hypothetical premise." Hypothetical judgements play an extremely important role in the theory and are indicated by the use of scope brackets ("$|[$" and "$]|$"). (This notation, borrowed from the book by Dijkstra and Feijen [DF], is not used by Martin-Löf but is one that I, personally, have introduced in my own accounts of the theory. It is likely that a number of my colleagues in this institute will also use the same notation for their own purposes. Although there may be some differences in interpretation you will not go far wrong if you consider all uses as meaning the same.) A hypothetical judgement has two parts, first a number of assumptions and then a number of conclusions that can be made in the context of those assumptions. In the notation used here the assumptions are separated from the conclusions by the symbol "$\triangleright$". In a logical sense the rule may be read as "if assuming that x is a proof of A it is possible to construct a proof $f(x)$ of B then $\lambda x.f(x)$ is a proof of $A \Rightarrow B$." In a computational sense the rule is read differently. "If in a context in which x is an object of type A the object $f(x)$ has type B then the function $\lambda x.f(x)$ is an object of type $A \Rightarrow B$."

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

The last two rules say how to construct a proof of a disjunction or, equally, how to construct an element of a disjoint sum. To prove $A \vee B$ we exhibit a proof of A and tag it with the constant **inl**, or we exhibit a proof of B and tag it with the constant **inr**. Put another way, an element of the disjoint sum of types A and B is an element of A tagged by **inl** or an element of B tagged by **inr**. The constants **inl** and **inr** are called *injection functions* and stand for **in**ject **l**eft and **in**ject **r**ight, respectively.

We use these rules in the proof of the proposition

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

Example 1.1

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

Derivation

0.0 |[$f \in [(A \vee (A \Rightarrow B)) \Rightarrow B]$

0.1.0 ▷ |[$x \in A$

▷ % 0.1.0,**inl**-introduction %

0.1.1 $\mathbf{inl}\ x \in A \vee (A \Rightarrow B)$

% 0.0,0.1.1, ⇒-elimination %

0.1.2 $f(\mathbf{inl}\ x) \in B$

]|

% 0.1.0, 0.1.2, ⇒-introduction %

0.1 $\lambda x.f(\mathbf{inl}\ x) \in A \Rightarrow B$

% 0.1, **inr**-introduction %

0.2 $\mathbf{inr}(\lambda x.f(\mathbf{inl}\ x)) \in A \vee (A \Rightarrow B)$

% 0.0, 0.2, ⇒-elimination %

0.3 $f(\mathbf{inr}(\lambda x.f(\mathbf{inl}\ x))) \in B$

]|

% 0.0, 0.3, ⇒-introduction %

1 $\lambda f.f(\mathbf{inr}(\lambda x.f(\mathbf{inl}\ x))) \in [(A \vee (A \Rightarrow B)) \Rightarrow B] \Rightarrow B$

(End of derivation)

There is an ulterior motive for presenting the above as an example of proof derivation in constructive mathematics, namely to explain the role of the law of the excluded middle. It is commonly — misleadingly — stated that the latter law is not valid in constructive mathematics. This is not so. What is valid is that there is no general method for establishing the law for an arbitrary proposition; a theory obtained by adding the law of the excluded middle to Type Theory would not be inconsistent. Indeed it is the case that the law of the excluded middle can never be refuted in constructive mathematics. Evidence for this is obtained from the above example. Specifically, by substituting $\emptyset$ for B and replacing $P \Rightarrow \emptyset$ by $\neg P$ 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 form $\neg\neg P$ is of interest because it asserts that P cannot be refuted. Other examples of propositions that are classically valid but cannot be generally established in constructive mathematics are the following:

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

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

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

For each such proposition, P, it is however the case that $\neg\neg P$ can be proven constructively. Indeed it is a theorem attributed by Kleene [Kl] to Glivenko [Gl] that if P is any tautology of the classical propositional calculus then the proposition $\neg\neg P$ is always constructively valid. For one method of modelling classical reasoning in a formal implementation of a constructive theory you are referred to [CH].

2 The Structure of the Rules

The programmer is, in his everyday activities, a user of formal systems — operating systems, text-processing systems and programming systems. The computing scientist is therefore, in his everyday activities, concerned with the construction and analysis of formal systems. What criteria should we use to assess a formal system? What is it that distinguishes an "elegant" formal system from an "inelegant" formal system? Certainly there have been many formalisations of constructive mathematics but none has gained as much acclaim among the computing scientist community as that of Per Martin-Löf. I believe that it is because his system exhibits a certain elegance that others lack.

On first encounter, however, the universal reaction among computing scientists appears to be that the theory is formidable. Indeed, several have specifically referred to the overwhelming number of rules in the theory. On closer examination, however, the theory betrays a rich structure — a structure that is much deeper than the superficial observation that types are defined by introduction, elimination and computation rules. Once recognised this structure considerably reduces the burden of understanding. The aim of this lecture is, therefore, to convey that structure to you.

There is a very practical reason for wanting to recognise the inherent structure of the formal system. As programmers using a typed programming language we are strongly encouraged to introduce and exploit our own type structures. Such declared data types are intended to reflect the structure of the given data and are in turn reflected in the structure of the programs that we write [see for example Ja]. Any formalisation of constructive reasoning should also strongly encourage the introduction of new type structures, but of course in a disciplined way. That his theory is already open to extension is a fact that was clearly intended by Martin-Löf. Indeed, it is a fact that has been exploited by

several individuals; Nordström, Petersson and Smith [NPS] have extended the theory to include lists, they and Constable et al. [Co] have added subset types and Constable et al. have introduced quotient types, Nordström has introduced multi-level functions [No], Chisholm has introduced a very special-purpose type of tree structure [Chi1] and Dyckhoff [Dyc] has defined the type of categories.

(Objections to such extensions can be made on the grounds that they can always be encoded within the existing theory, in particular using the W-type [Dyb], because they add to the complexity of the theory and because they might undermine the quality of the theory even to the extent of introducing inconsistencies. The experiences and arguments of others have convinced me that this view is wrong. In the context of this lecture, however, my main purpose is not to argue this view but to elucidate the structure of the rules as presented by Martin-Löf.)

The rules defining individual type constructors can be divided into five sets.

1. The formation rule.
2. The introduction rules.
3. An elimination rule.
4. Computation rules.
5. Congruence rules.

The formation rule specifies how a type constructor may be parameterised by other types; the introduction rules say how to form elements of the type and the elimination rule says how to reason about elements of the type (or equally since reasoning is constructive how to construct functions defined over the elements of the type). The elimination rule associates with the type constructor a so-called non-canonical object form; the computation rules then prescribe how to evaluate instances of this form. Finally, the congruence rules express substitutivity and extensionality properties. I shall not have time to discuss the latter rules; in any case their formulation is relatively straightforward.

The main contribution that we make here is to describe a scheme for computing the elimination rule and computation rules for a newly introduced type constructor. In other words, we show that it suffices to provide the type formation rule and the introduction rules for a new type constructor; together these provide sufficient information from which the remaining details can be deduced. The significance of this result is that it has the twin benefits of reducing the burden of understanding and the burden of definition. It reduces the burden of understanding since we now need to understand only the formation and introduction rules and the general scheme for inferring the remaining rules. Conversely, the burden of definition is reduced since it suffices to state the formation and introduction rules, the others being inferred automatically.

The method of inferring the elimination rule from the introduction rules is described by way of examples rather than formally, although a formal method does indeed underlie our descriptions and should be evident.

2.1 Lists

The list type constructor should be familiar. The formation rule and two introduction rules are as follows.

$$\frac{A\ type}{List(A)\ type}$$ List formation

$$\frac{A\ type}{[\,] \in List(A)}$$ [ ]-introduction

$$\frac{\begin{array}{l}A\ type\\ a \in A\\ l \in List(A)\end{array}}{a : l \in List(A)}$$:-introduction

It is normal to omit the premises of the formation rule from the premises of the introduction rules. Thus the premise "*A type*" would normally be omitted from the [ ]- and :-introduction rules above. We shall follow the same practice in the remainder of this discussion.

The (single) elimination rule for a given type constructor performs two functions: it says how to reason about objects of the type and it says how to define functions over objects of the type. (Because proofs are interpreted constructively these amount to the same thing.) The first premise (excluding the premises of the formation rule) of the elimination rule for type constructor Θ is therefore the statement that C, say, is a family of types indexed by objects of Θ. In other words C is postulated to be a property of objects of type Θ. The introduction rules represent the only way that canonical objects of the type may be constructed; so, in order to show that property C holds for an arbitrary object of type Θ, it suffices to show that it holds for each of the different sorts of canonical objects. There is thus one premise in the elimination rule for each of the introduction rules. Moreover the premises of an introduction rule become assumptions in the corresponding premise of the elimination rule.

In the case of lists there are just two sorts of canonical element, the empty list and composite lists consisting of a head element and a tail list. In order to prove that a property C is true of an arbitrary list we thus have to show that it is true of the empty list and of composite lists. Equally, to define a function over lists it suffices to define its value on the empty list and its value when applied to a composite list. The elimination rule is therefore as follows.

$$\frac{\begin{array}{l}|[\ \ w \in \mathit{List}(A)\\ \triangleright\ \ C(w)\ \mathit{type}\\ ]|\\ x \in \mathit{List}(A)\\ y \in C([\,])\\ |[\ \ a \in A; l \in \mathit{List}(A); h \in C(l)\\ \triangleright\ \ z(a,l,h) \in C(a:l)\\ ]|\end{array}}{\mathit{List\text{-}elim}(x,y,z) \in C(x)}\ \text{List-elimination}$$

In this rule the third premise is the one corresponding to [ ]-introduction; it is not hypothetical since apart from the premises of List formation there are no premises in the [ ]-introduction rule. The fourth premise corresponds to the :-introduction rule; it is hypothetical since the :-introduction rule has two premises in addition to the premises of List formation. To emphasise the way in which the premises of the introduction rule become assumptions of the corresponding premise in the elimination rule we have used the same symbols, a and l in the :-introduction rule and in the elimination rule.

Note that there is an additional assumption ("$h \in C(l)$") in the elimination rule arising from the fact that l is a recursive introduction variable.

The computation rules for a type introduce a third judgement form about which we need to make some preparatory remarks before going into the details of the computation rules for lists. The judgement form is

$$p = q \in P$$

and is read as "p and q are equal objects within the type P". Thus implicit in such a judgement are the judgements that p is an object of P and that q is an object of P.

Computation in the theory is lazy. That is, to evaluate an expression like $\mathit{List\text{-}elim}(\ldots)$ the first parameter is evaluated to its canonical form and then further evaluation involving the other parameters takes place. Since the introduction rules specify the only forms that the canonical objects of a type can take it suffices to provide a computation rule corresponding to each of the introduction rules. For the *List* type constructor we must therefore explain how to evaluate expressions of the form $\mathit{List\text{-}elim}([\,],\ldots)$ and of the form $\mathit{List\text{-}elim}(a:l,\ldots)$. We do so by replacing the premise "$x \in \mathit{List}(A)$" in the List elimination rule by the premises of the introduction rule. Taking first the [ ]-introduction we obtain the following computation rule.

$$\frac{\begin{array}{l}|[\ \ w \in List(A)\\ \triangleright\ \ C(w)\ type\\ ]|\\ y \in C([\,])\\ |[\ \ a \in A; l \in List(A); h \in C(l)\\ \triangleright\ \ z(a,l,h) \in C(a:l)\\ ]|\end{array}}{List\text{-}elim([\,], y, z) = y \in C([\,])}\ [\,]\text{-computation}$$

Since there are no premises in the [ ] introduction rule the effect of the replacement is simply to reduce the number of premises by one. The conclusion of the rule is also straightforward to see. Note the parameter to the elimination hypothesis C in the conclusion.

The computation rule for composite lists is a little more difficult to understand. As before the premise "$x \in List(A)$" in the elimination rule is replaced this time by the premises of the :-introduction rule. The construction of the conclusion of the rule is guided by its type part, viz. $C(a : l)$. The right side of the equality must be an object of this type. But the last premise of the List elimination rule tells us how to construct such an object: we have to exhibit objects a, l and h of appropriate type and, having done so, the expression $z(a, l, h)$ has type $C(a : l)$. The type of h is $C(l)$; to construct something of this type given that l has type $List(A)$ we would use List elimination. Thus we obtain the following rule.

$$\frac{\begin{array}{l}|[\ \ w \in List(A)\\ \triangleright\ \ C(w)\ type\\ ]|\\ a \in A\\ l \in List(A)\\ y \in C([\,])\\ |[\ \ a \in A; l \in List(A); h \in C(l)\\ \triangleright\ \ z(a,l,h) \in C(a:l)\\ ]|\end{array}}{List\text{-}elim(a : l, y, z) = z(a, l, List\text{-}elim(l, y, z)) \in C(a : l)}\ :\text{-computation}$$

One final comment should be made about the computation rules to avoid misunderstanding. The two rules above should be regarded as left-to-right rewrite rules for the purposes of evaluating an expression involving *List-elim*. As such the rules involve a recursive computation. The number of recursive evaluations of *List-elim* may however be smaller than the length of the given list — this occurs for example when the expression $z(a, l, h)$ contains no occurrences of the variable h. This is what is meant by saying that evaluation

is "lazy." As a consequence an expression may well contain occurrences of the constant $\emptyset$-*elim* — discussed in section 2.3 and for which no computation rules are given — without evaluation of the expression being in any way divergent.

2.2 Disjoint sums

We may now return to the disjoint sum type whose introduction rules were presented in section 1.2. Since there are two introduction rules there are four premises in the elimination rule — the two standard premises which postulate the existence of a family of types C and an object of the type, and one premise for each introduction rule.

$$\frac{\begin{array}{l}|[\ \ w \in A \vee B \\ \triangleright\ \ C(w)\ \mathit{type} \\ ]| \\ d \in A \vee B \\ |[\ \ a \in A \\ \triangleright\ \ e(a) \in C(\mathbf{inl}\ a) \\ ]| \\ |[\ \ b \in B \\ \triangleright\ \ f(b) \in C(\mathbf{inr}\ b) \\ ]|\end{array}}{\vee\text{-}\mathit{elim}(d, a.e(a), b.f(b)) \in C(d)}\ \vee\text{-elimination}$$

Note how the premises of the introduction rules become assumptions in the corresponding premises of the elimination rule. Note also the parameterisation of C in each of the premises.

There are two computation rules for $\vee$-*elim* objects, one for each sort of canonical object.

$$\frac{\begin{array}{l}|[\ \ w \in A \vee B \\ \triangleright\ \ C(w)\ \mathit{type} \\ ]| \\ a \in A \\ |[\ \ a \in A \\ \triangleright\ \ e(a) \in C(\mathbf{inl}\ a) \\ ]| \\ |[\ \ b \in B \\ \triangleright\ \ f(b) \in C(\mathbf{inr}\ b) \\ ]|\end{array}}{\vee\text{-}\mathit{elim}(\mathbf{inl}\ a, a.e(a), b.f(b)) = e(a) \in C(\mathbf{inl}\ a)}\ \mathbf{inl}\text{-computation}$$

$$\frac{\begin{array}{l}|[\ \ w \in A \vee B\\ \triangleright\ \ C(w)\ \mathit{type}\\ ]|\\ b \in B\\ |[\ \ a \in A\\ \triangleright\ \ e(a) \in C(\mathbf{inl}\ a)\\ ]|\\ |[\ \ b \in B\\ \triangleright\ \ f(b) \in C(\mathbf{inr}\ b)\\ ]|\end{array}}{\vee\text{-}\mathit{elim}(\mathbf{inr}\ b, a.e(a), b.f(b)) = f(b) \in C(\mathbf{inr}\ b)}\ \mathbf{inr}\text{-computation}$$

The operational understanding of $\vee$-$elim$ is that $\vee$-$elim(t,a.e(a),b.f(b))$ picks out either $e(a)$ or $f(b)$ depending on the form taken by t. If it has the form **inl** p then e is evaluated with the parameter a bound to p. On the other hand if it has the form **inr** q then f is evaluated with the parameter b bound to q.

2.3 The empty set

It is always instructive to consider extreme cases. Let us therefore consider the empty type. The formation rule is just the axiom:

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

There are no introduction rules for the empty type (since it would be absurd to construct an element of the empty type). Thus there are no premises in the elimination rule other than the standard ones.

$$\frac{\begin{array}{l}|[\ \ w \in \emptyset\\ \triangleright\ \ C(w)\ \mathit{type}\\ ]|\\ r \in \emptyset\end{array}}{\emptyset\text{-}\mathit{elim}(r) \in C(r)}\ \emptyset\text{-elimination}$$

This rule is easily recognised as the absurdity rule — if it is possible to establish an absurdity then it is possible to establish any proposition whatever.

Since there are no introduction rules there are no computation rules. The object $\emptyset$-$elim(r)$ should thus be considered as a divergent computation.

2.4 Finite Sets

Suppose we wish to define a type constructor $\Im$ such that $\Im(A)$ is the type of finite subsets of A.[3] Any such subset can be constructed by listing its elements. Conversely any list of elements of A may be regarded as a finite subset of A provided that we disregard the order of the elements and repeated occurrences of the same element. $\Im(A)$ is thus the quotient of $List(A)$ with respect to the equivalence relation that defines two lists as equal if they have the same elements independent of order and number of repeated occurrences.

We define the type constructor $\Im$ by adding to the introduction rules for List two additional rules defining the above equivalence. In full the rules are as follows.

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

$$\frac{}{\phi \in \Im(A)}\ \phi\text{-introduction}$$

$$\frac{a \in A \qquad s \in \Im(A)}{a; s \in \Im(A)}\ ;\text{-introduction}$$

$$\frac{a \in A \qquad s \in \Im(A)}{a; a; s = a; s \in \Im(A)}\ \text{repetition}$$

$$\frac{a \in A \qquad b \in A \qquad s \in \Im(A)}{a; b; s = b; a; s \in \Im(A)}\ \text{order}$$

How should we construct the elimination rule for $\Im$? The best way to begin is to view the rule as a method of defining a function over objects of the type. If a function is to be truly a function then it must give equal values when applied to equal objects. Looking at it from the point of view of proofs, a proof that an object has some property must be independent of the way the object was constructed. Thus the $\Im$-elimination rule is constructed like the List-elimination

[3] The material in this section is due to [Chi2] to whom the readers are referred for further discussion.

rule but with two additional premises, one corresponding to the repetition rule and the other corresponding to the order rule.

$$
\begin{array}{l}
|[\ \ w \in \Im(A) \\
\triangleright\ \ C(w)\ type \\
]| \\
x \in \Im(A) \\
y \in C(\phi) \\
|[\ \ a \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ \ z(a,s,h) \in C(a;s) \\
]| \\
|[\ \ a \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ \ z(a,\ a; s,\ z(a,s,h)) = z(a,s,h) \in C(a;s) \\
]| \\
|[\ \ a \in A; b \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ \ z(a,\ b; s,\ z(b,s,h)) = z(b,\ a; s,\ z(a,s,h)) \in C(a;b;s) \\
]| \\
\hline
\Im\text{-}elim(x,y,z) \in C(x)
\end{array}
\qquad \Im\text{-elimination}
$$

The premise corresponding to the repetition rule

$$
\begin{array}{l}
|[\ \ a \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ \ z(a,\ a; s,\ z(a,s,h)) = z(a,s,h) \in C(a;s) \\
]|
\end{array}
$$

is constructed as follows. The assumptions are derived from the premises of the repetition rule as in our discussion of lists. The judgement asserts that the proof object of $C(a;s)$ is the same whether we choose to evaluate it from $a;s$ or $a;a;s$. In the former case we evaluate $z(a,s,h)$ and in the latter case we evaluate $z(a,\ a;s,\ z(a,s,h))$.

The premise corresponding to the order rule

$$
\begin{array}{l}
|[\ \ a,b \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ \ z(a,\ b; s,\ z(b,s,h)) = z(b,\ a; s,\ z(a,s,h)) \in C(a;b;s) \\
]|
\end{array}
$$

is constructed similarly.

2.5 Polynomials over $\{0,1\}$

Consider now the representation of numbers in binary form. A binary numeral is a list of 1's and 0's in which leading 0's are insignificant. Thus $11 = 011 = 0011$ and so on. A binary numeral is, however, one particular interpretation of such a list. More generally we may regard such a list as denoting a polynomial; thus, 11 denotes $1 \times x + 1$. Using Λ to denote the empty list we can define a

type, called P say, of lists of 0's and 1's in which leading 0's are insignificant as follows.

$$\frac{}{\Lambda \in P} \qquad \Lambda\text{-introduction}$$

$$\frac{p \in P}{p0 \in P} \qquad 0\text{-introduction}$$

$$\frac{p \in P}{p1 \in P} \qquad 1\text{-introduction}$$

$$\frac{}{\Lambda 0 = \Lambda \in P} \qquad \text{leading zeroes}$$

Given these four introduction rules the elimination rule for P has four premises. The four premises state that to define a function over P it is necessary to consider three cases — the case where the argument is Λ, the case where it is of the form $p0$ and the case where it is of the form $p1$ — and furthermore it is necessary to show that the insignificance of leading zeroes is respected. Specifically, we have the following rule.

$$\frac{\begin{array}{l}|[\ \ w \in P \\ \triangleright\ \ C(w)\ type \\ ]| \\ x \in P \\ y_1 \in C(\Lambda) \\ |[\ \ p \in P; h \in C(p) \\ \triangleright\ \ y_2(p,h) \in C(p0) \\ ]| \\ |[\ \ p \in P; h \in C(p) \\ \triangleright\ \ y_3(p,h) \in C(p1) \\ ]| \\ y_2(\Lambda, y_1) = y_1 \in C(\Lambda)\end{array}}{P\text{-}elim(x, y_1, y_2, y_3) \in C(x)} \qquad P\text{-elimination}$$

3 The Boyer-Moore Majority-Vote Algorithm

3.1 Preliminary remarks

This lecture is concerned with examining the relationship between the heuristics used in inductive proof and the heuristics used in the development of loop invariants [Ba1,DF,Gr] in algorithm design. Before we do so it is necessary to introduce two additional type structures, the natural numbers and the subset type [Co,NP,Pe2]. Given the discussion in the previous lecture the type of natural numbers is easy to explain. There are just two introduction rules, the first asserting that 0 is a natural number, and the second asserting that the successor of m is a natural number whenever m is a natural number.

$$\frac{}{0 \in \mathbb{N}} \qquad \text{0-introduction}$$

$$\frac{m \in \mathbb{N}}{m + 1 \in \mathbb{N}} \qquad +1\text{-introduction}$$

The elimination rule for natural numbers is the familiar rule of simple mathematical induction. We leave as an exercise the reconstruction of the elimination and computation rules.

The subset type is less straightforward since it introduces a new concept, that of information loss. Quite early on in the application of the theory to computing science it was recognised that some proof objects have no computational content. Proof objects that witness equalities are the most obvious example, to which we can also add proof objects of negations and other propositions built with these two as basis. For this reason the subset type was introduced into the theory. The subset type is like the existential or Σ-type. To construct an object of the latter we have to construct a pair.

$$\frac{\begin{array}{l} a \in A \\ b \in B(a) \end{array}}{\langle a, b \rangle \in \exists(A, x.B(x))} \qquad \exists\text{-introduction}$$

In the subset type the information contained in the proof of the second component is lost (the reason being that it very often carries no computational content).

$$\frac{\begin{array}{l} a \in A \\ B(a)\ \mathbf{true} \end{array}}{a \in Set(A, x.B(x))} \qquad \text{Subset introduction}$$

The judgement form P **true** means that P is a proposition that has been established to be true. This judgement form offers a possible mechanism for integrating other formal proof systems with that of type theory. One of the rules for establishing such a judgement should be, of course, that if P is constructively true then P is true.

$$\frac{p \in P}{P \textbf{ true}} \qquad \text{Constructive truth}$$

There is however no reason why one should not also allow instances of the law of the excluded middle to be **true** judgements.

$$\frac{}{P \vee \neg P \textbf{ true}} \qquad \text{excluded middle}$$

The elimination rule for subset types is however a little harder to use since the constructive evidence is unavailable.

$$\frac{\begin{array}{l}|[\ \ w \in Set(A, x.B(x)) \\ \triangleright\ \ C(w)\ type \\ ]| \\ x \in Set(A, x.B(x)) \\ |[\ \ a \in A;\ \ B(a) \textbf{ true} \\ \triangleright\ \ c(a) \in C(a) \\ ]|\end{array}}{c(x) \in C(x)} \qquad \text{Subset elimination}$$

3.2 Problem statement

The problem we use to illustrate algorithm development in Martin-Löf's theory of types is called the majority-vote problem. It may briefly be described as determining whether or not one of the candidates in a ballot has received a majority of the votes. The solution on which our development is based is described in [MG] and is attributed by them to R. Boyer and J.S. Moore.

Let us suppose that the number of votes cast in a ballot is n and that the votes are recorded in an array a of length n. To be completely formal we further suppose that the candidates are drawn from the (non-empty) type A. Equality on A is necessarily decidable. Thus from now on we work within the following context.

$$\begin{array}{ll}|[& A \in U_1 \\ ; & .eq. \in \forall(A, a.\forall(A, b.(a = b) \vee \neg(a = b)) \ \% \text{ We write } a.eq.b \\ ; & n \in \mathbb{N} \\ ; & a \in Set(\mathbb{N}, i.i < n) \longrightarrow A\end{array}$$

; $x_0 \in A$
▷

In most ballots there are few candidates and many voters. Thus an obvious solution is to create a pigeon hole for each candidate and to put votes one-by-one into the pigeon holes. When all votes have been counted the number of votes of the candidate receiving most votes can be compared with n **div** 2. This means, however, that we must either assume known the number of candidates, allow for n candidates, or perform a preliminary analysis to determine the exact number of candidates. We obtain a much more elegant solution by doing none of these and abandoning the pigeon-hole solution altogether.

The specification in type theory of the program we require is the following theorem.

(0) $$Set(A, x.majority(x)) \vee \neg Set(A, x.majority(x))$$

where

$$majority(x) \equiv \mathbf{N}(i : 0 \leq i < n : ai = x) > n \textbf{ div } 2.$$

Note that (0) is trivially true in classical mathematics; in constructive mathematics it is only true if one can provide a proof of either the proposition $Set(A, x.majority(x))$ — i.e. exhibit a candidate receiving a majority of votes — or the proposition $\neg Set(A, x.majority(x))$. Note also that an object in the right summand of (0) carries no computational content. What is significant is that the specification is deterministic: any two objects that achieve the specification must be equal.

3.3 Solution strategy

In searching problems such as this a common strategy is to replace a proposition that may or may not be satisfiable by one that is always satisfiable but in such a way that a simple test on a satisfying instance determines whether the original proposition is satisfiable. This, for example, is the strategy adopted when a sentinel is added to the end of an array during a linear search for an element x. It is also the strategy used in specifying binary search when we seek an index to a given array that separates all values at most a given value x from those greater than x, rather than determining whether or not x occurs in the array [Ba1]. And it is the strategy used in the Knuth-Morris-Pratt string searching algorithm where the search for a pattern in a string is replaced by the computation of a failure function [KMP]. In this case we recognise that an easily solved problem is that of determining whether or not a given candidate x occurs a majority of times in a. This problem has specification:

(1) $$\forall(A, x.majority(x) \vee \neg majority(x)).$$

We leave it as an exercise for the reader to construct an object of (1).

Our solution to the majority-vote problem is based on combining a solution to (1) with a solution to the following:

(2) $$Set(A, x.pm(n, x)),$$

where the predicate pm has yet to be defined. (The parameter n occurs in pm in anticipation of later developments.)

Comparing (2) with (0) immediately suggests a definition of pm.

(3) $$pm(n, x) = majority(x) \vee \neg Set(A, x.majority(x)).$$

Of course a pair of objects of types (1) and (2) is not the same as an object of (0). However such an object can be easily recovered. Specifically, the function

$$\lambda q.\mathbf{split}(q, (f, p).\mathbf{when}(fp, a.\mathbf{inl}\ p, b.\mathbf{inr}\ (\lambda x.x))$$

is of type (1) $\wedge$ (2) $\Rightarrow$ (0). This can be seen by the following derivation (which the reader may choose to skip).

% The abbreviation S is used throughout for % $Set(A, x.majority(x)) \vee \neg Set(A, x.majority(x))$

0.0 |[$f \in \forall(A, x.majority(x) \vee \neg majority(x))$

0.1 ; $p \in Set(A, x.pm(n, x))$

▷ % $pm(n, x) \Rightarrow (\neg majority(x) \Rightarrow \neg Set(A, x.majority(x))$, % %exercise 1.13 (b)%

0.2 $p \in Set(A, x.\neg majority(x) \Rightarrow \neg Set(A, x.majority(x)))$

0.3.0 |[$x \in A$

0.3.1 ; $g \in \neg majority(x) \Rightarrow \neg Set(A, x.majority(x))$

▷ % 0.0,0.3.0, $\forall$-elim %

0.3.2 $fx \in majority(x) \vee \neg majority(x)$

0.3.3.0 |[$a \in majority(x)$

▷ % 0.3.0, 0.3.3.0, Subtype-intro, **inl**-intro %

0.3.3.1 **inl** $x \in S$

]|

0.3.4.0 |[$b \in \neg majority(x)$

▷ % 0.3.1,0.3.4.0, $\Rightarrow$-elim %

0.3.4.1 $gb \in \neg Set(A, x.majority(x))$

% 0.3.4.1, example 2.1, **inr**-intro %

0.3.4.2 **inr**$(\lambda x.x) \in S$

]|

% 0.3.2,0.3.3,0.3.4, $\vee$-elim %

0.3.5 **when**$(fx, a.\mathbf{inl}\ x, b.\mathbf{inr}(\lambda x.x)) \in S$

]|

% 0.2,0.3, Subtype-elim %

0.4 **when**$(fp, a.\mathbf{inl}\ p, b.\mathbf{inr}(\lambda x.x)) \in S$

]|

The identifier "*pm*" has been chosen as an abbreviation for "possible-majority candidate". From (3) we observe that an object $x \in Set(A, x.pm(n, x))$ satisfies the property

$$\neg majority(x) \Rightarrow \neg Set(A, x.majority(x)) \tag{4}$$

(since the right side of (3) formally implies (4)). Because a candidate obtaining a majority of votes is always unique, if one exists, (4) may be read as the statement that x excludes all other candidates from being in the majority.

3.4 Invariants versus Inductive Hypotheses

We choose to prove (2) by elimination on n (i.e. by induction over the natural numbers). The basis is trivial since no candidate can occur a majority of times among 0 votes: thus we can straightforwardly exhibit an object of the right summand of (3) and any object will do as our possible-majority candidate. Problems occur when we try to perform the induction step. Suppose that $x \in Set(A, x.pm(k, x))$ for some $k \in \mathbb{N}$. How does one construct an object $y \in Set(A, x.pm(k + 1, x))$? It is clear that more information is needed about the object x — we must strengthen our induction hypothesis.

In programming terms our aim is simply to construct a loop of the form **for** $i := 1$ **to** n **do** that exhibits a possible-majority candidate at each iteration. The notion of inductive hypothesis corresponds directly to the notion of invariant property. Strengthening the inductive hypothesis corresponds to introducing additional auxilliary variables into the computation.

Too strong a hypothesis would be the conjunction of (0) and

$$\forall(A, x.majority(x) \vee \forall(x, \neg majority(x)) \Rightarrow pm(n, x))$$

since it defeats the purpose of introducing the predicate *pm*. (Such a hypothesis states that x is a possible-majority candidate if either it is a majority candidate or no value is a majority candidate. It is a hypothesis likely to be proposed by a mathematician with no regard for the computational efficiency of the proof.) Instead we wish to strengthen the induction hypothesis as little as possible.

Another hypothesis we might consider is that along with the possible majority candidate is known its number of occurrences in the array segment. This is both too strong and too weak. It is too weak to stand alone as an inductive hypothesis. It is too strong because if we do try to prove it inductively we are obliged to consider a hypothesis in which the number of occurrences of every candidate is known, i.e. we have to revert to the pigeon-hole method.

The hypothesis we actually make is that not only is there a possible-majority candidate x but there is also an "estimate" of its number of occurrences in the array segment.

The information that the estimate must convey is when to discard one value and replace it by another. Suppose x is a possible-majority candidate for the

first k votes. Then it is necessary to discard x as a possible-majority value for the first $(k+1)$ votes if the number of occurrences of x among these votes does not guarantee that no other value is a possible-majority value. This will be so when the number of occurrences of x is at most $(k+1)$ **div** 2. This suggests that the estimate we maintain is an upper bound on the number of occurrences of x. Denoting the estimate by e we require:

$$no{-}of{-}occurrences(k,x) \leq e \tag{5.1}$$

(where $no{-}of{-}occurrences(k,x) = \mathbf{N}(i : 0 \leq i < k : ai = x)$).

But this property alone is insufficient. We need to know that e is not a gross overestimate of the number of occurrences of x (the value $e = k$ satisfies (5.1)). The value e must also represent some limit on the number of occurrences of other candidates which precludes their being majority values. We propose therefore that e should also have the property:

$$\forall(A, y.(y = x) \vee no{-}of{-}occurrences(k,y) \leq k - e). \tag{5.2}$$

The property (5.2) will imply the property $pm(k,x)$ if we also add the requirement:

$$k \leq 2 * e. \tag{5.3}$$

In summary, the property we require to prove is

$$Set(A \times \mathbb{N}, (x,e).ind{-}hypo(n,x,e)) \tag{6}$$

where $ind{-}hypo(k,x,e)$ is the conjunction of properties (5.1), (5.2) and (5.3).

We now have two tasks. The first is to prove (6) by induction. An object of (6) will therefore take the form $\mathbb{N}\text{-}elim(n, basis, (m,h)induction{-}step)$. The second task is to verify that the conjunction of properties (5) is indeed a stronger property than $pm(k,x)$. This is needed in order to show that the function **fst** maps an object of type (6) into the required object of type (2).

The second task is a problem of integer arithmetic and is one that we do not tackle here. We assume therefore that (the reader will verify for himself that)

$$\mathbf{fst} \in \forall((6), (x,e).pm(n,x)).$$

Let us now turn to the inductive proof of (6). The basis is a trivial problem of integer arithmetic since we can exhibit an arbitrary candidate, x_0 say, as possible-majority candidate and 0 as the estimate of its number of occurrences.

For the induction step we assume that x and e satisfy properties (5) and show how to construct new values x' and e' satisfying $(5)[k := k+1]$. (To be completely formal we should assume that z, say, is an object of (6) and then split z into its components x and e.) Consider now the effect of the inclusion of the kth vote, ak, in the votes cast.

If $ak = x$ the estimate e of its occurrences increases by one and x is retained as a possible-majority value. That is $(x, e+1) \in (6)[k := k+1]$.

If, however, $ak \neq x$ the estimate e of x's occurrences remains constant and the situation is more complicated. One possibility is that $k = 2 * e$ and hence $k + 1 > 2 * e$. This clearly indicates that x is not a possible-majority value and there is the possiblility that some other value occurs a majority of times. The only value this could be is ak since by the induction hypothesis no value y different from x is a majority value in the first k votes and ak is the only value whose number of occurrences has increased in the process of extending the array segment. An upper bound on the number of occurences of ak is $k - e + 1$ since it occurs, by (5.2), at most $k - e$ times among the first k votes and, trivially, once more among the first $(k + 1)$ votes. Thus if $ak \neq x$ and $k = 2 * e$ the new possible majority value is $x' = ak$ and the new estimate is $e' = k - e + 1$ (which we observe equals $e + 1$).

The final possiblility is that although $ak \neq x$ the bound $k+1 \leq 2*e$ remains true. In this case $k + 1 - e$ is exactly one more than than $k - e$ and the number of occurrences of each element y in the first $(k + 1)$ votes is at most one more than its number of occurrences is the first k votes. Thus $(5)[k := k + 1]$ is true of $x' = x$ and $e' = e$.

Summarising, the inductive step takes the form:

$$
\begin{array}{l}
\textbf{if}\ \ ak = x \longrightarrow \langle x, e+1\rangle \\
[]\ \ (ak \neq x) \wedge (2 * e = k) \longrightarrow \langle ak, e+1\rangle \\
[]\ \ (ak \neq x) \wedge (2 * e > k) \longrightarrow \langle x, e\rangle \\
\textbf{fi}
\end{array}
$$

and the complete program (including a solution to (1)) takes the form:

$$
\begin{array}{l}
\textbf{split}(\\
\quad \mathbb{N}\text{-}elim(n \\
\qquad , < x_0, 0 > \\
\qquad , (k, p).\textbf{split}(p \\
\qquad\qquad , (x, e).\textbf{if}\ \ ak = x \longrightarrow < x, e+1 > \\
\qquad\qquad\qquad\qquad []\ \ (ak \neq x) \wedge (2 * e = k) \longrightarrow < ak, e+1 > \\
\qquad\qquad\qquad\qquad []\ \ (ak \neq x) \wedge (2 * e > k) \longrightarrow < x, e > \\
\qquad\qquad\qquad\qquad \textbf{fi} \\
\qquad\qquad) \\
\qquad) \\
\quad , (x, e). \\
\qquad (\lambda x.\textbf{when}(\\
\qquad\qquad \mathbb{N}\text{-}elim(n, 0, (k, c)\textbf{if}\ ak = x \longrightarrow c + 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad []\ ak \neq x \longrightarrow c \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi} \\
\qquad\qquad\qquad) \\
\qquad\qquad\qquad .gt.n\ \textbf{div}\ 2
\end{array}
$$

```
                    , a.inl x
                    , b.inr b
                    )
            )x
    )
]|
```

4 The future of type structure

The world of programming languages seems to be split into two quite distinct and antagonistic parts — the world of untyped languages and the world of typed languages. The best-known example of the former is probably Lisp but it also includes Prolog, all command languages such as Cshell and text-processing languages like TEX. The best-known example of the latter is probably Pascal but it also includes modern functional languages like SML [Mi2]. An illuminating account of the differences between Lisp and Pascal is afforded by the following quotation from the Foreword written by Alan J. Perlis to Abelson and Sussman's book, "Structure and Interpretation of Computer Programs" [AS].

"It would be difficult to find two languages that are the communicating coin of two more different cultures than those gathered around these two languages. Pascal is for building pyramids — imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms — imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place. ... In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.

The tension that exists between the typed and type-free worlds will never, in my view, be completely reconciled. Those of us who advocate typed languages are, in so doing, also advocating a discipline that ensures that the structure of our "pyramids" is always evident. Discipline means constraint. But there will always be a need for "throw-away" programs, "organisms" that are used, perhaps quite intensively but for a short period of time and then discarded.

Although the tensions will never be reconciled the aim must surely be to bring the two sides closer and closer together. Such certainly is the aim of ML with its introduction of the notion of polymorphic type. Moreover, on the other side, noone would argue against the idea that a clearer structure would facilitate and not hinder the reuse of software.

Alongside the dichotomy between typed and type-free languages most programmers would recognise a dichotomy between "static", or "compile-time", type checking and "dynamic", or "run-time" type-checking. This view of type is however a severe impediment to future progress because there is indeed no

such dichotomy; there is a trichotomy. There is a third time at which type checking can take place and that is at development time.

There are many properties of a program that cannot be discovered either at run-time or at compile-time because of either theoretical or practical impossibility. I need only mention one — termination. Many would argue, however, that static type checking is an *a priori* requirement on any notion of type in programming languages, that such a machine-check substantially increases the reliability of our programs. The truth is though that the most significant benefit of a well-defined type structure is the support that it gives to organising the development of programs and that an experienced programmer will (or should?) never make major type errors in just the same way that he never makes major syntactic errors. The standards that we require of professional programmers should at least ensure that.

For there to be any progress in the exploitation of type structure in improving the quality of computer programs it is vital that it be linked to the development process rather than to issues of implementation. Martin-Löf's theory, with its concept of dependent types has sufficiently enriched the language of types that they may be equated with specifications. There can be no going back to an impoverished, statically-checkable language. The direction has been set for development-time type-checking and we must continue to pursue it.

5 References

[AS] H. Abelson and G.J. Sussman with J. Sussman
Structure and Interpretation of Computer Programs
MIT Press, Cambridge Mass. (1985)

[Ba1] R.C. Backhouse *Program Construction and Verification*
Prentice-Hall International, London (1986)

[Ba2] 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.
Göteborg, Sweden

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

[Bi] 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).

[Chi1] P. Chisholm "Derivation of a parsing algorithm in Martin-Löf's Theory of Types,"
Science of Computer Programming *8* (1987) 1–42 .

[Chi2] P. Chisholm "A theory of finite sets in constructive type theory"
Heriot-Watt Univ., Dept. of Computer Sci., Edinburgh, Scotland. (1987)

[Chu] A. Church *The Calculi of Lambda-Conversion*

Annals of Mathematical Studies **6**,
Princeton University Press, Princeton (1951)

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

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

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

[DF] E.W. Dijkstra and W.H.J. Feijen
Een Methode van Programmeren
Academic Service, Den Haag (1984)

[Dyb] P. Dybjer "Inductively-defined sets in Martin-Löf's set theory"
Programming Methodology Group, Dept. of Computer Sciences,
Chalmers University of Technology, Göteborg, Sweden (April 1987).

[Dyc] R. Dyckhoff "Category theory as an extension of Martin-Löf's Type theory,"
University of St. Andrews, Dept. of Computational Science,
Fife, Scotland. (1985)

[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).

[Gl] V. Glivenko "Sur quelques points de la logique de M. Brouwer"
Academie Royale de Belgique, Bulletins de la classe des sciences,
ser. 5, vol. 15, pp. 183–188 (1929).

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

[Gr] D. Gries *The Science of Programming,*
Springer-Verlag, New York (1981).

[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).

[Kl] S.C.Kleene *Introduction to Metamathematics*
North-Holland Publ. Co. (1952).

[KMP] D.E. Knuth, J.H. Morris and V.R. Pratt
"Fast pattern matching in strings,"
SIAM J. Computing *6*, pp. 325–350. (1977)

[MG] J. Misra and D. Gries
"Finding repeated elements,"
Science of Computer Programming, *2*, 143–152 (1982).

[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, June 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).

[No] B. Nordström "Multilevel Functions in Type Theory,"
Programming Methodolgy Group, Chalmers University of Technology,
S-412296 Göteborg, Sweden.

[NP] B. Nordström and K. Petersson
"Types and Specifications" in *Information Processing 83,* R.E.A. Mason (Ed.)
pp. 915–920, Elsevier Science Publishers B.V. (North Holland)

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

[Pe1] K. Petersson "A programming system for type theory"
Memo 21, Programming Methodology Group, Chalmers Inst. of Technology,
Göteborg, Sweden (1982).

[Pe2] K. Petersson "The subset type former and the type of small types in Martin-Löf's theory of types (Types and Specifications Part II)."
Programming Methodology Group, Chalmers Inst. of Technology,
Göteborg, Sweden.

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

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