Do-it-yourself Type Theory

Roland Backhouse, Paul Chisholm, Grant Malcolm,
Erik Saaman

CS 8811

Do-it-yourself Type Theory

Roland Backhouse Paul Chisholm Grant Malcolm
Erik Saaman

University of Groningen
Department of Mathematics and Computing Science
P.O. Box 800
9700 AV Groningen
The Netherlands

September 19, 1988

Thus you see, ladies, how this story *might* have been written, if the author had but a mind; for, to tell the truth, he is just as familiar with Newgate as with the palaces of our revered aristocracy, and has seen the outside of both. But as I don't understand the language or manners of the Rookery, nor that polyglot conversation, which, according to the fashionable novelists, is spoken by the leaders of *ton;* we must, if you please, preserve our middle course modestly....

W.M. Thackeray, *Vanity Fair.*

1 Introduction

As long ago as 1972 the stage had already been set for much of the current research into mathematical methodologies for the development of verifiable software. In the classic book entitled "Structured Programming" by O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare[19], two topics that were prominently discussed were the use of invariants in program proofs and the use of abstract data types in program design. Since then both the methodologies of program proof and of data structuring have flourished and have become an accepted part of the computing science curriculum in universities.

Both methodologies have flourished — and yet, separately. The "craft" (Reynolds,[57]), "discipline" (Dijkstra,[20]), "logic" (Hehner,[29]) or "science" (Gries,[27]) — call it what you will — of programming is now a well-developed subject which, in the hands of skilled practitioners, is extremely convincing — but typically for small-scale problems. The immense volume of textbooks on data structures, on the other hand, testifies to the enormous improvement in our ability to represent abstract structures within the constraints imposed by the conventional digital computer.

But these texts rarely discuss issues of correctness and their discussion of abstract data structures pales before their discussion of concrete implementations.

Of course, this is all a matter of opinion, an opinion that will undoubtedly be vehemently opposed by many. There has indeed always been a desire to combine the methodology of proof and the methodology of data type abstraction, which has resulted in a considerable amount of ongoing research. The present work is a contribution to this endeavour. Based on the theory of types developed by Martin-Löf[40], we discuss formal methods for introducing and reasoning about abstract types. The examples that we use (binary numerals, finite sets, forests etc.) have no intrinsic interest; our intention is not to provide a fixed set of general-purpose type constructors that are sufficient to encompass all the future needs of all programmers but to provide a discipline whereby programmers may invent their own type constructors peculiar to their own problem domain — "cartesian rings", "integrated mix", "disarrays", or whatever.

A fundamental argument for the use of type information in the design of large programs is that the structure of the program is governed by the structure of the data. A well-established example is the use of recursive descent to structure the parsing (and compilation) of strings defined by a context-free grammar; here the structure of the data is defined by its grammar and the structure of the parsing program is identical. This idea is extended in the denotational description of programming languages[59,62] where a fundamental initial step is the definition of so-called domain equations; those familiar with denotational semantics know that once this step has been taken the later steps are often relatively mundane and straightforward. Users of strongly-typed languages like Pascal will argue strongly that the effective use of type declarations is extremely important for subsequent program development, and even users of untyped languages like Lisp will admit that the programming errors they make are often caused by type violations. A fundamental aspect of Martin-Löf's theory of types is that the connection between type (or data) structure and program structure is made evident. One defines a new type by specifying the way its elements are formed. These are the so-called introduction rules for the type. From these rules there is a systematic method to construct a so-called elimination rule for the type, which says how to reason constructively about (i.e. write programs on) the elements of the type. Computation rules, which in effect define the operational semantics of the programming construct defined by the elimination rule, are also derived from the introduction rules. Our paper describes this process with the aid of a variety of examples.

We have entitled our work "do-it-yourself" type theory because our objective is in part to demystify Martin-Löf's theory. For us a major fascination of the theory is its elegant structure which encourages experimentation with novel data types. This is of considerable practical value because it means that the programmer may work directly within the problem domain rather than within some representation of the problem domain. In order to encourage such experimentation we have chosen to depend on examples rather than to present a theoretical account complete with all kinds of soundness arguments. It does not mean however that the programmer's task has been trivialised and that the professional programmer is no longer required. Quite the opposite. What it does mean is that we may expect rather more from the professional programmer in terms of the clarity and precision of his work. Indeed, in this paper we go into considerably more detail in respect of formal statements of the correctness of our programs than is usually considered necessary. There is thus more work involved in the development process, but the extra effort is in our view fully justified!

In section 2 we discuss the notion of propositions as types, which notion is fundamental to Martin-Löf's theory of types. This section prepares the way for the later sections; it discusses and exemplifies the different judgement forms, in particular the membership judgement form, and describes our style of presenting proof derivations.

Section 3 is central to the remainder of the paper. In it we describe the elegant structure of the rules which characterises Martin-Löf's theory. We show how the elimination and computation rules for a given type constructor are derived from its formation and introduction rules. We do so for free type structures, for congruence types, and for types with information loss. The mechanisms for deriving basic properties of type constructors — closure, individuality and cancellation properties — are also considered.

The remaining sections apply the theory developed earlier. Section 4 exemplifies the way that algorithm development is conducted in a constructive framework. Of particular interest here is the relationship between the formulation of inductive hypotheses and the construction of invariant properties. Section 5 describes and compares two ways of defining and reasoning about binary numerals. Finally, section 6 discusses constructive reasoning on mutually recursive type structures.

2 Propositions As Types

2.1 The Membership Judgement Form

The basis for our work is the theory of types developed by Martin-Löf[39]. In outline, Martin-Löf's theory is a formal system for making so-called "judgements" about certain well-formed formulae. Underlying the construction of the formal system is the principle of "propositions-as-types", which principle is generally attributed to Curry[18] and Howard[31] and pervades a number of foundational studies including the Automath project[7] and categorical logic[37]. In this section we briefly explain the principle, introduce some of the rules in Martin-Löf's theory and discuss the notation and proof format that we use.

"Judgements" in Martin-Löf's theory take one of four possible forms.

$$P \textbf{ type},$$

$$p \in P,$$

$$p = q \in P,$$

$$P = Q.$$

(Here and elsewhere p, q, P and Q stand for expressions.) The fourth judgement form states that P and Q denote equal types. The third judgement form states that "within the type P the objects p and q are equal". We use both judgement forms quite extensively, in particular the third judgement form, but not for the moment. The reader's forbearance is therefore requested until section 3.1.

A judgement of the form P **type** is read as "(the expression) P denotes (or is) a type". For example, we have the judgements

$$\mathbb{N} \textbf{ type}$$

meaning "the set of natural numbers is a type", and

$$List(\mathbb{N})\ \mathbf{type}$$

meaning "the set of lists of natural numbers is a type". The type structure in the theory is very expressive to the extent that whether or not the judgement P **type** can be made about an arbitrary expression, P, is undecidable. Nevertheless, for the purposes of this introductory section, you may regard such a judgement as stating that P is a syntactically well-formed type expression.

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 are

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

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. We call elements of U_1 *small types*.

In "intuitionistic" or "constructive" logic the judgement form $p \in P$ admits a different reading. If P is a proposition (i.e. well-formed formula constructed from the propositional connectives $\wedge, \vee$ etc.) then the judgement form $p \in P$ can be interpreted as the statement 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 so-called principle of propositions-as-types.

The following paragraphs discuss the principle in more detail; table 1 summarises the discussion. In order that the reader may understand the discussion it is necessary to make some preliminary remarks about the notation we use. We assume the reader has a basic familiarity with the lambda calculus[10,62] and the concepts of bound variables, α-, β- and η-reduction. For function application we have chosen to use a period rather than juxtaposition for the simple reason that we wish to use multi-letter identifiers. This decision precluded us from using a period to denote abstraction. Square brackets take its place. An abstraction has the syntax $\lambda([\langle variable\rangle]\langle expression\rangle)$ and denotes the function that given argument a evaluates $\langle expression\rangle$ with the dummy $\langle variable\rangle$ replaced everywhere by a. Note that the scope of the dummy extends to the first unmatched closing parenthesis. We assume that function application associates to the left (thus $f.g.h$ and $(f.g).h$ are the same). 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

Proposition	*Type*	*Type Name*	*Example*
$P \Rightarrow Q$	$P \rightarrow Q$	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	$\lambda([x]\langle x, x\rangle) \in A \Rightarrow (A \wedge A)$ $\lambda([y]\mathit{fst}.y) \in (A \wedge B) \Rightarrow A$
$P \vee Q$	$P + Q$	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	$\langle \mathbb{N}, 0\rangle \in \exists(U_1,\ [A]A)$ $\langle \mathbb{N}, \lambda([x]x)\rangle \in \exists(U_1,\ [A]A \Rightarrow A)$
$\forall(P,\ [x]Q(x))$	$\Pi(P,\ [x]Q(x))$	dependent function space	$\lambda([A]\lambda([x]x)) \in \forall(U_1,\ [A]A \Rightarrow A)$
$\neg P$	$P \rightarrow \emptyset$		$\lambda([f]f.\emptyset) \in \neg\forall(U_1,\ [A]A)$

Table 1: Propositions as types

$P \Rightarrow (Q \Rightarrow R)$. This completes, for the time being, our remarks on notation and we may return to the principle of propositions-as-types.

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 $P \rightarrow Q$ of (total) 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 of 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, $P \wedge Q$ 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 proves A and y proves B, constructs a proof 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. It is an abbreviation for the expression $\lambda([p]\wedge\text{-}\mathit{elim}(p,\ [x,y]x))$. In general, $\wedge\text{-}\mathit{elim}(p,\ [x,y]e)$ 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 $\wedge\text{-}\mathit{elim}(p,\ [x,y]x)$ 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. The construct $\wedge-\mathit{elim}$ is explained in more detail later.)

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

The notation $\forall(P,\ [x]Q(x))$ denotes a universal quantification. We prefer 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 integer array of size n and outputs its maximum element, or a function that inputs the details of a person and then depending on whether they are living or dead, outputs their employment status or details of their estate.

Underlying type theory is a theory of expressions which details how the expressions that represent types and objects may be constructed. Each expression is associated with an arity (a simple form of typing), and a relation of definitional (or intensional) equality between expressions, denoted by $\equiv$, is defined. The theory was developed by Martin-Löf and is discussed in detail by Nordström et al[53]. The relation of definitional equality includes the α-, β- and η-reduction rules of the lambda calculus. In particular, the rule of η-reduction says that the expressions $[x]p(x)$ and p are definitionally equal ($[x]p(x) \equiv p$) provided p contains no free occurrences of x. The symbol Q in $\forall(P,\ [x]Q(x))$ is a schematic variable so does not contain x. Thus, by η-reduction, $\forall(P,\ [x]Q(x)) \equiv \forall(P,\ Q)$. We make use of this fact in abbreviating expressions later.

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

The type $\exists(P,\ Q)$ 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. The idea that algebras are described by the existential or Σ type is due to Nordström and Petersson[51]. The same idea was reported by Mitchell and Plotkin[48].

A consequence of the identification of propositions and types is that the absurdity proposition ($\perp$) is identified with the empty type ($\emptyset$). There can be no proof of the absurdity proposition, so its corresponding type can have no members. Conversely, the empty type contains no members, so its corresponding proposition is unprovable.

Negation is not a primitive concept of type theory. It is defined via the empty type. The negation $\neg P$ is defined to be $P \Rightarrow \emptyset$.

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

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 an object 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 straightforward — we exhibit a counter-example to the proposition that every small type is non-empty, 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 $f.A$, 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.
Functional composition proves the transitivity of implication:

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

The propositional equivalent of currying:

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

Uncurrying:

$$\lambda([f]\lambda([w]\wedge\text{-}\mathit{elim}(w,\ [x,y]f.x.y))) \in (A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)$$

Strengthening the antecedent:

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

Distributivity property of $\Rightarrow$:

$$\begin{aligned}&\lambda([w]\lambda([x]\vee\text{-}\mathit{elim}(w,\ [f]f.(\mathit{fst}.x),\ [g]g.(\mathit{snd}.x))))\\ &\qquad \in [(A \Rightarrow C) \vee (B \Rightarrow C)] \Rightarrow [(A \wedge B) \Rightarrow C]\end{aligned}$$

2.2 An Example Derivation

Martin-Löf's theory is defined by a number of natural deduction style[24] inference rules. For the purposes of illustration we consider just five rules for the moment. These are the assumption rule, (simplified forms of) the rules for function introduction and elimination, and the two rules for ∨ introduction.

$$\frac{A \text{ type}}{\begin{array}{l} |[\ x \in A \\ \rhd\ x \in A \\ ]| \end{array}} \quad \text{assumption}$$

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

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

$$\frac{a \in A}{\mathbf{inl}(a) \in A \vee B} \quad \mathbf{inl}\text{-introduction}$$

$$\frac{b \in B}{\mathbf{inr}(b) \in A \vee B} \quad \mathbf{inr}\text{-introduction}$$

The first of these rules introduces the notion of a *hypothetical judgement*. 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[21], is not used by Martin-Löf but is one introduced by Backhouse[2] in his accounts of the theory.) A rough paraphrase of the assumption rule is the statement that if A is a type then it is possible to introduce a context in which it is assumed that the variable x has type A. The bracket-pair, "|[" and "]|", delimits the scope of the hypothesis $x \in A$. The symbol "▷" separates the hypothesis from the conclusions that may be drawn from it. The basic rule of assumption therefore states that if A is a type then in a context in which x is assumed to have type A it may be concluded that x has type A.

The second rule (λ-introduction) says how functions can be constructed. It has one, hypothetical, premise. 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 $\lambda(f)$ (i.e. $\lambda([x]f(x))$) 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 $\lambda(f)$ 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 $\lambda(f)$. The binding of variables is always associated with the discharge of assumptions.

The third of these rules ($\Rightarrow$-elimination) can also be read in both a logical sense and 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 $f.a$ — 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 $f.a$, the result of applying the function f to a, has type B.

The last two rules say how to construct a proof of a disjunction or, equivalently, 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 **inj**ect left and **inj**ect right, respectively.

We use these rules in the proof of the proposition

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

where A and B are assumed to be small types.

Example 1

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

The derivation is given in figure 1. In this derivation the line numbers (0.0, 0.1 etc.) and the comments enclosed within braces are not part of the derivation but are meant as aids to the reader. The use of the assumption rule is so fundamental that we have chosen not to comment it — it can always be recognised by the appearance of scope brackets. Also, in applying the assumption rule we have chosen not to repeat the hypothesis both before and after the symbol "$\triangleright$".

Step by step one may read through the above derivation as follows. The required conclusion has the form $P \Rightarrow Q$, where the antecedent is $(A \vee (A \Rightarrow B)) \Rightarrow B$ and the consequent is B. We therefore begin in step 0.0 by assuming that f is an element of the antecedent and try to establish the consequent. Looking ahead to steps 0.4 and 1 we see that, having constructed an element of B, λ-introduction can be used to complete the derivation. Now, in steps 0.1.0, 0.1.1 and 0.1.2 we construct an object of $A \Rightarrow B$. First (step 0.1.0) we assume that x is an object of A. From that assumption we may conclude (step 0.1.1) by **inl**-introduction that $\mathbf{inl}(x)$ is an object of $A \vee (A \Rightarrow B)$, and hence, in step 0.1.2, by $\Rightarrow$-elimination, that $f.\mathbf{inl}(x)$ is an object of B. (Note particularly in this step how the context in which judgements are made plays its role.) Step 0.2

```
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, $\Rightarrow$-elimination }
0.1.2               $f.\mathbf{inl}(x) \in B$
                ]|
                    { 0.1.0, 0.1.2, $\lambda$-introduction }
0.2             $\lambda([x]f.\mathbf{inl}(x)) \in A \Rightarrow B$
                    { 0.2, **inr**-introduction }
0.3             $\mathbf{inr}(\lambda([x]f.\mathbf{inl}(x))) \in A \vee (A \Rightarrow B)$
                    { 0.0, 0.3, $\lambda$-elimination }
0.4             $f.\mathbf{inr}(\lambda([x]f.\mathbf{inl}(x))) \in B$
        ]|
            { 0.0, 0.4, $\lambda$-introduction }
1       $\lambda([f]f.\mathbf{inr}(\lambda([x]f.\mathbf{inl}(x)))) \in [(A \vee (A \Rightarrow B)) \Rightarrow B] \Rightarrow B$
```

Figure 1: Derivation for example 1

now follows by λ-introduction — note the discharge of assumption 0.1.0 — and then steps 0.3 and 0.4 follow by **inr**-introduction and $\Rightarrow$-elimination respectively. Finally, as mentioned earlier, we obtain the required conclusion by discharging the initial assumption.

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. As is well-known the law of the excluded middle is not valid in constructive mathematics. More precisely, there is no general method for establishing for an arbitrary proposition whether the proposition or its negation is true; a theory obtained, however, by adding the law of the excluded middle to type theory would not be inconsistent[61]. 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 all expressions of the form $P \Rightarrow \emptyset$ 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 proposition, A, that refutes the law of the excluded middle.

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[35] to Glivenko[25] 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 [17].

3 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. We believe this 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 is suggested by the superficial observation that types are defined by formation, introduction, elimination and computation rules. Once recognised, this structure considerably reduces the burden of understanding. The aim of this section 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[33]. 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[53] have extended the theory to include lists, they and Constable et al[16] have added subset types and Constable et al have introduced quotient types, Nordström has introduced multi-level functions[49], Chisholm has introduced a very special-purpose type of tree structure[8], and Dyckhoff[23] has defined the type of categories.

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

1. A formation rule.
2. The introduction rules.
3. An elimination rule.

4. The computation rules.

5. The 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.

The main contribution that we make here is to describe a scheme for inferring 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 comes from 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. 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[3] and should be evident. (The idea that it may be so inferred was apparently first put forward by Gentzen himself, and later developed by Prawitz[56]. Schröder-Heister's work[60] also supports this view. The role of proof objects, however, is not considered by either Prawitz or Schröder-Heister.)

We have divided the discussion into three parts — free types, congruence types and types with information loss. Free types are those in which canonical objects (i.e., those objects which have been fully evaluated) are equal only if they have the same constructor and they have equal components (thus they are "free" of additional equalities). In Martin-Löf's original theory all types were free types. Congruence types are types in which we choose to postulate additional equalities on the canonical elements. Thus the objects of the type are congruence classes of elements in the corresponding free type. Finally, types with information loss are those in which some information about proof objects is not recorded in the process of constructing the type or its elements. In Martin-Löf's formalisation the only type involving information loss was the equality type. The most widely used example of a type with information loss is the subset type introduced by Constable[12] and Nordström and Petersson[52]. We shall, however, discuss other examples of information loss.

3.1 Free Type Structures

The canonical objects of a type are those formed by the introduction rules. In a "free" type two canonical objects are equal if they have the same constructor and they have equal components. Free types are thus the simplest possible. Free types include cartesian product, disjoint sum, enumerated types, the natural numbers, finite lists, and many more. We begin our account by discussing how finite lists are formalised within the theory. Other examples included are conjunction (cartesian product) and the empty set.

3.1.1 Lists

Formation and Introduction Rules

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

$$\frac{A\ \mathbf{type}}{\mathit{List}(A)\ \mathbf{type}} \qquad \text{List-formation}$$

$$\frac{A\ \mathbf{type}}{\mathbf{nil} \in \mathit{List}(A)} \qquad \mathbf{nil}\text{-introduction}$$

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

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 **nil**- and ::-introduction rules above. We shall follow this practice in the remainder of this discussion.

The formation rule simply says that *List*(A) is a type whenever A is a type. The two introduction rules state, respectively, that **nil** has type *List*(A), for an arbitrary type A, and $a :: l$ has type *List*(A) whenever a has type A and l has type *List*(A) (and, of course, A is a type).

Elimination Rule

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 an arbitrary 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 Θ may be constructed; so, in order to show that property C holds of an arbitrary object of type Θ, it suffices to show that it holds of 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.

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

In this rule the third premise is the one corresponding to **nil**-introduction; it is not hypothetical since apart from the premises of List formation there are no premises in the **nil**-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. More formally, let Θ be a type and θ be a canonical constant of Θ. If the introduction rule for θ has a premise of the form $x \in \Theta$, x is called a *recursive introduction variable*. The effect of this recursive introduction variable is to add an assumption of the form $h \in C(x)$ to that premise of Θ-elimination corresponding to θ-introduction.

The third parameter of *Listelim*, z, is an abstraction. That is, the term z is definitionally equal to $[a, l, h]z(a, l, h)$.

We may read the rule as follows. The first premise is the supposition that C is a well-defined property (or type) over elements of *List*(A). The second premise is the supposition that that x is an arbitrary element of *List*(A). The third and fourth premises determine how to infer that x has property C ("$C(x)$" in the conclusion). Specifically, in the third premise we suppose that y proves $C(\mathbf{nil})$, and in the fourth we suppose that $z(a, l, h)$ proves $C(a :: l)$ assuming a is an element of A, l an element of *List*(A), and z proves $C(l)$. From the justifications of these four premises we conclude that the object *Listelim*$(x,\ y,\ z)$ proves the proposition $C(x)$ (or is an element of the type $C(x)$). The evaluation of *Listelim* expressions is detailed in the computation rules.

For later reference we shall name the three sets of premises in the elimination rule as follows. The first premise is called the ***type*** premise, the second premise is called the ***major*** premise, and the remaining premises are called the ***minor*** premises. There is a minor premise for each introduction rule; the premise corresponding to some canonical constant θ is called the θ-premise. Thus the third premise of the List elimination rule is the **nil**-premise and the fourth premise is the ::-premise. The type abstraction, C, in the type premise is called the ***elimination hypothesis***. (For recursively defined types like lists the more familiar terminology would be ***induction hypothesis***.)

As an example, consider the list append operation. It is defined as:

$$l@m \equiv \mathit{Listelim}(l,\ m,\ [x, y, h]x :: h)$$

```
0.0     |[  l ∈ List(A); m ∈ List(A)
0.1.0   ▷   |[  x ∈ A; y ∈ List(A); h ∈ List(A)
            ▷      { 0.1.0, ::-introduction }
0.1.1          x :: h ∈ List(A)
            ]|
               { 0.0, 0.1, List-elimination }
0.2         Listelim(l, m, [x, y, h]x :: h) ∈ List(A)
               { definition of @ }
0.3         l@m ∈ List(A)
        ]|
```

Figure 2: well-formedness of @

•

To establish the well-formedness of this definition, we must verify the following judgement.

Example 2

$$|[\ l \in \mathit{List}(A);\ m \in \mathit{List}(A)\ \triangleright\ l@m \in \mathit{List}(A)\ ]|$$

The derivation is given in figure 2.

Computation Rules

To express the computation rules we need to make use of the third judgement form in the theory, that is the form

$$p = q \in P.$$

We recall that such a judgement means that p and q are equal elements in the type P.

Computation in the theory is lazy. That is, to evaluate an expression like *Listelim*(...) 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 *Listelim*(**nil**, ...) and of the form *Listelim*($a :: l$, ...). We do so by replacing the major premise "$x \in \mathit{List}(A)$" in the List elimination rule by the premises of the introduction rule. Taking the **nil**-introduction rule we obtain the following computation rule.

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

Since there are no premises in the **nil**-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.

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

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.

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

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

The operational understanding of $\vee$-*elim* is that $\vee\text{-}\mathit{elim}(t,\ e,\ f)$ picks out either e or f depending on the form taken by t. If it has the form $\textbf{inl}(p)$ then $e(p)$ is evaluated. On the other hand if it has the form $\textbf{inr}(q)$ then $f(q)$ is evaluated.

3.1.4 The empty type

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

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

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.

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

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. We encounter $\emptyset$-*elim* frequently in program development. Such occurrences arise within case analyses where one case is always excluded from the computation. It can be proved within the theory that whenever $\emptyset$-*elim* occurs in an expression, its argument can be replaced by an arbitrary expression[2]. We choose to use the constant 0 for the argument in all cases.

Since there are no introduction rules there are no computation rules. Any attempt to evaluate $\emptyset$-$elim(r)$ may thus be considered as a divergent computation.

In conclusion we note that the rule $\Rightarrow$-elimination given in section 2.2 is not the rule we would construct from its corresponding introduction rule, but it is logically equivalent. We have chosen this rule both for historical reasons (it is the type theoretic counterpart of modus ponens), and because it directly corresponds to the notion of function application in typed functional languages. The general form of the rule [42] also requires the notion of "hypothetical hypotheses", which notion we do not discuss here.

3.2 More on Equality and Type Judgements

3.2.1 Families of Types

So far we have said little about type or equality judgements. As we shall see these are not unrelated.

Type judgements are, mostly, very straightforward. An example that we have occasion to use very shortly is the type judgement for the existential type.

$$\frac{\begin{array}{l} A \textbf{ type} \\ |[\ x \in A \\ \rhd\ B(x) \textbf{ type} \\ ]| \end{array}}{\exists(A,\ B) \textbf{ type}} \qquad \exists\text{-formation}$$

The rule states that if A is a type and if $B(x)$ is a type whenever x is an element of A then $\exists(A,\ B)$ is a type.

Note that $B(x)$ may depend on the object x, but as yet we have seen no mechanism within the theory by which such a dependence can be introduced! One such mechanism involves the use of the universes. Specifically, any element of a universe is a type. Moreover if two objects A and B are equal in U_1 then they are equal types. (We give the rule only for the first universe but it is also valid for U_i generally where $1 \leq i$.)

$$\frac{A \in U_1}{A \textbf{ type}} \qquad\qquad \frac{A = B \in U_1}{A = B} \qquad\qquad U_1\text{-elimination}$$

(The name, "U_1-elimination", may be misleading; the rule is not an elimination rule in the same sense as, say, the List elimination rule.)

This seemingly innocuous rule can be combined to great advantage with the elimination rules given earlier to construct so-called "families of types". For example consider a context in which d is declared to have type $A \vee B$ for some types A and B. Then using the $\vee$-elimination rule with elimination hypothesis U_1 we can conclude that $\vee\text{-}elim(d,\ [x]\mathbb{N},\ [y]\{red, yellow, blue\})$ is an element of U_1. Hence, by U_1-elimination it is a type. Indeed, by the computation rules for $\vee$, it is a type with value $\mathbb{N}$ or $\{red, yellow, blue\}$ depending on whether d has value $\mathbf{inl}(a)$, for some a, or $\mathbf{inr}(b)$, for some b. With some imagination one can see how this example can be extended to the construction of quite complex expressions that specify functions that return a result whose type depends on the values of its arguments. (Functions that fail on some arguments offer the most obvious examples; integer division, for example, is typically implemented as a function that returns an integer when its second argument is nonzero but returns an error message otherwise.)

3.2.2 The Equality Type

The second mechanism for introducing objects into type expressions is via the *equality type*. For objects a and b of type A, we define the type $a =_A b$ which is identified with the proposition "a and b are equal objects of type A." It is closely related to the equality judgement, the two different judgement forms

$$a = b \in A$$

and

$$c \in a =_A b$$

```
0.0      |[  $w \in A \vee B$
0.1.0    ▷   |[  $x \in A$
             ▷     { 0.1.0, inl-intro }
0.1.1            $\mathbf{inl}(x) \in A \vee B$
                   { 0.0, 0.1.1, =-formation }
0.1.2            $\mathbf{inl}(x) =_{A \vee B} w$ type
             ]|
               { $A$ type, 0.1.0, 0.1.2, ∃-formation }
0.2          $\exists(A, [x]\mathbf{inl}(x) =_{A \vee B} w)$ type
               { similarly }
0.3          $\exists(B, [y]\mathbf{inr}(y) =_{A \vee B} w)$ type
               { 0.2, 0.3, ∨-formation }
0.4          $\exists(A, [x]\mathbf{inl}(x) =_{A \vee B} w) \vee \exists(B, [y]\mathbf{inr}(y) =_{A \vee B} w)$ type
         ]|
```

Figure 3: Derivation for example 3

meaning essentially the same thing.

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

$$\frac{\begin{array}{l} A\ \mathbf{type} \\ a \in A \\ b \in A \end{array}}{a =_A b\ \mathbf{type}} \qquad \text{=-formation}$$

Note how the =-formation rule relies on the ability to make judgements of the form $a \in A$. This is a significant aspect of the formal system with the implication that it is no longer possible to claim, as we did in section 2.1, that the judgement A **type** means that A is a "well-formed type expression". The two judgement forms are inextricably bound together!

The following example illustrates the use of this rule. It will be used again shortly.

Example 3

```
|[  $w \in A \vee B$
▷   $\exists(A, [x]\mathbf{inl}(x) =_{A \vee B} w) \vee \exists(B, [y]\mathbf{inr}(y) =_{A \vee B} w)$ type
]|
```

The derivation is given in figure 3.

We shall drop the subscript A in $=_A$ when it is clear from the context the type A that is intended.

3.2.3 General Rules

Equality obeys the usual rules of reflexivity, symmetry, transitivity and substitutivity. (In the following rules, **type** premises are omitted.)

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

Reflexivity

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

Symmetry

$$\frac{\begin{array}{l} a = b \in A \\ b = c \in A \end{array}}{a = c \in A} \qquad \frac{\begin{array}{l} A = B \\ B = C \end{array}}{A = C}$$

Transitivity

$$\frac{\begin{array}{l} a = b \in A \\ |[\ x \in A \\ \triangleright\ B(x) = C(x) \\ ]| \end{array}}{B(a) = C(b)} \qquad \frac{\begin{array}{l} a = b \in A \\ |[\ x \in A \\ \triangleright\ c(x) = d(x) \in B(x) \\ ]| \end{array}}{c(a) = d(b) \in B(a)}$$

Substitution

Though not directly concerned with equality, we also require substitution rules for the **type** and $\in$ judgement forms.

$$\frac{\begin{array}{l} a \in A \\ |[\ x \in A \\ \triangleright\ B(x)\ \mathbf{type} \\ ]| \end{array}}{B(a)\ \mathbf{type}} \qquad \frac{\begin{array}{l} a \in A \\ |[\ x \in A \\ \triangleright\ b(x) \in B(x) \\ ]| \end{array}}{b(a) \in B(a)}$$

Substitution

If two types are equal then any element of one is also an element of the other, and equal elements in one are also equal elements in the other.

$$\frac{a \in A \quad A = B}{a \in B} \qquad \frac{a = b \in A \quad A = B}{a = b \in B} \qquad \text{Equality of types}$$

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

$$\frac{a = b \in A}{\mathbf{eq} \in a =_A b} \qquad \text{=-introduction}$$

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

$$\frac{c \in a =_A b}{a = b \in A} \qquad \text{=-elimination}$$

3.2.4 Closure and Individuality Properties

With the rules that we have now assembled we are able to prove two quite remarkable results, first that the only elements of a disjoint sum are those of the form $\mathbf{inl}(a)$ or $\mathbf{inr}(b)$, and that $\mathbf{inl}(a)$ is different from $\mathbf{inr}(b)$, where a and b are elements of A and B respectively.

Example 4

$$\lambda([d]\vee\text{-}elim(d,\ [a]\langle a, \mathbf{eq}\rangle,\ [b]\langle b, \mathbf{eq}\rangle))$$
$$\in \forall(A \vee B,\ [d]\exists(A,\ [x]\mathbf{inl}(x) =_{A\vee B} d) \vee \exists(B,\ [y]\mathbf{inr}(y) =_{A\vee B} d))$$

Example 4 is called the *closure property* for $\vee$. Its derivation is given in figure 4. (Note: readers studying figure 4 are referred to section 3.4.3 for the details of the $\exists$- and $\forall$-introduction rules.)

In other formalisations of type systems one often encounters verbal statements of the form "nothing else is an element of the type", e.g. [30]. The fact that the above proposition can be derived using an elimination rule of a general nature is therefore remarkable. Similar closure properties are straightforward to state and prove for other type constructors. For example, every element of a list is either **nil** or $a :: l$ for some element of the base type, a, and some list, l.

Example 5

$$\begin{array}{l} |[\ a \in A;\ \ b \in B \\ \triangleright\ \mathbf{inl}(a) \neq_{A\vee B} \mathbf{inr}(b) \\ ]| \end{array}$$

0.0 $|[$ $d \in A \vee B$
0.1.0 $\triangleright$ $|[$ $a \in A$
$\triangleright$ $\{$ 0.1.0, **inl**-intro. $\}$
0.1.1 $\mathbf{inl}(a) \in A \vee B$
$\{$ 0.1.1, refl. $\}$
0.1.2 $\mathbf{inl}(a) = \mathbf{inl}(a) \in A \vee B$
$\{$ 0.1.2, $=$-intro. $\}$
0.1.3 $\mathbf{eq} \in \mathbf{inl}(a) =_{A \vee B} \mathbf{inl}(a)$
$\{$ 0.1.0, 0.1.3, $\exists$-intro. $\}$
0.1.4 $\langle a, \mathbf{eq} \rangle \in \exists(A, [x]\mathbf{inl}(x) =_{A \vee B} \mathbf{inl}(a))$
$\{$ 0.1.4, **inl**-intro. $\}$
0.1.5 $\mathbf{inl}(\langle a, \mathbf{eq} \rangle)) \in \exists(A, [x]\mathbf{inl}(x) =_{A \vee B} \mathbf{inl}(a)) \vee \exists(B, [y]\mathbf{inr}(y) =_{A \vee B} \mathbf{inl}(a))$
$]|$
0.2.0 $|[$ $b \in B$
$\triangleright$ $\{$ similarly $\}$
0.2.1 $\mathbf{inr}(\langle b, \mathbf{eq} \rangle)) \in \exists(A, [x]\mathbf{inl}(x) =_{A \vee B} \mathbf{inr}(b)) \vee \exists(B, [y]\mathbf{inr}(y) =_{A \vee B} \mathbf{inr}(b))$
$]|$
$\{$ example 3, 0.0, 0.1, 0.2, $\vee$-elim. $\}$
0.3 $\vee\text{-}elim(d, [a]\langle a, \mathbf{eq} \rangle, [b]\langle b, \mathbf{eq} \rangle) \in \exists(A, [x]\mathbf{inl}(x) =_{A \vee B} d) \vee \exists(B, [y]\mathbf{inr}(y) =_{A \vee B} d)$
$]|$

Figure 4: Derivation for example 4

where $a \neq_A b$ abbreviates $\neg(a =_A b)$. Example 5 is called the *individuality property* for $\vee$. Its derivation is given in figure 5.

Similar individuality properties for other types are straightforward. For example, the individuality property for *List* states that **nil** is different from $a :: l$ for all a and l.

Other fundamental properties of the type system that can be formally derived include cancellation properties (e.g. if two pairs are equal then their components are equal).

3.3 Congruence Types

Recall that objects of free types are equal if and only if they are built from the same constructor, and the arguments of the constructor are equal. Thus, the expressions $0 :: \mathbf{nil}$ and $0 :: 0 :: \mathbf{nil}$ are distinct objects of the type *List*($\mathbb{N}$). They are built from the same constructor ($::$) and have the same first argument (0), but their second arguments are different (**nil** and $0 :: \mathbf{nil}$ are built from different constructors). However, not all types enjoy this property. Consider a type of finite sets whose canonical objects are of the form ϕ (construct the empty set) and $a \bullet s$ (add the element a to the set s). The expressions $0 \bullet \phi$ and $0 \bullet 0 \bullet \phi$ denote equal sets since they have the same membership properties. However, the free type whose constructors are ϕ and $\bullet$ would distinguish the objects $0 \bullet \phi$ and $0 \bullet 0 \bullet \phi$. Congruence types allow us to impose equalities on the canonical objects of a type over and above those implied by a free type. The equalities are specified by extra introduction rules, which we refer to as *congruence rules*. We describe congruence types in this section by defining finite bags (multisets) and finite sets. Bags are constructed from lists by

```
0       $\emptyset \in U_1$
1       $\mathbb{N} \in U_1$
2.0     |[   $d \in A \vee B$
        ▷        { 0,1, $\vee$-elim. }
2.1          $\vee\text{-}elim(d, [x]\emptyset, [y]\mathbb{N}) \in U_1$
                 { 2.1, refl. }
2.2          $\vee\text{-}elim(d, [x]\emptyset, [y]\mathbb{N}) = \vee\text{-}elim(d, [x]\emptyset, [y]\mathbb{N}) \in U_1$
        ]|
3.0     |[   $a \in A;\ \ b \in B$
        ▷        { 3.0, inl-intro, 2.1, inl-comp }
3.1          $\vee\text{-}elim(\mathbf{inl}(a), [x]\emptyset, [y]\mathbb{N}) = \emptyset \in U_1$
                 { similarly }
3.2          $\vee\text{-}elim(\mathbf{inr}(b), [x]\emptyset, [y]\mathbb{N}) = \mathbb{N} \in U_1$
3.3.0        |[   $r \in \mathbf{inl}(a) =_{A \vee B} \mathbf{inr}(b)$
             ▷        { 3.3.0, =-elimination }
3.3.1             $\mathbf{inl}(a) = \mathbf{inr}(b) \in A \vee B$
                      { 2.0, 2.2, 3.3.1, substitution }
3.3.2             $\vee\text{-}elim(\mathbf{inl}(a), [x]\emptyset, [y]\mathbb{N}) = \vee\text{-}elim(\mathbf{inr}(b), [x]\emptyset, [y]\mathbb{N}) \in U_1$
                      { 3.1, 3.2, 3.3.2, trans., sym. }
3.3.3             $\emptyset = \mathbb{N} \in U_1$
                      { 3.3.3, $U_1$-elim. }
3.3.4             $\emptyset = \mathbb{N}$
                      { 3.3.4, $0 \in \mathbb{N}$, type equality }
3.3.5             $0 \in \emptyset$
             ]|
                 { 3.3.0, 3.3.5, $\Rightarrow$-introduction }
3.4          $\lambda([r]0) \in \mathbf{inl}(a) \neq_{A \vee B} \mathbf{inr}(b)$
        ]|
```

Figure 5: Derivation for example 5

adding a congruence rule which identifies lists which differ only in the order of elements. Sets are constructed from bags by identifying those bags which differ only in the number of occurrences of elements.

3.3.1 Finite Bags

Suppose we wish to define a type constructor $\Im$ such that $\Im(A)$ is the type of finite bags of A. Any such bag can be constructed by listing its elements. Conversely any list of elements of A may be regarded as a finite bag of A provided that we disregard the order of the elements. $\Im(A)$ 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.

We define the type constructor $\Im$ by adding to the introduction rules for *List* a congruence rule defining the above equivalence. In full the rules are:

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

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

$$\frac{\begin{array}{l} a \in A \\ s \in \Im(A) \end{array}}{a \bullet s \in \Im(A)} \qquad \bullet\text{-introduction}$$

$$\frac{\begin{array}{l} a \in A \\ b \in A \\ s \in \Im(A) \end{array}}{a \bullet b \bullet s = b \bullet a \bullet s \in \Im(A)} \qquad \text{order}$$

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 rule but with an additional premise corresponding to the order rule. As with free types, each introduction rule yields a premise in the elimination rule.

$|[\ w \in \Im(A) \ \triangleright\ C(w)\ \textbf{type}\ ]|$
$t \in \Im(A)$
$c \in C(\phi)$
$|[\ a \in A;\ s \in \Im(A);\ h \in C(s)$
$\triangleright\ d(a,s,h) \in C(a \bullet s)$
$]|$
$|[\ a \in A;\ b \in A;\ s \in \Im(A);\ h \in C(s)$
$\triangleright\ d(a, b \bullet s, d(b,s,h)) = d(b, a \bullet s, d(a,s,h)) \in C(a \bullet b \bullet s)$
$]|$

$$\overline{\hspace{20em}} \quad \Im\text{-elimination}$$

$\Im\text{-}elim(t, c, d) \in C(t)$

•

The premise corresponding to the order rule

$|[\ a \in A;\ b \in A;\ s \in \Im(A);\ h \in C(s)$
$\triangleright\ d(a, b \bullet s, d(b,s,h)) = d(b, a \bullet s, d(a,s,h)) \in C(a \bullet b \bullet s)$
$]|$

is constructed as follows. The assumptions are derived from the premises of the order rule as in the discussion of lists. From the assumptions $b \in A$, $s \in \Im(A)$ and $h \in C(s)$, the •-premise establishes

$$d(b,s,h) \in C(b \bullet s) \tag{1}$$

From $a \in A$, $b \bullet s \in \Im(A)$ and (1), the •-premise also establishes

$$d(a, b \bullet s, d(b,s,h)) \in C(a \bullet b \bullet s) \tag{2}$$

By similar reasoning we get

$$d(b, a \bullet s, d(a,s,h)) \in C(b \bullet a \bullet s)$$

The order rule states that the expressions $a \bullet b \bullet s$ and $b \bullet a \bullet s$ are equal in the type $\Im(A)$, so the types $C(a \bullet b \bullet s)$ and $C(b \bullet a \bullet s)$ must also be equal. Thus, by type equality we have

$$d(b, a \bullet s, d(a,s,h)) \in C(a \bullet b \bullet s) \tag{3}$$

Viewing the elimination rule as a method for constructing functions on the type $\Im(A)$, (2) is the expression to be evaluated when the function is applied to $a \bullet b \bullet s$, and (3) is the expression to be evaluated when the function is applied to $b \bullet a \bullet s$. Equal arguments must produce equal results. The order premise formalises the requirement that the objects of (2) and (3) are equal.

The computation rules are constructed similarly. ϕ-computation has all the premises of $\Im$-elimination except $t \in \Im(A)$ and has conclusion

$$\Im\!-\!elim(\phi, c, d) = c \in C(\phi)$$

•-computation has the extra premise

$$a \in A$$

and conclusion

$$\Im\text{−}elim(a \bullet t, c, d) = d(a, t, \Im\text{−}elim(t, c, d)) \in C(a \bullet t)$$

As an example, consider the union operation over bags. It is similar to the corresponding operation on sets but repeated occurrences of elements must be retained. That is, for any a in A, if there are m occurrences of a in the bag s and n occurrences of a in t, then there are $m + n$ occurrences of a in the union of s and t. The operation is defined as

$$s \cup t \equiv \Im\text{−}elim(s, t, [x, y, h]x \bullet h)$$

In clausal form, this definition would be written as

$$\begin{array}{lcl} \emptyset \cup t & = & t \\ (a \bullet s) \cup t & = & a \bullet (s \cup t) \end{array}$$

To verify the well-definedness of $\cup$, we must establish the judgement

$$|[\ s \in \Im(A);\ t \in \Im(A)\ \triangleright\ s \cup t \in \Im(A)\ ]|$$

The instances of the minor premises of the elimination rule are

$$\begin{array}{ll} t \in \Im(A) & \phi\text{−premise} \\ |[\ x \in A;\ y \in \Im(A);\ h \in \Im(A)\ \triangleright\ x \bullet h \in \Im(A)\ ]| & \bullet\text{−premise} \\ |[\ a \in A;\ b \in A;\ y \in \Im(A);\ h \in \Im(A) & \text{order − premise} \\ \triangleright\ a \bullet b \bullet h = b \bullet a \bullet h \in \Im(A) & \\ ]| & \end{array}$$

The ϕ-premise is established by assumption, the $\bullet$-premise by $\bullet$-introduction, and the order-premise by the order introduction rule. For brevity, we shall not formally verify the correctness of this definition of $\cup$. Instead, we establish an important property of $\cup$: commutativity. That is, we will verify the judgement

$$|[\ s \in \Im(A);\ t \in \Im(A)\ \triangleright\ s \cup t = t \cup s \in \Im(A)\ ]|$$

which clearly should hold since the order of elements in a bag is irrelevant. The derivation is detailed in figure 6. We assume $\bullet$ has greater binding power than $\cup$.

As a second example, we define the cardinality operation over bags which counts the number of elements in a bag including repeated occurrences of the same element. It is defined as

$$|b| \equiv \Im\text{−}elim(b, 0, [x, y, h]\mathbf{succ}(h))$$

The instances of the minor premises of the elimination rule when proving the well-formedness of $|\_|$ are

$$\begin{array}{ll} 0 \in \mathbb{N} & \phi\text{−premise} \\ |[\ x \in A;\ y \in \Im(A);\ h \in \mathbb{N}\ \triangleright\ \mathbf{succ}(h) \in \mathbb{N}\ ]| & \bullet\text{−premise} \\ |[\ a \in A;\ b \in A;\ y \in \Im(A);\ h \in \mathbb{N} & \text{order − premise} \\ \triangleright\ \mathbf{succ}(\mathbf{succ}(h)) = \mathbf{succ}(\mathbf{succ}(h)) \in \mathbb{N} & \\ ]| & \end{array}$$

```
0.0     |[  $s, t \in \Im(A)$
        ▷      { minor premises of $\cup$ well-formedness, $\phi$-computation }
0.1         $\phi \cup t = t \in \Im(A)$
               { $\Im$-elimination with $t \in \Im(A)$ as major premise }
0.2         $t \cup \phi = t \in \Im(A)$
               { 0.2, symmetry, 0.1, transitivity }
0.3         $\phi \cup t = t \cup \phi \in \Im(A)$
               { 0.3, =-introduction }
0.4         $\mathrm{eq} \in \phi \cup t =_{\Im(A)} t \cup \phi$
0.5.0       |[   $a \in A;\ y \in \Im(A);\ h \in y \cup t =_{\Im(A)} t \cup y$
0.5.1       ▷        $a \bullet y \cup t$
                 $=_{\Im(A)}$          { definition of $\cup$ }
                     $\Im\!-elim(a \bullet y, t, [x, y, h]x \bullet h)$
                 $=_{\Im(A)}$          { 0.5.0, minor premises of $\cup$ well-formedness, $\bullet$-computation }
                     $a \bullet \Im\!-elim(y, t, [x, y, h]x \bullet h)$
                 $=_{\Im(A)}$          { definition of $\cup$ }
                     $a \bullet (y \cup t)$
                 $=_{\Im(A)}$          { 0.5.0, =-elimination, $\bullet$-introduction, subst }
                     $a \bullet (t \cup y)$
                 $=_{\Im(A)}$          { $\Im$-elimination with $t \in \Im(A)$ as major premise }
                     $t \cup a \bullet y$
                 { 0.5.1, =-introduction }
0.5.2            $\mathrm{eq} \in a \bullet y \cup t =_{\Im(A)} t \cup a \bullet y$
            ]|
0.6.0       |[   $a, b \in A;\ y \in \Im(A);\ h \in y \cup t =_{\Im(A)} t \cup y$
            ▷      { 0.6.0, $\bullet$-introduction, 0.5, subst }
0.6.1            $\mathrm{eq} = \mathrm{eq} \in a \bullet b \bullet y \cup t =_{\Im(A)} t \cup a \bullet b \bullet y$
            ]|
               { 0.0, 0.4, 0.5, 0.6, $\Im$-elimination }
0.7         $\Im\!-elim(s, \mathrm{eq}, [x, y, h]\mathrm{eq}) \in s \cup t =_{\Im(A)} t \cup s$
               { 0.7, =-elimination }
0.8         $s \cup t = t \cup s \in \Im(A)$
        ]|
```

Figure 6: Commutativity of $\cup$

They are easily established using the introduction rules for $\mathbb{N}$ and reflexivity.

One possible approach to specifying the correctness of the cardinality operation is to consider bijections between the elements of a bag and some subset of the natural numbers. Let $\{s\}$ denote the type whose objects are exactly the members of the bag s (where different occurrences of the same element in s are distinguished in $\{s\}$), and $\overline{n}$ denote the type of natural numbers less than n. The specification of cardinality states that for any bag s, there exists a bijection between the types $\{s\}$ and $\overline{|s|}$. A bijection between types A and B is defined to be

$$Bijection(A, B) \equiv \exists(A \Rightarrow B, [f] Injective(A, B, f) \wedge Surjective(A, B, f))$$

that is, an injective (1-1) and surjective (onto) function from A to B.

$$\begin{aligned} Injective(A, B, f) &\equiv \forall(A, [a]\forall(A, [b] f.a =_B f.b \Rightarrow a =_A b)) \\ Surjective(A, B, f) &\equiv \forall(B, [b]\exists(A, [a] f.a =_B b)) \end{aligned}$$

The correctness of $|\_|$ is established by verifying the judgement

$$|[\ s \in \Im(A) \ \triangleright \ p \in Bijection(\{s\}, \overline{|s|})\ ]|$$

for some p.

3.3.2 Finite Sets

Sets, like bags, are independent of the order in which elements appear in their construction. In addition, sets are independent of the number of occurrences of any element appearing in their construction. Therefore, we can define a type of finite sets by adding to the type $\Im$ a congruence rule which identifies bags which differ only in the number of occurrences of elements. The extra rule is

$$\frac{a \in A}{a \bullet a \bullet \emptyset = a \bullet \emptyset \in \Im(A)} \qquad \text{repetition}$$

which yields the following extra premise in the elimination rule.

$$|[\ a \in A \ \triangleright \ d(a, a \bullet \emptyset, d(a, \emptyset, c)) = d(a, \emptyset, c) \in C(a \bullet \emptyset)\ ]|$$

The repetition premise is derived from the repetition congruence rule as follows. From $a \in A$ (assumption), $\phi \in \Im(A)$ (ϕ-introduction), and $c \in C(\phi)$ (ϕ-premise), the $\bullet$-premise establishes

$$d(a, \phi, c) \in C(a \bullet \phi) \tag{4}$$

From $a \in A$, $a \bullet \phi \in \Im(A)$ ($\bullet$-introduction), and 4, the $\bullet$-premise also establishes

$$d(a, a \bullet \phi, d(a, \phi, c)) \in C(a \bullet a \bullet \phi)$$

The repetition rule states that the expressions $a \bullet \phi$ and $a \bullet a \bullet \phi$ are equal in the type $\Im(A)$, so the types $C(a \bullet \phi)$ and $C(a \bullet a \bullet \phi)$ must also be equal by substitution properties. By type equality we have

$$d(a, a \bullet \phi, d(a, \phi, c)) \in C(a \bullet \phi) \qquad (5)$$

Viewing the elimination rule as a method for constructing functions on the type $\Im(A)$, (4) is the expression to be evaluated when the function is applied to $a \bullet \phi$, and (5) is the expression to be evaluated when the function is applied to $a \bullet a \bullet \phi$. Equal arguments must produce equal results. The repetition premise formalises the requirement that the objects of (4) and (5) be equal.

The computation rules for finite sets are the same as those for finite bags but with the addition of the repetition premise.

The repetition premise for sets means that there are more conditions to be satisfied when constructing functions over sets than there are when constructing functions over bags. Any function we define over sets is also a function over bags, but there are functions over bags which are not functions over sets. Consider the two operations defined on bags earlier. The union operation remains valid with respect to sets. To verify that $\cup$ is well-formed in the type of finite sets, we have to verify the premises from the proof for bags plus the additional repetition premise. The particular instance is

$$|[\ a \in A \ \triangleright\ a \bullet a \bullet t = a \bullet t \in \Im(A)\ ]|$$

which is verified by $\Im$ elimination with the repetition rule as basis, and substitution properties and the order rule being used for the inductive step.

The derivation of commutativity of $\cup$ is very similar. We merely have the extra premise

$$|[\ a \in A \ \triangleright\ \mathbf{eq} = \mathbf{eq} \in a \bullet \emptyset \cup t =_{\Im(A)} t \cup a \bullet \emptyset\ ]|$$

which is verified by substitution and reflexivity on the $\bullet$-premise.

In contrast to $\cup$, the cardinality operation defined above is not valid with respect to sets. The number of occurrences of an element in a bag is significant, so the cardinality operation counts each occurrence of each element. The repetition premise states that the number of occurrences of an element in a set is insignificant. Formally, the problem arises in the repetition premise when proving the well-formedness of $|\_|$ with the definition given above. The instance of the repetition premise is

$$|[\ a \in A \ \triangleright\ \mathbf{succ}(\mathbf{succ}(0)) = \mathbf{succ}(0) \in \mathbb{N}\ ]|$$

This judgement is not provable. In fact, the individuality property of $\mathbb{N}$ shows the judgement to be inconsistent. An alternative definition of cardinality must be given for sets. It must count each distinct element in a set only once, regardless of how many times an element appears in the construction of a set. This is achieved as follows.

$$|s| \equiv \Im\text{–}elim(s, 0, [x, y, h]\mathbf{if}\ x \epsilon y\ \mathbf{then}\ h\ \mathbf{else}\ \mathbf{succ}(h))$$

where

$$a \epsilon s \equiv \Im\text{–}elim(s, false, [x, y, h]\mathbf{if}\ a.eq.x\ \mathbf{then}\ true\ \mathbf{else}\ h)$$

The symbol ϵ denotes set membership. It is only definable if we have a decidable equality relation over the objects of type A, which has been denoted by the infix operator $.eq.$ in the above expression.

Thus, we are only able to define the cardinality operation for those types $\Im(A)$ such that A is decidable. Other than the requirement of decidability, the specification of cardinality for sets is the same as for bags.

Starting with lists, we added a congruence rule to give bags. A further congruence rule gave sets. Meertens[43] takes this process one step further. He begins with binary trees, whose canonical forms construct the empty tree, a tip, and compose two trees. Lists are obtained by adding congruence rules stating that tree composition is associative, and the empty tree is both a left and a right identity of tree composition. Trees constructed by empty, tip, and composition are then viewed as the nil list, unit list, and list append respectively. Bags and sets follow by adding rules concerning the commutativity and idempotence of composition respectively.

3.3.3 The NuPrl Quotient Type

The motivation behind congruence types is to allow stronger equality relations between objects of a type than those implied by a free type. An important aspect of our method is that the congruence types we construct can be viewed as primitive types of the theory; they have the same status as the types $\forall$, $\mathbb{N}$, etc. Thus, the definition of a congruence type of finite bags allows us to reason directly in the theory of bags. An alternative approach is taken by the NuPrl group[16]. They have introduced a quotient operation which allows one to construct a new type by defining a stronger equality relation over an existing type. Given A **type** and $|[x, y \in A \rhd E(x, y)$ **type**$]|$, the type

$$A//E$$

is the quotient of A by the (equivalence) relation E. The objects of $A//E$ are the objects of A, but equality on $A//E$ need not be the same as equality on A. Two objects a and b are equal in $A//E$ exactly when we can prove the judgement $p \in E(a, b)$ for some p.

Whereas we view congruence types as primitive, the quotient type forces one to view types as defined. Reasoning about a quotient type involves reasoning about the primitive types which were used to define it, with the disadvantage that we can no longer work directly in the theory of interest. Cleaveland and Panangaden[11] and Chisholm[9] give different formulations of finite sets using the quotient type, the former by quotienting finite maps and the latter by quotienting finite lists. In both cases, the size and complexity of proofs in the defined type is substantially greater than the primitive type of sets we give above.

The Nuprl quotient type is an instance of a congruence type. Its elimination rule can be derived from the introduction rule using the method described above. Note, however, that the quotient type also exhibits information loss, which is discussed in the following section.

3.4 Computational Redundancy and Types with Information Loss

A feature of the types $=_A$ (section 3.2.2) and $\emptyset$ (section 3.1.4) encountered earlier is that one is interested only in whether they are inhabited; their objects do not contribute any computational information and are in that sense redundant. Closely related to computational redundancy is the notion of information loss. The conclusions of the inference rules of the free and congruence types we have encountered so far retain all the information embodied in their premises. In particular, when a judgement of the form $a \in A$ appears as a premise of an introduction rule, the object a

also appears in the conclusion. From a computational point of view, the conclusion of each rule inherits the computational content of its premises. In this section, we describe how types exhibiting information loss can be introduced into the theory. First, the types $=_A$ and $\emptyset$ are discussed in more detail as their computational redundancy is vital for the effective use of information loss.

3.4.1 Computational Redundancy

Given any proof of an equality type, such as

$$p \in a =_A b$$

where p may be arbitrarily complex, the rule =-elimination establishes

$$a = b \in A$$

and =-introduction gives

$$\mathbf{eq} \in a =_A b$$

That is, any object derived from a proof of an equality type can be transformed in two steps to the constant **eq**. Thus, the object synthesised from a derivation of an equality type is uninteresting. Only the existence of the object is important since it witnesses the truth of the equality specified by the type. One can view the type $=_A$ as a special case of information loss since the object r in the premise of =-elimination does not appear in the conclusion.

The type $\emptyset$ is computationally uninteresting for the simple reason that it contains no objects. Unlike $=_A$, it is not a special case of information loss. The conclusions of its rules retain all the information in the premises.

We shall say that a type A *exhibits computational redundancy* if for each a in A there exists an a' in A such that a' is a closed expression and $a = a' \in A$. Thus we can always get rid of free variables from objects of types exhibiting computational redundancy. The types $=_A$ and $\emptyset$ exhibit computational redundancy. Using them as a basis, we can construct other types exhibiting computational redundancy. For example, all objects of the type $A \Rightarrow \emptyset$ (i.e. $\neg A$), where A is an arbitrary type, can be simplified to $\lambda([x]x)$. Objects of negation types thus have no computational content. The important point to note about such types, and types exhibiting computational redundancy in general, is that their objects can always be transformed to equal objects containing no free variables.

3.4.2 Information Loss: The Subset Type

Types with information loss allow unwanted proof objects to be discarded, for example from types exhibiting computational redundancy. In general, however, we may construct objects which have computational content but which are not interesting for the problem at hand. Recall that to verify the correctness of the cardinality operation over finite bags, we must establish the judgement

$$|[\ s \in \Im(A) \ \triangleright\ p \in \mathit{Bijection}(\{s\}, \overline{|s|})\ ]|$$

for some p. The first component of the object p derived from the proof is a bijection between $\{s\}$ and $\overline{|s|}$, that is, a computable function from $\{s\}$ to $\overline{|s|}$. In this situation, p's computational content is irrelevant. We merely wish to construct and verify the cardinality operation.

Uninteresting proof objects can be discarded using the subset type[52,16]. It allows the construction of a set of objects of some type with a common property. The formation rule is:

$$\frac{\begin{array}{l} A \ \textbf{type} \\ |[\ x \in A \ \rhd \ B(x) \ \textbf{type}\ ]| \end{array}}{\mathit{Set}(A,B) \ \textbf{type}} \qquad \mathit{Set}\text{-formation}$$

The subset type is so called because its objects are a subset of the objects of type A — more specifically, those objects a of type A such that the type $B(a)$ is inhabited (contains at least one object). The introduction rule is:

$$\frac{\begin{array}{l} a \in A \\ b \in B(a) \end{array}}{a \in \mathit{Set}(A,B)} \qquad \mathit{Set}\text{-introduction}$$

The object b in the second premise does not appear in the conclusion, its information is lost. Unlike the other types introduced so far, *Set* has no canonical objects of its own. Its elements are merely a subset of the elements of A. The elimination rule is:

$$\frac{\begin{array}{l} |[\ w \in \mathit{Set}(A,B) \ \rhd \ C(w) \ \textbf{type}\ ]| \\ a \in \mathit{Set}(A,B) \\ |[\ x \in A;\ y \in B(x) \\ \rhd \ c(x) \in C(x) \\ ]| \end{array}}{c(a) \in C(a)} \qquad \mathit{Set}\text{-elimination}$$

The first two premises are the standard type and major premises appearing in all elimination rules. There is one introduction rule, hence one minor premise. The assumptions of the minor premise are derived from the premises of *Set*-introduction as usual. From these two assumptions, the judgement $x \in \mathit{Set}(A,B)$ follows by *Set*-introduction so $C(x)$ is a well-formed type. The information loss in the introduction rule is reflected in the minor premise by the constraint that the variable y may not appear free in the consequent $c(x) \in C(x)$, just as the object b does not appear in the conclusion of *Set*-introduction.

Since *Set* has no canonical constants, it is unnecessary to have an elimination constant. Likewise, there are no computation rules.

The importance of computational redundancy for information loss is seen in the constraint on the minor premise of *Set*-elimination. The variable y may not appear free in the consequent.

There are two commonly occurring cases in which dependence upon the variable y may be obviated: $B(x)$ exhibits computational redundancy; and $C(x)$ is itself a type involving information loss. The first case is exemplified by the equality type. Let α be some fixed object of type A, and take $B(x) \equiv (x =_A \alpha)$. Then, as explained above, from the assumption $y \in (x =_A \alpha)$ we obtain, in two steps, $\mathbf{eq} \in (x =_A \alpha)$ and so the variable y is replaced by the closed term $\mathbf{eq}$. The second case, where $C(x)$ involves information loss, can be exemplified by the subset type. Take $B(x) \equiv P(x) \wedge Q(x)$ and $C(x) \equiv Set(A, P)$: from the assumption $y \in P(x) \wedge Q(x)$ we obtain $fst.y \in P(x)$ and then, by subset introduction, $x \in Set(A, P)$. Thus, although in both cases a proof of $C(x)$ may depend upon the *truth* of $B(x)$, it is possible to eliminate any dependence upon the variable y.

Set allows us to verify the correctness of the cardinality operation without unwanted proof objects appearing in the result. We would now verify:

$$|[\ s \in \Im(A)\ \ \triangleright\ \ |s| \in Set(\mathbb{N}, [n]Bijection(\{s\}, \overline{n}))\ ]|$$

Although an object of type $Bijection(\{s\}, \overline{|s|})$ would be constructed during the derivation, it is discarded when the rule *Set*-introduction is applied.

Comparison of the *Set*- and $\exists$-introduction rules is educational, not only because it allows one to see explicitly the information that is lost but because it also suggests other forms of information loss.

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

Note that objects of an existential type are ordered pairs; objects of a *Set* type can be considered as objects of the corresponding existential type but where the second component has been discarded.

Instead of discarding the second component we might choose to discard the first component. This would give objects of a union type.

$$\frac{\begin{array}{l} a \in A \\ b \in B(a) \end{array}}{b \in \cup(A, B)}\qquad \cup\text{-introduction}$$

An object of $\cup(A, B)$ is an object of some member $B(a)$ of a family of types $B(x)$, indexed by x in A, but where the information about which particular member has been lost. We shall not pursue this type any further since it does not yet appear to have found practical application. A useful exercise for the reader, however, is to construct the elimination rule for the type by analogy with the *Set*-elimination rule.

3.4.3 Information Loss: The Polymorphic Function Type

Dual to the information loss that occurs in going from an existential type to a union type is the loss of dependency in going from a universal type to an implication. We see the latter by comparing their type formation rules.

$$\frac{\begin{array}{l} A \textbf{ type} \\ |[\ x \in A \\ \rhd\ B \textbf{ type} \\ ]| \end{array}}{A \Rightarrow B \textbf{ type}} \ \Rightarrow\text{-formation} \qquad \frac{\begin{array}{l} A \textbf{ type} \\ |[\ x \in A \\ \rhd\ B(x) \textbf{ type} \\ ]| \end{array}}{\forall(A,\ B) \textbf{ type}} \ \forall\text{-formation}$$

In general a $\forall$-type is much more specific than an $\Rightarrow$-type: if $f \in \forall(A,\ B)$ then $f \in A \Rightarrow \cup(A, B)$, but the converse is not always the case since the specific information about the range of the function f is lost through the $\cap$-type. For example, $\forall(\mathbb{N},\ [n]Set(\mathbb{N}, [m]m = n \textbf{ div } 2))$ is satisfied uniquely by division by 2, but $\mathbb{N} \Rightarrow \cup(\mathbb{N}, [n]Set(\mathbb{N}, [m]m = n \textbf{ div } 2))$ is satisfied by all total functions from $\mathbb{N}$ to $\mathbb{N}$, including division by 2 but also including, say, the identity function. (Observe that $\mathbb{N}$ has precisely the same elements as $\cup(\mathbb{N}, [n]Set(\mathbb{N}, [m]m = n \textbf{ div } 2))$.)

The examples above (*Set*, $\cup$ and $\Rightarrow$) suggest that we can play a syntactic game with the type constructors we have seen so far whereby we choose to discard individual items of information. Two forms of polymorphism arise naturally in this way, one of them subsuming the notion of type polymorphism, the importance of which for computation was first recognised by Milner [46].

The first, and more general form, we shall refer to as the $\cap$ type constructor. It has formation rule:

$$\frac{\begin{array}{l} A \textbf{ type} \\ |[\ x \in A\ \rhd\ B(x) \textbf{ type}\ ]| \end{array}}{\cap(A, B) \textbf{ type}} \ \cap\text{-formation}$$

The polymorphic function type may be viewed as a special case of $\forall$ type, whose objects are constant functions. The introduction rule is:

$$\frac{\begin{array}{l} |[\ x \in A \\ \rhd\ b \in B(x) \\ ]| \end{array}}{b \in \cap(A, B)} \ \cap\text{-introduction}$$

which should be compared with the introduction rule for the $\forall$ type:

0.0 $\quad |[\quad A \in U_1$

0.1.0 $\quad \triangleright \quad |[\quad a \in A$

$\qquad\qquad \triangleright \quad \{\ 0.1.0\ \}$

0.1.1 $\qquad\qquad a \in A$

$\qquad\quad]|$

$\qquad\quad \{\ 0.1,\ \Rightarrow\text{-}\mathbf{introduction}\ \}$

0.2 $\qquad \lambda([a]a) \in A \Rightarrow A$

$\quad]|$

$\quad \{\ 0,\ \cap\text{-}\mathbf{introduction}\ \}$

1 $\quad \lambda([a]a) \in \cap(U_1, [A]A \Rightarrow A)$

Figure 7: Derivation for polymorphic identity function.

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

The important difference between the two rules is that the $\cap$-introduction rule imposes the restriction that x may not appear free in the expression b. (It may, on the other hand, appear free in the type expression $B(x)$.) Thus b is an element of $\cap(A, B)$ if it is an element of each type in the family $B(x)$ where x ranges over elements of A. In particular, if some element a of type A is exhibited then b is an element of $B(a)$. This is expressed by the $\cap$-elimination rule:

$$\frac{\begin{array}{l} b \in \cap(A, B) \\ a \in A \end{array}}{b \in B(a)} \qquad \cap\text{-elimination}$$

The type $\cap$ is nothing more than the notion of polymorphic function from Martin-Löf's logical framework[22], but at the level of types rather than categories.

One use of $\cap$ is in the construction of objects which are not cluttered up with unnecessary type information. For example, given an arbitrary type A the identity function $\lambda([a]a)$ is an object of type $A \Rightarrow A$. We can quantify over the type argument using $\cap$ and construct the polymorphic identity function.

$$\lambda([a]a) \in \cap(U_1, [A]A \Rightarrow A)$$

The derivation is given in figure 7. By $\cap$-elimination, we can apply the polymorphic identity function to the argument $\mathbb{N} \in U_1$ giving:

$$\lambda([a]a) \in \mathbb{N} \Rightarrow \mathbb{N}$$

In the absence of the type $\cap$, we would justify step 1 of figure 7 by $\forall$-introduction giving:

$$\lambda([A]\lambda([a]a)) \in \forall(U_1,\ [A]A \Rightarrow A)$$

Using $\forall$ makes the type information explicit in the identity function itself.

Just as the type $\Rightarrow$ is the non-dependent form of the type $\forall$, there is a non-dependent form of $\cap$. The introduction rule is:

$$\frac{\begin{array}{l} |[\ x \in A \\ \rhd\ b \in B \\ ]| \end{array}}{b \in A \mapsto B} \qquad \mapsto\text{-introduction}$$

where x does not occur free in b and B. The elimination rule is:

$$\frac{\begin{array}{l} b \in A \mapsto B \\ a \in A \end{array}}{b \in B} \qquad \mapsto\text{-elimination.}$$

Initially, the type $\mapsto$ may seem to be of little use. The argument to a polymorphic function contributes type information only, but in the non-dependent form it does not even contribute type information (since the result type of the function is not dependent on its argument). The utility of $\mapsto$ is in the construction of functions that have constraints on their domain type. Say we have proved $b \in A \mapsto B$. Applying $\mapsto$-elimination then establishes $b \in B$, but only if we can construct an object a of type A (or, equivalently, the condition A is satisfied). For example, consider the head function over finite lists. We must restrict the domain to non-empty lists but we do not wish to impose any restrictions on the base type of the lists: we want to construct a truly *polymorphic* head function. The head function is defined as

$$hd(x) \equiv Listelim(x,\ \emptyset\text{-}elim(0),\ [a, l, h]a)$$

and we would like to prove, for arbitrary type A,

$$\begin{array}{l} |[\ \ l \in Set(List(A), [l]l \neq \mathbf{nil}) \\ \rhd\ \ hd(l) \in A \\ ]| \end{array}$$

The reader may be surprised to learn that this judgement is not provable in the current theory without polymorphic functions. The best we are able to do is

$$\begin{array}{l} |[\ \ l \in Set(List(A), [l]l \neq \mathbf{nil}) \\ \rhd\ \ Listelim(l,\ \lambda([x]\emptyset\text{-}elim(0)),\ [a, l, h]\lambda([x]a)).\lambda([x]0) \in A \\ ]| \end{array}$$

which is somewhat more complex. The problem is that we require as our elimination hypothesis

$$x \neq \mathbf{nil} \Rightarrow A$$

```
0.0          |[  $l \in Set(List(A), [l]l \neq \mathbf{nil})$
0.1.0        ▷  |[  $m \in List(A);\ n \in m \neq \mathbf{nil}$
0.1.1.0          ▷  |[  $x \in \mathbf{nil} \neq \mathbf{nil}$
                    ▷      { 0.1.1.0, absurdity }
0.1.1.1                 $0 \in \emptyset$
                           { 0.1.1.1, $\emptyset$-elimination }
0.1.1.2                 $\emptyset\text{-}elim(0) \in A$
                    ]|
                      { 0.1.1, $\mapsto$-introduction }
0.1.2               $\emptyset\text{-}elim(0) \in \mathbf{nil} \neq \mathbf{nil} \mapsto A$
0.1.3.0             |[  $a \in A;\ l \in List(A);\ h \in l \neq \mathbf{nil} \mapsto A$
0.1.3.1.0           ▷  |[  $x \in a :: l \neq \mathbf{nil}$
                       ▷      { 0.1.3.0 }
0.1.3.1.1                  $a \in A$
                       ]|
                          { 0.1.3.1, $\mapsto$-introduction }
0.1.3.2                $a \in a :: l \neq \mathbf{nil} \mapsto A$
                    ]|
                      { 0.1.0, 0.1.2, 0.1.3, List-elimination }
0.1.4               $Listelim(m, \emptyset\text{-}elim(0), [a, l, h]a) \in m \neq \mathbf{nil} \mapsto A$
                      { 0.1.4, defn. of hd, 0.1.0, $\mapsto$-elimination }
0.1.5               $hd(m) \in A$
                ]|
                  { 0.0, 0.1, Set-elimination }
0.2             $hd(l) \in A$
            ]|
```

Figure 8: Derivation of head function.

where $x \in List(A)$. We must perform *List* elimination on a function type and apply the result of the elimination to an object of the function's argument type. The outcome is an object containing unnecessary λ abstractions and applications.

Using the non-dependent polymorphic function type, we can establish the desired judgement. The proof is given in figure 8. More examples of the use of $\mapsto$ for this purpose can be found in section 5; for further discussion see [38].

4 Algorithm Design in Type Theory

This section is concerned with examining the relationship between the heuristics used in inductive proof[26,5] and the heuristics used in the development of loop invariants[27,21,4] in algorithm design. The problem we use as illustration 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. More specifically, suppose the candidates in a ballot are drawn from the type A and votes for each candidate are recorded in the list l of length n. The problem is to determine whether or not one of the elements of A occurs more than n **div** 2 times in l and if so to exhibit that element. For example, in the list

$$[a, b, d, a, a, c, b, a, b, a, a]$$

the element a occurs a majority of times (6 times in a list of length 11) but in the list

$$[a, b]$$

no element occurs a majority of times.

This problem is a particularly attractive one to consider for several reasons. First it is easily stated and readily understood. Second it is a problem for which all programmers are able to propose a solution within a space of a few minutes, and therefore one that is all too easily dismissed as "trivial" or "uninteresting". Nevertheless, the solution on which our development is based — described in [47] and originally due to J S. Moore [6] — is quite remarkable and not obvious. It is a solution that involves a transformation from a deterministic into a non-deterministic problem specification, and one that requires considerable creativity in the invention of an appropriate inductive hypothesis, but for which the resulting program is compact, elegant and — most importantly — difficult to understand by purely operational arguments.

In order to proceed more formally, we introduce the following context wherein it is assumed that A is a non-empty type with decidable equality, and l is a list of objects of type A.

$$\begin{array}{ll} |[& A \in U_1 \\ ; & eq \in \forall(A, [a]\forall(A, [b](a =_A b) \vee (a \neq_A b))) \\ ; & \alpha \in A \\ ; & l \in List(A) \\ \triangleright & \end{array}$$

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

$$Set(A, [x]majority(l, x)) \vee \neg Set(A, [x]majority(l, x)) \tag{6}$$

where

$$\begin{array}{lll} majority(l, x) & \equiv & no\text{-}of\text{-}occurrences(l, x) > length(l) \textbf{ div } 2 \\ no\text{-}of\text{-}occurrences(l, x) & \equiv & Listelim(l,\ 0,\ [a, m, h]\textbf{if } a = x \textbf{ then } h + 1 \textbf{ else } h) \\ length(l) & \equiv & Listelim(l,\ 0,\ [a, m, h]h + 1) \end{array}$$

Note that (6) is trivially true in classical mathematics; in constructive mathematics it is only true if one can provide a proof of either the proposition $Set(A, [x]majority(l, x))$ or its negation — i.e., exhibit a candidate receiving a majority of votes or prove that it is impossible to do so. Note also that an object in the right summand of (6) carries no computational content. What is significant is that the specification is deterministic: any two objects that satisfy the specification must be equal.

4.1 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. It is also the strategy used in specifying binary search when we seek an index to an ordered array which partitions all elements less than or equal to a given value x from those elements greater than x, rather than determining whether or not x occurs in the array[4]. 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[36]. In the present 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 the list l. This problem has specification:

$$\forall(A, [x]majority(l, x) \vee \neg majority(l, x)) \tag{7}$$

We leave it as an exercise for the reader to construct an object of (7).

Our solution to the majority-vote problem is based on combining a solution to (7) with a solution to the following:

$$Set(A, [x]pm(l, x)) \tag{8}$$

where definition of the predicate *pm* should be such that we can recover a solution to the original problem as follows. First, use the solution to (8) to generate an object a of A. Then subject a to the test specified by (7). If a is found to occur a majority of times in the list then injecting it into the left summand of (6) is clearly all that is required; otherwise, we wish to infer that no element of A can be a majority value. In summary, therefore, the element a should *exclude* all other elements from being majority values. Thus we take the following as our definition of *pm*.

$$pm(l, a) \equiv \neg majority(l, a) \Rightarrow \neg Set(A, [x]majority(l, x))$$

Of course, a pair of objects of types (7) and (8) is not the same as an object of (6). However such an object can be easily recovered. Specifically, the function

$$\lambda([a]\lambda([f]\vee\text{-}elim(f.a,\ [y]\mathbf{inl}(a),\ [z]\mathbf{inr}(\lambda([x]x)))))$$

is of type (8) $\Rightarrow$ (7) $\Rightarrow$ (6) as can be seen from the derivation given in figure 9.

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

$$majority(l, a) \vee \neg Set(A, [x]majority(l, x)) \tag{9}$$

Because a candidate obtaining a majority of votes is always unique, if one exists, (9) is another way of saying that a excludes all other candidates from being in the majority.

```
0.0        |[  $f \in \forall(A, [x]majority(l, x) \vee \neg majority(l, x))$
0.1        ;   $a \in Set(A, [x]pm(l, x))$
0.2.0      ▷   |[  $x \in A$
0.2.1          ;   $g \in \neg majority(l, x) \Rightarrow \neg Set(A, [x]majority(l, x))$
               ▷       { 0.0, 0.2.0, ∀-elim }
0.2.2              $f.x \in majority(l, x) \vee \neg majority(l, x)$
0.2.3.0            |[  $y \in majority(l, x)$
                   ▷       { 0.2.0, 0.2.3.0, Set-intro, inl-intro }
0.2.3.1                $\mathbf{inl}(x) \in (6)$
                   ]|
0.2.4.0            |[  $z \in \neg majority(l, x)$
                   ▷       { 0.2.1, 0.2.4.0, ⇒-elim }
0.2.4.1                $g.z \in \neg Set(A, [x]majority(l, x))$
                           { 0.2.4.1, section 3.4.1 }
0.2.4.2                $\lambda([x]x) \in \neg Set(A, [x]majority(l, x))$
                           { 0.2.4.2, inr-intro }
0.2.4.3                $\mathbf{inr}(\lambda([x]x)) \in (6)$
                   ]|
                       { 0.2.2, 0.2.3, 0.2.4, ∨-elim }
0.2.5              $\vee\text{-}elim(f.x, [y]\mathbf{inl}(x), [z]\mathbf{inr}(\lambda([x]x))) \in (6)$
               ]|
                   { 0.1, 0.2, Set-elim }
0.3            $\vee\text{-}elim(f.a, [y]\mathbf{inl}(a), [z]\mathbf{inr}(\lambda([x]x))) \in (6)$
           ]|
```

Figure 9: Problem Decomposition

4.2 Invariants versus Inductive Hypotheses

We choose to prove (8) by elimination on l (i.e. by induction over the structure of lists). The basis is trivial since no candidate can occur a majority of times in the empty list, and any object will do as our possible-majority candidate. Problems occur when we try to perform the induction step. Suppose that $h \in Set(A, [x]pm(m, x))$ for some $m \in List(A)$. How does one construct an object of $Set(A, [x]pm(a :: m, x))$? It is clear that more information is needed about the object h — we must strengthen our induction hypothesis.

In an imperative programming language, our aim would be to construct a loop that examines each element of the list and exhibits a possible-majority candidate at each iteration. The initialisation that precedes execution of the loop corresponds to the basis of the proof by induction, and the loop body to the proof of the inductive step. The notion of inductive hypothesis corresponds to the notion of invariant property. Strengthening the inductive hypothesis corresponds to introducing additional auxiliary variables into the computation.

Too strong a hypothesis would be the conjunction of (6) and

$$\forall(A, [x]majority(l, x) \vee \forall(A, [y]\neg majority(l, y)) \Rightarrow pm(l, x))$$

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 the existence of both a possible majority candidate and its number of occurrences in the array segment. This is 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.

A suitable hypothesis can be formulated by first examining some properties of pm and $majority$. Suppose $m \in List(A)$, $x \in A$ and $pm(m, x)$. Suppose also that $a \in A$. Previous remarks suggest that this information is insufficient to be able to deduce an object of $Set(A, [x]pm(a :: m, x))$, but what if we extend the list by one more element? Suppose $b \in A$. Is there a relationship between $pm(m, x)$ and $pm(a :: b :: m, x)$? Indeed there is: in the case that a and b are distinct. For in this case we observe that:

$$no\text{-}of\text{-}occurrences(a :: b :: m, x) \leq no\text{-}of\text{-}occurrences(m, x) + 1 \tag{10}$$

From (10) we can derive in turn:

$$majority(a :: b :: m, x) \Rightarrow majority(m, x) \tag{11}$$

and

$$pm(m, x) \Rightarrow pm(a :: b :: m, x) \tag{12}$$

These are proved as follows. First (11).

$$\begin{array}{ll}
 & majority(a :: b :: m, x) \\
\equiv & \qquad \{ \text{ definition } \} \\
 & no\text{-}of\text{-}occurrences(a :: b :: m, x) > length(a :: b :: m, x) \textbf{ div } 2 \\
\Rightarrow & \qquad \{ \text{ (10) and arithmetic } \} \\
 & no\text{-}of\text{-}occurrences(m, x) + 1 > (length(m) \textbf{ div } 2) + 1 \\
\equiv & \qquad \{ \text{ arithmetic, definition of majority } \} \\
 & majority(m, x)
\end{array}$$

Expanding the definition of $pm(a :: b :: m, x)$ we see that to prove (12) we have to prove the following.

$$pm(m, x) \Rightarrow \neg majority(a :: b :: m, x) \Rightarrow Set(A, [y]majority(a :: b :: m, y)) \Rightarrow \emptyset$$

Thus we make the following assumptions:

$$\begin{array}{ll}
|[& pm(m, x) \\
; & \neg majority(a :: b :: m, x) \\
; & y \in A \\
; & majority(a :: b :: m, y)
\end{array}$$

From the second and fourth assumptions it follows that

$$y \neq_A x$$

From (11) and the fourth assumption we also have

$$majority(m, y)$$

Thus, since majority values are always unique,

$$\neg majority(m, x)$$

But, expanding the definition of $pm(m, x)$, we obtain

$$\neg Set(A, [y]majority(m, y))$$

which clearly contradicts the earlier conclusion $majority(m, y)$; thus we have:

$$\emptyset$$

]|

Discharging the above assumptions by Set-elimination and $\Rightarrow$-introduction we have established (12).

The proposition (12) is only valid in the case that a and b are distinct, so we are still confronted with the computation of a $y \in A$ such that $pm(a :: a :: m, y)$. If we try extending the list m by yet one more element we will still face difficulties computing $pm(a :: a :: a :: m, y)$, and so on. Our problem has therefore generalised to the problem of, given $a \in A$ and $n \in \mathbb{N}$, compute y such that $pm(a^n :: m, y)$. That is, construct a function of type

$$\forall(A,\ [a]\forall(\mathbb{N},\ [n]Set(A, [y]pm(a^n :: m, y)))) \tag{13}$$

where $a^n :: m$ denotes a list consisting of n occurrences of a followed by the list m. Formally,

$$a^n :: m \equiv \mathbb{N}\text{-}elim(n,\ m,\ [k, h]a :: h)$$

Note that (13) is indeed a generalisation of our original problem since if f is an object of type (13) then

$$f.\alpha.0 \in Set(A, [y]pm(\alpha^0 :: m, y))$$

I.e.,

$$f.\alpha.0 \in Set(A, [y]pm(m, y))$$

Since we are in the midst of an exploratory investigation let us take the bold step of proposing (13) as the induction hypothesis.

4.3 Program development

Let $C(m)$ denote $\forall(A,\ [a]\forall(\mathbb{N},\ [n]Set(A, [y]pm(a^n :: m, y))))$. We propose the construction of an object of type $C(l)$ by List-elimination. That is, if we can construct programs ϕ and ψ such that

$$\phi \in C(\mathbf{nil}) \tag{14}$$

and

$$\begin{array}{ll} |[& b \in A;\quad m \in List(A); h \in C(m) \\ \triangleright & \psi \in C(b :: m) \\ ]| & \end{array} \tag{15}$$

then an application of List-elimination on $l \in List(A)$ will give us:

$$Listelim(l,\ \phi,\ [b, m, h]\psi) \in C(l) \tag{16}$$

and hence

$$Listelim(l,\ \phi,\ [b, m, h]\psi).\alpha.0 \in Set(A, [x]pm(l, x)) \tag{17}$$

Taking (14) first:

```
0.0     |[   $a \in A;\quad n \in \mathbb{N}$
        ▷      {trivially}
0.1          $pm(a^n :: \mathbf{nil}, a)$
               {0.0, 0.1, Set-intro.}
0.2          $a \in Set(A, [x]pm(a^n :: \mathbf{nil}, a))$
        ]|
```

$\{0.0, 0.2, \forall\text{-intro.}\}$

1 $\lambda([a]\lambda([n]a)) \in C(\mathbf{nil})$

For the inductive step, (15), we make the assumptions:

2.0 $|[\quad b \in A;\ \ m \in List(A); h \in C(m)$

2.1.0 $\triangleright\quad |[\quad a \in A;\ \ n \in \mathbb{N}$

$\triangleright$

and we try to construct an object of

$$Set(A, [x]pm(a^n :: b :: m, x)) \tag{18}$$

Recalling our remarks of the previous section, it is necessary to consider two cases, $a = b$ and $a \neq b$. By assumption we have

2.1.1 $eq.a.b \in (a =_A b) \vee (a \neq_A b)$

Considering the left summand first:

2.1.2.0 $|[\quad a =_A b$

2.1.2.1 $\triangleright\quad h.a.(n+1) \in Set(A, [x]pm(a^{n+1} :: m, x))$

Now we note that

$$\begin{array}{ll} & a^{n+1} :: m \\ = & \quad \{\text{trivially}\} \\ & a^n :: a :: m \\ = & \quad \{\text{assumption}\} \\ & a^n :: b :: m \end{array}$$

So we conclude

2.1.2.2 $h.a.(n+1) \in Set(A, [x]pm(a^n :: b :: m, x))$

$]|$

Now consider the right summand of (2.1.1):

2.1.3.0 $|[\quad (a \neq_A b)$

$\triangleright$

We now perform $\mathbb{N}$-elimination on n with induction hypothesis

$$D(k) \equiv Set(A, [x]pm(a^k :: b :: m, x))$$

(In fact we are only interested in whether $n = 0$ or $n \neq 0$ and we do not make use of the induction hypothesis. However the method we employ is technically preferable to such a case analysis.)

For the base case, then, we want to find an object of $D(0)$, i.e. $Set(A, [x]pm(b :: m, x))$. Now, 2.1.3.1 $h.b.1 \in Set(A, [x]pm(b^1 :: m, x))$

Thus, since $b^1 :: m = b :: m$,

2.1.3.2 $h.b.1 \in D(0)$

For the induction step we assume:

2.1.3.3.0 $|[\quad k \in \mathbb{N}$

$\triangleright$

(As remarked earlier we make no use of the induction hypothesis $D(k)$; we have therefore omitted it from our list of assumptions.)

We note that

2.1.3.3.1 $h.a.k \in Set(A, pm(a^k :: m, x))$

But since, by assumption, $a \neq_A b$, we can apply (12) to infer that:

2.1.3.3.2 $h.a.k \in Set(A, pm(a :: b :: a^k :: m, x))$

The property *pm* is, however, a property of bags rather than lists — it is independent of the order of elements in the list — and so:

2.1.3.3.3 $h.a.k \in Set(A, pm(a^k :: a :: b :: m, x))$

I.e.,

2.1.3.3.4 $h.a.k \in D(k+1)$

$]|$

By $\mathbb{N}$-elimination,

2.1.3.4 $\mathbb{N}\text{-}elim(n,\ h.b.1,\ [k, \_]h.a.k) \in Set(A, [x]pm(a^n :: b :: m, x))$

$]|$

Performing $\vee$-elimination on (2.1.1) gives:

2.1.4 $\vee\text{-}elim(eq.a.b,\ [\_]h.a.(n+1),\ [\_]\mathbb{N}\text{-}elim(n,\ h.b.1,\ [k, \_]h.a.k))$
$\in Set(A, [x]pm(a^n :: b :: m, x))$

$]|$

and thus by $\forall$-introduction

2.2 $\lambda([a]\lambda([n]\vee\text{-}elim(eq.a.b,\ [\_]h.a.(n+1),\ [\_]\mathbb{N}\text{-}elim(n,\ h.b.1,\ [k, \_]h.a.k))))$
$\in \forall(A,\ [a]\forall(\mathbb{N},\ [n]Set(A, [x]pm(a^n :: b :: m, x))))$

]|

As observed in (16) our program is obtained by applying List-elimination to steps 1 and 2, and applying the resulting program to α and 0. In full, the complete program to compute a possible-majority candidate given list l is as follows.

```
3       List-elim(l
                , λ([a]λ([n]a))
                , [b, m, h]λ([a]λ([n]∨-elim(eq.a.b
                                            , [_]h.a.(n + 1)
                                            , [_]ℕ-elim(n, h.b.1, [k, _]h.a.k)
                                            )))
                )
        .α.0
        ∈ Set(A, [x]pm(l, x))
```

If we consider the case when $l = \mathbf{nil}$, our program reduces (by **nil**-computation) to

$$\lambda([a]\lambda([n]a)).\alpha.0 \in Set(A, [x]pm(\mathbf{nil}, x))$$

and further to

$$\alpha \in Set(A, [x]pm(\mathbf{nil}, x))$$

The distinguished element α is in this sense a "default value", a guarantee that the program will always produce some value even though the list may be empty.

5 Binary Numerals

In this section, we apply the theory developed earlier to define two alternative formalisations of binary numerals. In the first formulation, we use a congruence rule to identify those binary numerals which differ only in the number of leading zeroes. In the second formulation, information loss is used to exclude numerals with leading zeroes from the type.

5.1 Binary Numerals as a Congruence Type

A binary numeral is a sequence of 1's and 0's in which leading 0's are insignificant. Thus 11 = 011 = 0011 etc. A binary numeral is, however, one particular interpretation of such a sequence. More generally we may regard such a sequence as denoting a polynomial over {0,1}; thus, 101 denotes $x^2 + 1$. We can define a type, called *BN* say, of sequences of 0's and 1's in which leading 0's are insignificant as follows.

$$\frac{}{BN\ \mathbf{type}} \qquad BN\text{-formation}$$

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

$$\frac{b \in BN}{b0 \in BN} \qquad 0\text{-introduction}$$

$$\frac{b \in BN}{b1 \in BN} \qquad 1\text{-introduction}$$

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

0- and 1- introduction construct a new numeral from an existing numeral by adding a 0 or 1, respectively, as the least significant digit. Given the four introduction rules, we derive four corresponding premises for the elimination rule. These premises state that to define a function over *BN* it is necessary to consider three cases — the case where the argument is Λ, the case where it is of the form $b0$ and the case where it is of the form $b1$ — and furthermore it is necessary to show that the insignificance of leading zeroes is respected. Specifically, we have the following rule.

$$\frac{\begin{array}{l} |[\ w \in BN \ \triangleright\ C(w)\ \mathbf{type}\ ]| \\ b \in BN \\ c \in C(\Lambda) \\ |[\ x \in BN; h \in C(x) \ \triangleright\ d(x,h) \in C(x0)\ ]| \\ |[\ x \in BN; h \in C(x) \ \triangleright\ e(x,h) \in C(x1)\ ]| \\ d(\Lambda, c) = c \in C(\Lambda) \end{array}}{\mathit{BNelim}(b, c, d, e)} \qquad BN\text{-elimination}$$

The three computation rules are summarised by the equations:

$$\begin{aligned} \mathit{BNelim}(\Lambda, c, d, e) &= c \in C(\Lambda) \\ \mathit{BNelim}(b0, c, d, e) &= d(b, \mathit{BNelim}(b, c, d, e)) \in C(b0) \\ \mathit{BNelim}(b1, c, d, e) &= e(b, \mathit{BNelim}(b, c, d, e)) \in C(b1) \end{aligned}$$

The type *BN* can be used to model some of the tasks that a hardware designer faces. Suppose that we regard objects of *BN* as binary representations of natural numbers; the task is to construct functions that represent the common arithmetic operations, addition, subtraction and so on. Here we shall describe the construction and verification of some operations on *BN*.

To verify that the constructed operations on BN do indeed represent operations on numbers it is necessary to relate $\mathbb{N}$ and BN. Thus we shall define an operation abs that maps an object b of BN to an object $abs(b)$ of $\mathbb{N}$. We also define and verify two operations inc and dec which, respectively, add one to and subtract one from a binary numeral.

The representation operation, abs, is defined to be

$$abs(b) \equiv BNelim(b, 0, [x, h]2 * h, [x, h]\mathbf{succ}(2 * h))$$

Note that to verify the well-definedness of abs we have to verify the following.

$$\begin{array}{ll} (a) & 0 \in N \\ (b) & |[\ x \in BN; h \in N \ \triangleright\ 2 * h \in N\ ]| \\ (c) & |[\ x \in BN; h \in N \ \triangleright\ \mathbf{succ}(2 * h) \in N\ ]| \\ (d) & 0 = 2 * 0 \in N \end{array}$$

Clause (d) is of course the appropriate instance of the leading zeroes premise. For brevity we denote $abs(b)$ by b'.

Consider now the operation inc which takes an object b in BN and returns the numeral one greater than b. It is defined as

$$inc(b) \equiv BNelim(b, \Lambda 1, [x, h]x1, [x, h]h0))$$

or in clausal form

$$\begin{array}{rcl} inc(\Lambda) & = & \Lambda 1 \\ inc(b0) & = & b1 \\ inc(b1) & = & (inc(b))0 \end{array}$$

Formally we can verify inc by establishing the judgement

$$|[\ b \in BN \ \triangleright\ inc(b) \in Set(BN, [y]y' =_{\mathbf{N}} \mathbf{succ}(b'))\ ]|$$

That is, the natural number corresponding to $inc(b)$ must be one greater than the natural number corresponding to b. The proof is detailed in figure 10. The structure of the argument reflects the steps taken to construct the function inc. Steps 0.3, 0.4 and 0.5 establish the correctness of the constructions for the cases Λ, $b0$ and $b1$ respectively. Step 0.6 proves the operation respects the leading zeroes constraint.

Complementary to inc is the function dec that subtracts 1 from a binary numeral b. It is defined as

$$dec(b) \equiv BNelim(b, \emptyset\text{-}elim(0), [x, h]h1, [x, h]x0))$$

or in clausal form

$$\begin{array}{rcl} dec(\Lambda) & = & \emptyset\text{-}elim(0) \\ dec(b0) & = & (dec(b))1 \\ dec(b1) & = & b0 \end{array}$$

Note that dec only produces sensible answers when applied to non-empty numerals. See section 3.1.4 for a justification of the use of 0 as the argument to $\emptyset$-$elim$ in the Λ clause. We can verify

```
0.0     |[  $b \in BN$
        ▷      { Λ-intr, 1-intr }
0.1         $\Lambda 1 \in BN$
0.2             $\mathbf{succ}(\Lambda')$
            $=_{\mathbb{N}}$        { BN-comp, subst }
                $\mathbf{succ}(0)$
            $=_{\mathbb{N}}$        { BN-comp, ℕ-comp }
                 $(\Lambda 1)'$
              { 0.1, 0.2, Set-intr }
0.3         $\Lambda 1 \in Set(BN, [y]y' = \mathbf{succ}(\Lambda'))$
0.4.0       |[  $x \in BN; h \in Set(BN, [y]y' = \mathbf{succ}(x'))$
            ▷      { 0.4.0, 1-intr }
0.4.1           $x1 \in BN$
0.4.2               $\mathbf{succ}((x0)')$
                $=_{\mathbb{N}}$        { BN-comp, subst }
                    $\mathbf{succ}(2 * x')$
                $=_{\mathbb{N}}$        { BN-comp, ℕ-comp }
                     $(x1)'$
                  { 0.4.1, 0.4.2, Set-intr }
0.4.3           $x1 \in Set(BN, [y]y' = \mathbf{succ}((x0)'))$
            ]|
0.5.0       |[  $x \in BN; h \in Set(BN, [y]y' = \mathbf{succ}(x'))$
            ▷      { 0.5.0, Set-elim, 0-intr }
0.5.1           $h0 \in BN$
0.5.2               $\mathbf{succ}((x1)')$
                $=_{\mathbb{N}}$        { BN-comp, subst }
                    $\mathbf{succ}(\mathbf{succ}(2 * x'))$
                $=_{\mathbb{N}}$        { ℕ-comp }
                    $2 * \mathbf{succ}(x')$
                $=_{\mathbb{N}}$        { 0.5.0, Set-elim, subst }
                    $2 * h'$
                $=_{\mathbb{N}}$        { BN-comp }
                     $(h0)'$
                  { 0.5.1, 0.5.2, Set-intr }
0.5.3           $h0 \in Set(BN, [y]y' = \mathbf{succ}((x1)'))$
            ]|
              { 0.3, refl }
0.6         $\Lambda 1 = \Lambda 1 \in Set(BN, [y]y' = \mathbf{succ}(\Lambda'))$
              { 0.0, 0.3, 0.4, 0.5, 0.6, BN-elim }
0.7         $inc(b) \in Set(BN, [y]y' = \mathbf{succ}(b'))$
        ]|
```

Figure 10: Verification of *inc*

elimination hypothesis:

```
|[ $b \in BN$
▷ $b \neq \Lambda \mapsto Set(BN, [y]inc(y) = b)$ **type**
]|

                 { 1-premise }
0.0       |[   $x \in BN$
0.1       ;    $h \in x \neq \Lambda \mapsto Set(BN, [y]inc(y) = x)$
0.2.0     ▷    |[   $u \in x1 \neq \Lambda$
               ▷        { 0.0, 0-intr }
0.2.1               $x0 \in BN$
                        { BN-comp }
0.2.2               $inc(x0) = x1$
                        { 0.2.1, 0.2.2, Set-intr }
0.2.3               $x0 \in Set(BN, [y]inc(y) = x1)$
               ]|
                  { 0.2, $\mapsto$-intr }
0.3            $x0 \in x1 \neq \Lambda \mapsto Set(BN, [y]inc(y) = x1)$
          ]|
                  { leading zeroes premise }
1.0       |[   $u \in \Lambda \neq \Lambda$
          ▷        { $\Lambda$-intr, refl }
1.1            $\Lambda = \Lambda$
                  { 1.0, 1.1, $\neg$-elim }
1.2            $0 \in \emptyset$
                  { 1.2, absurdity }
1.3            $\emptyset\text{-}elim(0) = \emptyset\text{-}elim(0)1 \in Set(BN, [y]inc(y) = \Lambda)$
          ]|
               { 1, $\mapsto$-intr }
2         $\emptyset\text{-}elim(0) = \emptyset\text{-}elim(0)1 \in \Lambda \neq \Lambda \mapsto Set(BN, [y]inc(y) = \Lambda)$
```

Figure 11: Verification of *dec*

dec by establishing the judgement

|[$b \in Set(BN, [y]y \neq \Lambda)$
▷ $dec(b) \in Set(BN, [y]inc(y) = b)$
]|

We actually prove

|[$b \in BN$
▷ $dec(b) \in b \neq \Lambda \mapsto Set(BN, [y]inc(y) = b)$
]|

from which the desired judgement easily follows (section 3.4.3). The details of verifying the 1-premise and the leading zeroes premise of the elimination rule are given in figure 11.

The operations *inc* and *dec* are the inverses of each other. Denoting composition of functions by $\circ$, we have the properties

$$dec \circ inc = \lambda([x]x) \in BN \Rightarrow BN$$

dec composed with *inc* is an identity function over binary numerals, and

$$inc \circ dec = \lambda([x]x) \in Set(BN, [y]y \neq \Lambda) \Rightarrow Set(BN, [y]y \neq \Lambda)$$

inc composed with *dec* is an identity function over non-empty numerals.

5.2 Binary Numerals Via Information Loss

In section 5.1, the type of binary numerals was defined so that any sequence of 0's and 1's is a valid numeral. A congruence rule was used to identify those numerals differing only in the number of leading zeroes. An alternative approach is to define the type so that those numerals containing leading zeroes are not valid members of the type. In this section, we give such a formulation of binary numerals using information loss to exclude leading zeroes.

The formation rule and the Λ- and 1- introduction rules are the same as for *BN*.

$$\frac{}{BN' \text{ type}} \qquad BN'\text{-formation}$$

$$\frac{}{\Lambda \in BN'} \qquad \Lambda\text{-introduction}$$

$$\frac{b \in BN'}{b1 \in BN'} \qquad 1\text{-introduction}$$

The 0-introduction rule is more complex. In *BN*, given some existing numeral b we simply construct the numeral $b0$. However, if b is Λ, the invalid numeral $\Lambda 0$ containing a leading zero is constructed. In order to exclude this possibility, the rule is strengthened to

$$\frac{\begin{array}{l} b \in BN' \\ p \in b \neq_{BN'} \Lambda \end{array}}{b0 \in BN'} \qquad 0\text{-introduction}$$

The object p in the second premise of 0-introduction does not appear in the conclusion of the rule. Thus, BN' exhibits information loss. Note that p is an object of a negated equality type. Such types are computationally redundant so this is a very simple case of information loss.

Each distinct term built up from the introduction rules for BN' denotes a distinct numeral. The leading zeroes congruence rule is unnecessary.

The elimination rule obtained from the introduction rules is

$$\frac{\begin{array}{l} |[\; w \in BN' \;\triangleright\; C(w) \textbf{ type }]| \\ b \in BN' \\ c \in C(\Lambda) \\ |[\; x \in BN' \;;\; h \in C(x) \;;\; p \in x \neq_{BN'} \Lambda \\ \triangleright\; d(x,h) \in C(x0) \\ ]| \\ |[\; x \in BN' \;;\; h \in C(x) \;\triangleright\; e(x,h) \in C(x1) \;]| \end{array}}{BN'elim(b,c,d,e) \in C(b)} \quad BN'\text{-elimination}$$

Note how the premise $p \in b \neq \Lambda$ of 0-introduction becomes an assumption of the 0-premise in the elimination rule, but the object p does not appear in the consequent of the premise because it does not appear in the conclusion of the 0-introduction rule.

The three computation rules are summarised by the equations:

$$\begin{aligned} BN'elim(\Lambda, c, d, e) &= c \in C(\Lambda) \\ BN'elim(b0, c, d, e) &= d(b, BN'elim(b, c, d, e)) \in C(b0) \\ BN'elim(b1, c, d, e) &= e(b, BN'elim(b, c, d, e)) \in C(b1) \end{aligned}$$

The 0-computation rule has all the premises of BN'-elimination plus

$$p \in b \neq \Lambda$$

We now compare the two formalisations of binary numerals by redoing the fragment of theory developed for BN in BN'.

The definition of abs is the same in BN' as in BN. Its construction is identical except that the leading zeroes premise of BN, namely $0 = 2 * 0 \in \mathbb{N}$, disappears.

The definition and specification of inc are similarly unchanged in BN'.

$$inc(b) \equiv BN'elim(b, \Lambda 1, [x,h]x1, [x,h]h0)$$

$$|[\; b \in BN' \;\triangleright\; inc(b) \in Set(BN', [y]y' =_{\mathbf{N}} \mathbf{succ}(b'))\;]|$$

The verification of inc in BN' is given in figure 12.

The proofs of inc in BN and BN' differ in three ways:

- the proof of the 0-premise in BN' (step 0.2) has the extra assumption $z \in x \neq \Lambda$. Such an assumption is essential for constructing the numeral $x0$ in BN', though not of course in BN.

```
0.0      |[  $b \in BN'$
         ▷      { similar to figure 10, steps 0.1-0.3 }
0.1          $\Lambda 1 \in Set(BN', [y]y' = \mathbf{succ}(\Lambda'))$
                { similar to figure 10, step 0.4 }
0.2.0        |[  $x \in BN'$ ; $h \in Set(BN', [y]y' = \mathbf{succ}(x'))$ ; $z \in x \neq \Lambda$
0.2.1        ▷   $x1 \in Set(BN', [y]y' = \mathbf{succ}((x0)'))$
             ]|
0.3.0        |[  $x \in BN'$ ; $h \in Set(BN', [y]y' = \mathbf{succ}(x'))$
             ▷      { 0.3.0, Set-elim }
0.3.1            $h' = \mathbf{succ}(x')$
                    { 0.3.1, $\mathbb{N}$-individuality }
0.3.2            $h' \neq 0$
                    { $\neg$-intr, 0.3.2, absurdity }
0.3.3            $h \neq \Lambda$
                    { 0.3.3, 0-intr }
0.3.4            $h0 \in BN'$
                    { 0.3.1, $BN'$-comp, subst }
0.3.5            $\mathbf{succ}((x1)') = (h0)'$
                    { 0.3.4, 0.3.5, Set-intr }
0.3.6            $h0 \in Set(BN', [y]y' = \mathbf{succ}((x1)'))$
             ]|
                { 0.0, 0.1, 0.2, 0.3, $BN'$-elim }
0.4          $inc(b) \in Set(BN', [y]y' = \mathbf{succ}(b'))$
         ]|
```

Figure 12: Verification of *inc* in *BN'*

- to justify the 1-premise in both proofs, it is necessary to establish the equality

  $$(h0)' =_{\mathbf{N}} \mathbf{succ}((x1)')$$

  One step in its derivation involves establishing

  $$(h0)' =_{\mathbf{N}} 2 * h'$$

  In *BN*, we simply apply the 0-computation rule. In *BN'*, however, we may only use the 0-computation rule when $h \neq \Lambda$. Extra work is needed to establish this fact.
- the leading zeroes premise of *BN* (step 0.6) does not appear in *BN'*. In this example, it is trivial to justify.

The second point above is a commonly occurring problem when reasoning in *BN'*. Whenever we want to use the 0-computation rule with $b0$, it is necessary to explicitly show $b \neq \Lambda$. In *BN*, it is sufficient to establish $b \in BN$.

The definition of *dec* is more complex in *BN'*. Consider subtracting 1 from the numeral $b1$. In *BN*, we simply return $b0$. If b is Λ, the constructed numeral $\Lambda 0$ is invalid in *BN'*. Thus, $b0$ is only returned if $b \neq \Lambda$, otherwise Λ is returned. The definition is

$$dec(b) \equiv BN'elim(b, \emptyset\text{-}elim(0), [x,h]h1, [x,h]\mathbf{if}\ x = \Lambda\ \mathbf{then}\ \Lambda\ \mathbf{else}\ x0)$$

The correctness condition for *dec* is the same as in *BN*. The justification of the 1-premise is detailed in figure 13. It is noticably more complex than the derivation in *BN* (step 0 of figure 11) as a direct result of the more complex definition of *dec* required in *BN'*.

The verification of *dec* generalises the problem associated with reasoning in *BN'* mentioned earlier. The general form of the 1-premise is

$$|[\ x \in BN'\ ;\ h \in C(x)\ \triangleright\ d(x,h) \in C(x1)\ ]|$$

If the justification of this premise requires, at any point, the construction of the numeral $x0$, it is necessary to establish $x \neq \Lambda$. Unless we can establish $x \neq \Lambda$ from the induction hypothesis $y \in C(x)$, it is necessary to do case analysis on x (complicating the proof and the derived program). *dec* is an example. Even when $x \neq \Lambda$ is a consequence of $y \in C(x)$, it must still be derived (complicating the proof but not the derived program). *inc* is an example.

Of course, the leading zeroes premise must be established when verifying *dec* in *BN*. Although easy in this example, consider the situation where one wants to prove a property of binary numerals, which is functional in character, by elimination. The leading zeroes premise then involves reasoning about equality between functions. In such a situation, the effort required to establish the leading zeroes premise becomes more significant and *BN'* would compare more favourably with *BN*.

We can view the congruence rules of a congruence type as defining equivalence classes of objects constructed from the other introduction rules. When these equivalence classes contain unique representatives, the congruence type can also be defined via information loss. Namely, that type which excludes all objects but the representatives of the equivalence classes. For example, the unique representatives of *BN* are those numerals without leading zeroes, which is just the type *BN'*. Types such as finite bags and finite sets defined in section 3.3 whose equivalence classes do not contain unique representatives can not be defined via information loss.

elimination hypothesis:

```
|[ $b \in BN$
▷ $b \neq \Lambda \mapsto Set(BN, [y]inc(y) = b)$ **type**
]|

                  { 1-premise }
0.0        |[   $x \in BN'$
0.1        ;    $h \in x \neq \Lambda \mapsto Set(BN', [y]inc(y) = x)$
0.2.0      ▷    |[   $u \in x1 \neq \Lambda$
                ▷       { $BN'$-elim }
0.2.1                $x = \Lambda \vee x \neq \Lambda$
0.2.2.0              |[   $x = \Lambda$
                     ▷       { $BN'$-comp, 0.2.2.0, 1-intr, subst }
0.2.2.1                   $inc(\Lambda) = \Lambda 1 = x1 \in BN'$
                             { $\Lambda$-intr, 0.2.2.1, $Set$-intr }
0.2.2.2                   $\Lambda \in Set(BN', [y]inc(y) = x1)$
                     ]|
0.2.3.0              |[   $x \neq \Lambda$
                     ▷       { 0.0, 0.2.3.0, 0-intr }
0.2.3.1                   $x0 \in BN'$
                             { $BN'$-comp }
0.2.3.2                   $inc(x0) = x1$
                             { 0.2.3.1, 0.2.3.2, $Set$-intr }
0.2.3.3                   $x0 \in Set(BN', [y]inc(y) = x1)$
                     ]|
                        { 0.2.1, 0.2.2, 0.2.3, Bool-elim }
0.2.4                **if** $x = \Lambda$ **then** $\Lambda$ **else** $x0 \in Set(BN', [y]inc(y) = x1)$
                ]|
                   { 0.2, $\mapsto$-intr }
0.3             **if** $x = \Lambda$ **then** $\Lambda$ **else** $x0 \in x1 \neq \Lambda \mapsto Set(BN', [y]inc(y) = x1)$
           ]|
```

Figure 13: Verification of *dec* in *BN'*

In defining BN', we have used information loss for a very specific purpose. Namely, to exclude certain objects from a type. These objects are, strictly speaking, valid objects of the type but their inclusion induces the wrong equality relation. When information loss is used for this purpose, an alternative formulation as a congruence type is possible. Namely, use congruence rules to identify each excluded object with the included object it is equal to.

6 Mutually Recursive Types

A recursively defined type, such as the natural numbers, is defined in terms of itself; thus, for example, we say that zero is a natural number, and that the successor of a natural number is also a natural number. This principle of inductive definition can easily be generalised to accommodate collections of types defined in terms of one another. We say that a collection of types is mutually recursive if each type in the collection is defined in terms of some other types in the collection. As data structures, such types have many applications in computing science: the two examples which we present in this section are trees and forests, and derivation trees for context-free grammars.

Mutual recursion introduces nothing substantially new into type theory. What innovations there are, reside in the elimination and computation rules. Ordinarily, a type's elimination rule contains one premise for each of its introduction rules; with a collection of mutually recursive types, the elimination rule for one type contains premises related to the introduction rules of other types in the collection as well. Induction hypotheses occur in the premises of the elimination rules in such a way as reflects the mutually recursive nature of the types, and this allows the construction of recursive functions. In fact, the non-canonical constants defined by the elimination and computation rules constitute a collection of mutually recursive functions.

The simplest way to explain the use of mutually recursive types in type theory is by example. Below, we present a trees-and-forests data structure, in which a node of a tree governs a list of subtrees (hence, a tree structure with an arbitrary branching factor), and, more generally, a mutually recursive collection of types which represent the derivation trees of a given context-free grammar. Both examples are brief, intended only to convey the basic principles of the mutually recursive definition of types; they are supplemented, however, by an extended example of an application of the trees and forests data structure, in which we construct a search algorithm commonly used in games-playing programs.

6.1 Trees and Forests

Our first example, then, is a trees and forests data structure which is parameterised by a base type. So much is expressed by the formation rules of the types:

$$\frac{A \ \textbf{type}}{\mathit{Tree}(A) \ \textbf{type}} \qquad \text{Tree-formation}$$

$$\frac{A \ \textbf{type}}{\mathit{Forest}(A) \ \textbf{type}} \qquad \text{Forest-formation}$$

A tree consists of an element of the base type together with a forest of subtrees:

$$\frac{a \in A \qquad f \in \mathit{Forest}(A)}{\mathit{node}(a, f) \in \mathit{Tree}(A)} \quad \mathit{node}\text{-introduction}$$

and a forest is a list-like structure of trees:

$$\frac{}{\mathbf{nilf} \in \mathit{Forest}(A)} \quad \mathbf{nilf}\text{-introduction}$$

$$\frac{t \in \mathit{Tree}(A) \qquad f \in \mathit{Forest}(A)}{t : f \in \mathit{Forest}(A)} \quad :\text{-introduction}$$

where **nilf** denotes the empty forest. These types are mutually recursive in that a tree may occur as a subexpression of a forest, and vice-versa. The elimination rules for the types introduce two non-canonical constants, *Telim* and *Felim*, which are themselves mutually recursive: evaluation of *Telim* on an object $t \in \mathit{Tree}(A)$ may involve a call of *Felim* on the forest which is a subexpression of t, and similarly for *Felim*. For this reason, premises pertaining to the forest introduction rules are included in the premises of Tree-elimination, and a premise pertaining to *node*-introduction is included in Forest-elimination: in other words, the minor premises of the two elimination rules are identical. There are three minor premises to each rule: one for trees and two for forests. This means that in order to prove a property $P(t)$ for some $t \in \mathit{Tree}(A)$, one must also prove that some other property, $Q(f)$, holds for all $f \in \mathit{Forest}(A)$. Here, then, are the elimination rules:

$$\frac{\begin{array}{l} |[\ x \in \mathit{Tree}(A) \ \triangleright\ P(x)\ \mathbf{type}\ ]| \\ |[\ x \in \mathit{Forest}(A) \ \triangleright\ Q(x)\ \mathbf{type}\ ]| \\ t \in \mathit{Tree}(A) \\ |[\ x \in A;\ y \in \mathit{Forest}(A);\ hy \in Q(y) \\ \triangleright\ a(x, y, hy) \in P(\mathit{node}(x, y)) \\ ]| \\ b \in Q(\mathbf{nilf}) \\ |[\ x \in \mathit{Tree}(A);\ y \in \mathit{Forest}(A);\ hx \in P(x);\ hy \in Q(y) \\ \triangleright\ c(x, y, hx, hy) \in Q(x : y) \\ ]| \end{array}}{\mathit{Telim}(t, a, b, c) \in P(t)} \quad \mathit{Tree}\text{-elim}$$

and Forest-elimination differs only in its major premise and its conclusion:

$$\frac{\begin{array}{l} |[\ x \in \mathit{Tree}(A)\ \triangleright\ P(x)\ \mathbf{type}\ ]| \\ |[\ x \in \mathit{Forest}(A)\ \triangleright\ Q(x)\ \mathbf{type}\ ]| \\ f \in \mathit{Forest}(A) \\ |[\ x \in A;\ y \in \mathit{Forest}(A);\ hy \in Q(y) \\ \triangleright\ a(x,y,hy) \in P(\mathit{node}(x,y)) \\ ]| \\ b \in Q(\mathbf{nilf}) \\ |[\ x \in \mathit{Tree}(A);\ y \in \mathit{Forest}(A);\ hx \in P(x);\ hy \in Q(y) \\ \triangleright\ c(x,y,hx,hy) \in Q(x:y) \\ ]| \end{array}}{\mathit{Felim}(f,a,b,c) \in Q(f)}\ \mathit{Forest}\text{-elim}$$

(To be consistent we should write *Tree-elim*(...) and *Forest-elim*(...). For brevity we prefer *Telim*(...) and *Felim*(...).)

The computation rules are summarised by the following equations:

$$\begin{aligned} \mathit{Telim}(\mathit{node}(x,y),a,b,c) &= a(x,y,\mathit{Felim}(y,a,b,c)) \\ \mathit{Felim}(\mathbf{nilf},a,b,c) &= b \\ \mathit{Felim}(x:y,a,b,c) &= c(x,y,\mathit{Telim}(x,a,b,c),\mathit{Felim}(y,a,b,c)) \end{aligned}$$

The mutually recursive nature of *Telim* and *Felim* is evident in these equations. The two elimination hypotheses—P and Q—of the elimination rules, if chosen appropriately, allow for very elegant proofs when reasoning about trees and forests, as we shall presently see. For the moment, before moving on to context-free grammars, we content ourselves with presenting a function which emphasises the list-like structure of forests. The function is like the function "*map*" defined on lists: it takes as arguments a function $g \in \mathit{Tree}(A) \Rightarrow B$ and a forest, and, applying g to each tree in the forest, concatenates the results into a list. Since we shall have cause to refer to this function later, we call it "*mapforest*" and define it thus:

$$\begin{array}{lll} \mathit{mapforest} \equiv & \lambda([g]\lambda([f]\mathit{Felim}(f & ,[x,y,hy]x \\ & & ,\mathbf{nil} \\ & & ,[x,y,hx,hy](g.x) :: hy \\ & &) \\ &) & \\ &) & \\ & \in\ (\mathit{Tree}(A) \Rightarrow B) \Rightarrow \mathit{Forest}(A) \Rightarrow \mathit{List}(B). & \end{array}$$

We derive the function (see figure 14) by assuming $g \in \mathit{Tree}(A) \Rightarrow B$ and $f \in \mathit{Forest}(A)$, and then performing Forest-elimination on f. As noted above, two of the minor premises of Forest-elimination pertain to forests—these we use to construct an object of $\mathit{List}(B)$—but there is also one premise which pertains to trees and we are required to prove some property, P, of trees. This latter property is superfluous, since our function *mapforest* requires no information concerning the individual trees which constitute the forest to which it is applied. In this case, we select some trivial proposition, here A which we prove from the assumptions of the premise pertaining to trees.

```
0.0          |[  $g \in Tree(A) \Rightarrow B$
0.1.0      ▷   |[  $f \in Forest(A)$
                ▷      { Forest-elimination, node-premise, prove $A$ }
0.1.1.0              |[   $x \in A$; $y \in Forest(A)$; $hy \in List(B)$
                     ▷        { trivially, from assumptions }
0.1.1.1                   $x \in A$
                     ]|
                          { Forest-elimination, nilf-premise, nil-intro }
0.1.2                $\mathbf{nil} \in List(B)$
                          { Forest-elimination, :-premise }
0.1.3.0              |[   $x \in Tree(A)$; $y \in Forest(A)$; $hx \in A$; $hy \in List(B)$
                     ▷        { function application, 0.0, 0.1.3.0 }
0.1.3.1                   $g.x \in B$
                              { ::-intro, 0.1.3.1, 0.1.3.0 }
0.1.3.2                   $(g.x) :: hy \in List(B)$
                     ]|
                          { Forest-elim, 0.1.1, 0.1.2, 0.1.3 }
0.1.4                $Felim(f, [x, y, hy]x, \mathbf{nil}, [x, y, hx, hy](g.x) :: hy) \in List(B)$
                ]|
             ]|
```

Figure 14: derivation of *mapforest*

It is worth noting that if a function $g \in Tree(A) \Rightarrow B$ is such that

$$g.(node(x, y)) = e(x, y) \in B$$

for some expression, e, then, for all $f \in Forest(A)$, $mapforest.g.f$ is equal to:

$$\begin{array}{ll} Felim(f & , [x, y, hy]e(x, y) \\ & , \mathbf{nil} \\ & , [x, y, hx, hy]hx :: hy \\ ). & \end{array}$$

The reader may care to prove the equality by using Forest-elimination to construct the above object as an element of $Set(List(B), [l]l = mapforest.g.f)$. The proof is simple, but provides a good example of how the two elimination hypotheses of the elimination rules can work in tandem.

6.2 CFG's and mutually recursive types

A possible application of mutually recursive types is in the development of parsing algorithms; see, for example, Chisholm [9]. We outline here a method for constructing a collection of types with which to represent derivation trees for a given context-free grammar. If the grammar is mutually recursive (two or more nonterminals are reachable from each other), then so too will be the collection of types which is constructed.

For each nonterminal symbol, "A", we introduce a type constructor A whose formation rule is the axiom A **type**. This type will have one introduction rule for each production of the grammar in which "A" occurs to the left of the rewrite arrow. Such a production will be of the form:

$$\mathrm{A} \longrightarrow x_0 \mathrm{A}_1 x_1 \ldots \mathrm{A}_n x_n$$

for $0 \leq n$ and where each A_i is a nonterminal and each x_i is a string of terminal symbols. The corresponding introduction rule will be:

$$\frac{\begin{array}{l} a_1 \in A_1 \\ \vdots \\ a_n \in A_n \end{array}}{\tau(a_1, \ldots, a_n) \in A}$$

where τ is a unique object-constructor and each A_i is the type constructor corresponding to the nonterminal "A_i". The elimination rule can thence be constructed as with trees and forests above: for two nonterminals "A" and "B", if "B" is reachable from "A" and "A" is reachable from "B", then premises pertaining to the introduction rules of B will be included in the premises of the elimination rule for A, and vice-versa, and in proving a property, P, of objects of type A, it will be necessary to prove a complementary property Q of objects of type B.

The objects of the types represent derivation trees in that each object-constructor is associated with a production of the grammar, and each of the subexpressions which it governs represents, in turn, an instance of a nonterminal which occurs to the right of the rewrite arrow in that production.

As an example, consider the following simplified fragment of the syntax of Pascal type declarations (from Jensen & Wirth [34]), where names between angle brackets denote nonterminals, and all other symbols are terminals.

$$\begin{array}{l}
\langle\text{Pascaltype}\rangle \longrightarrow \textbf{record}\ \langle\text{Fieldlist}\rangle\ \textbf{end} \\
\vdots \\
\langle\text{Fieldlist}\rangle \longrightarrow \langle\text{Recordsection}\rangle\ ;\ \langle\text{Fieldlist}\rangle \\
\vdots \\
\langle\text{Recordsection}\rangle \longrightarrow \langle\text{Id}\rangle\ :\ \langle\text{Pascaltype}\rangle
\end{array}$$

We wish to introduce types corresponding to the nonterminals above; call these *Ptype*, *Flist*, *Rsect* and *Id*. These types will have (among others) the following introduction rules:

$$\frac{f \in \mathit{Flist}}{\mathbf{rec}(f) \in \mathit{Ptype}}$$

$$\frac{\begin{array}{l} r \in \mathit{Rsect} \\ f \in \mathit{Flist} \end{array}}{r; f \in \mathit{Flist}}$$

$$\frac{x \in Id \qquad t \in Ptype}{x : t \in Rsect}$$

Since the nonterminals "Pascaltype", "Fieldlist" and "Recordsection" are all reachable from each other, their corresponding types are mutually recursive, and so the elimination rules for these types will contain premises pertaining to each of these introduction rules. Thus, the elimination rule for *Ptype* will have the form:

$$\frac{\begin{array}{l} |[\; x \in Ptype \;\triangleright\; P(x)\ \mathbf{type}\;]| \\ |[\; x \in Flist \;\triangleright\; Q(x)\ \mathbf{type}\;]| \\ |[\; x \in Rsect \;\triangleright\; R(x)\ \mathbf{type}\;]| \\ t \in Ptype \\ |[\; x \in Flist;\ hx \in Q(x) \\ \triangleright\; a(x, hx) \in P(\mathbf{rec}(x)) \\ ]| \\ \vdots \\ |[\; x \in Rsect;\ y \in Flist;\ hx \in R(x);\ hy \in Q(y) \\ \triangleright\; b(x, y, hx, hy) \in Q(x; y) \\ ]| \\ \vdots \\ |[\; x \in Id;\ y \in Ptype;\ hy \in P(t) \\ \triangleright\; c(x, y, hy) \in R(x : y) \\ ]| \end{array}}{Ptype\text{-}elim(t, a, \ldots, b, \ldots, c) \in P(t)}\ Ptype\text{-elim}$$

An obvious corollary of this example is that it is possible to construct types to represent derivation trees for context-free programming languages. The elimination rules then provide a means of reasoning about programs written in the language, one possible application being to formalise the language's denotational semantics.

It should be noted that what we have presented above is just a method for constructing a collection of types, and does not allow one to reason *about* context-free grammars within the theory. However, Synek and Petersson [55] have introduced into the theory a tree type which is a generalisation of the well-ordering type, and which can be used to represent mutually recursive data structures. They claim that with this type, it is possible to reason about the data structures themselves, rather than just about the objects of the data structures.

6.3 An application: games playing

We now present an application of the trees and forests data structure; to wit, the construction of a game-tree (a tree the paths through which represent admissible sequences of moves in a given game) and a search algorithm usually known as "*minimax*" which is often used in games-playing programs. Our purpose, however, is still primarily pædagogic, so we proceed slowly at first and construct some simple, generally-useful functions which will be combined in the end to form the minimax algorithm.

There is much to be said for this approach to program development. On our first attempt, we tried to derive the algorithm all in one go, which resulted in an ungainly derivation of a discouraging length. Our subsequent attempt resulted in what follows. The algorithm was divided into constituent functions of manageable size, and as each function was derived, we proved that it enjoyed certain properties which were useful when we came to combine them into larger functions. The derivations we give below are pleasantly short and quite readable, though we maintain a high degree of formality at each step.

Our program development is similar to that of Hughes in [32], although in that paper, the functions are written in a programming language with infinite objects, which allows a function to be more freely decomposed into constituent parts. The reason for this greater freedom is that, given a language with infinite objects, a non-terminating function may be composed with other functions in such a way that the composite function is guaranteed to terminate. In type theory, however, all constituent functions must be strongly terminating. Thus, for example, Hughes is able to derive separately two functions, the first of which constructs (lazily) a possibly infinite game-tree (and so may not terminate) and the second of which "prunes" the tree to a given depth. Type theory, on the other hand, only allows the derivation of terminating functions, and so below we have to derive a function which constructs a game-tree and which takes as a parameter the maximum depth of the tree to be constructed; this one function being equivalent to the composition of Hughes' two functions.

We assume throughout a base type, A, which we intend to be taken as a representation of positions in some game, but as we make few assumptions about this type, there is little loss of generality. The minimax algorithm searches a game-tree to select the best possible move which can be made from a given position of the game; we begin by deriving some functions which allow us to construct the game-tree.

The first small program we develop is a function, $roots \in Forest(A) \Rightarrow List(A)$, which, given a forest, returns the list of the roots of the trees in that forest. The function is defined to be:

$$\begin{array}{rll} roots & \equiv & \lambda([f]Felim(f \quad ,[x,y,hy]x \\ & & \qquad\qquad\qquad ,\mathbf{nil} \\ & & \qquad\qquad\qquad ,[x,y,hx,hy]hx :: hy \\ & & \qquad\qquad\quad) \\ & & \quad) \\ & \in & Forest(A) \Rightarrow List(A). \end{array}$$

The function is derived by using forest-elimination, which involves two elimination hypotheses, one pertaining to trees, the other to forests. For the hypothesis pertaining to trees, we choose A; for forests, $List(A)$. If we compare the derivation of *roots* (figure 15) to that of the function *mapforest* which we derived earlier, we see that they are quite similar, except that *roots* does make use of the hypothesis pertaining to trees.

Next we derive a function, $mkf \in (A \Rightarrow Forest(A)) \Rightarrow List(A) \Rightarrow Forest(A)$, which constructs a forest. The function takes as arguments a function, $g \in A \Rightarrow Forest(A)$, and a list of elements

```
0.0      |[  $f \in Forest(A)$
         ▷      { Forest-elimination, node-premise: construct an object of $A$ }
0.1.0         |[  $x \in A$; $y \in Forest(A)$; $hy \in List(A)$
0.1.1         ▷   $x \in A$
              ]|
                 { Forest-elimination, nilf-premise, nil-intro }
0.2           $\mathbf{nil} \in List(A)$
                 { Forest-elimination, :-premise: construct an object of $List(A)$ }
0.3.0         |[  $x \in Tree(A)$; $y \in Forest(A)$; $hx \in A$; $hy \in List(A)$
              ▷      { $::$-intro, 0.3.0 }
0.3.1             $hx :: hy \in List(A)$
              ]|
                 { Forest-elimination, 0.0, 0.1, 0.2, 0.3 }
0.4           $Felim(f,\ [x,y,hy]x,\ \mathbf{nil},\ [x,y,hx,hy]hx :: hy) \in List(A)$
         ]|
```

Figure 15: derivation D1

of type A and returns a forest of trees whose roots are the elements of the given list and whose subtrees are generated by the function g applied to the root. *mkf* is defined to be:

$$\lambda([g]\lambda([l]Listelim(l,\ \mathbf{nilf},\ [x,y,hy]node(x,g.x) : hy)))$$

and its derivation is given in figure 16.

Now we can show a useful property of *roots* and *mkf*, namely that for all $g \in A \Rightarrow Forest(A)$ and $l \in List(A)$:

$$roots.(mkf.g.l) = l \in List(A)$$

The proof, given in figure 17, uses the computation rules for *Felim*, but since the premises of those rules have effectively been given in the previous derivations, we omit them and use only the equations given in the introduction to trees and forests.

We turn now to the construction of the game-tree, which should be such that the root of a subtree denotes a position which can be reached from its ancestor in one move. That is, if we have a function, $moves \in A \Rightarrow List(A)$, which, given a position in a game, returns the list of positions which may be reached from that position in one move of the game, then we want to specify that for every subtree, $node(a,f)$, of the game-tree, $roots.f = moves.a \in List(A)$. It is a simple matter to specify that this relation holds between the root of a tree and its immediate subtrees, but how can we express that this relation holds recursively for all subtrees of a tree?

The solution lies with the non-canonical constants *Telim* and *Felim*: their mutual recursion allows the specification of just such properties. For example, suppose that Q is a relationship between objects $a \in A$ and objects $f \in Forest(A)$, i.e., $Q(a,f) \in U_1$ whenever $a \in A$ and $f \in Forest(A)$. Then we can construct a property P of trees such that

$$P(node(a,f)) \equiv Q(a,f).$$

```
0.0        |[  $g \in A \Rightarrow Forest(A)$
0.1.0      ▷   |[  $l \in List(A)$
               ▷      { induction on $l$, base case, nilf-intro }
0.1.1             $\mathbf{nilf} \in Forest(A)$
                      { induction step }
0.1.2.0           |[  $x \in A$; $y \in List(A)$; $hy \in Forest(A)$
                  ▷      { function application, 0.0, 0.1.2.0 }
0.1.2.1              $g.x \in Forest(A)$
                         { node-intro, 0.1.2.0, 0.1.2.1 }
0.1.2.2              $node(x, g.x) \in Tree(A)$
                         { :-intro, 0.1.2.2, 0.1.2.0 }
0.1.2.3              $node(x, g.x) : hy \in Forest(A)$
                  ]|
                      { list-elim, 0.1.0, 0.1.1, 0.1.2 }
0.1.3             $Listelim(l, \mathbf{nilf}, [x, y, hy]node(x, g.x) : hy) \in Forest(A)$
               ]|
           ]|
```

Figure 16: derivation D2

We can specify this for an arbitrary $t \in Tree(A)$ by defining

$$\begin{array}{rcll} P(t) & \equiv & Telim(t & , [x, y, hy]Q(x, y) \\ & & & , \top \\ & & & , [x, y, hx, hy]\top \\ & &) & \\ & \in & U_1 . & \end{array}$$

Since, for the moment, we are not interested in specifying properties of forests, we let $\top$ in the second and third clauses of the *Telim* expression denote some always-inhabited type (hence, "true"). However, it is a simple matter to extend this definition to a property, FP, of forests, which expresses that for $f \in Forest(A)$, each tree in the forest f enjoys property P:

$$\begin{array}{rcll} FP(f) & \equiv & Felim(f & , [x, y, hy]Q(x, y) \\ & & & , \top \\ & & & , [x, y, hx, hy]hx \wedge hy \\ & &) & \\ & \in & U_1 . & \end{array}$$

Here, the $\top$ in the second clause expresses that FP is vacuously true of the empty forest.

This can be generalised one step further to allow the relation to depend upon the recursion variable hy (to allow, for example, that the relation hold recursively for each subtree of a given tree). Thus, we posit a new relation, R, such that:

```
|[  $x \in A$; $y \in Forest(A)$; $hy \in U_1$
▷   $R(x, y, hy) \in U_1$
]|
              { $roots \equiv \lambda([f]Felim(f, [x,y,hy]x, \mathbf{nil}, [x,y,hx,hy]hx :: hy))$ }
              { $mkf \equiv \lambda([g]\lambda([l]Listelim(l, \mathbf{nilf}, [x,y,hy]node(x,g.x) : hy)))$ }
0.0        |[   $g \in A \Rightarrow Forest(A)$
0.1.0      ▷    |[   $l \in List(A)$
                ▷      { induction on $l$, base case, definition of $mkf$ }
0.1.1                $mkf.g.\mathbf{nil} = \mathbf{nilf} \in Forest(A)$
0.1.2                    $roots.(mkf.g.\mathbf{nil})$
                     $=_{List(A)}$          { congruence of function application, 0.1.1 }
                         $roots.\mathbf{nilf}$
                     $=_{List(A)}$          { definition of $roots$ }
                         $\mathbf{nil}$
                       { induction step }
0.1.3.0              |[   $x \in A$; $y \in List(A)$; $hy \in (roots.(mkf.g.y) =_{List(A)} y)$
                     ▷      { definition of $mkf$ }
0.1.3.1                   $mkf.g.(x :: y) = node(x,g.x) : (mkf.g.y) \in Forest(A)$
0.1.3.2                       $roots.(mkf.g.(x :: y))$
                          $=_{List(A)}$          { congruence, 0.1.3.1 }
                              $roots.(node(x,g.x) : (mkf.g.y))$
                          $=_{List(A)}$          { definition of $roots$ }
                              $x :: (roots.(mkf.g.y))$
                          $=_{List(A)}$          { hypothesis, 0.1.3.0, $::$-congruence }
                              $x :: y$
                     ]|
                       { list-elim, 0.1.0, 0.1.2, 0.1.3, suppressing proof-object }
0.1.4                $roots.(mkf.g.l) = l \in List(A)$
                ]|
           ]|
```

Figure 17: derivation D3

We formulate a new property, P, which expresses that, for $t \in \mathit{Tree}(A)$, R holds for the components of t, and possibly for each subtree of t:

$$\begin{array}{rll} P(t) & \equiv & \mathit{Telim}(t \quad , [x, y, hy]R(x, y, hy) \\ & & \qquad\qquad , \top \\ & & \qquad\qquad , [x, y, hx, hy]hx \wedge hy \\ & & \qquad) \\ & \in & U_1. \end{array}$$

And, similarly, the corresponding property defined on forests:

$$\begin{array}{rll} FP(f) & \equiv & \mathit{Felim}(f \quad , [x, y, hy]R(x, y, hy) \\ & & \qquad\qquad , \top \\ & & \qquad\qquad , [x, y, hx, hy]hx \wedge hy \\ & & \qquad) \\ & \in & U_1. \end{array}$$

Now, using P and FP as above, we can show that mkf preserves certain properties in that

$$\mathit{mkf} \in \forall(A,\ [a]\mathit{Set}(\mathit{Forest}(A), [f]P(\mathit{node}(a, f)))) \Rightarrow \mathit{List}(A) \Rightarrow \mathit{Set}(\mathit{Forest}(A), FP).$$

That is, given a function $g \in A \Rightarrow \mathit{Forest}(A)$ such that for all $a \in A$, $P(\mathit{node}(a, g.a))$ holds, and given a list $l \in \mathit{List}(A)$, $FP(\mathit{mkf}.g.l)$ also holds. The proof is given in figure 18.

Now we assume that we have a function $\mathit{moves} \in A \Rightarrow \mathit{List}(A)$ which, given a position, returns the list of positions which may be reached from that position in one move. Our game-tree, then, will be such that for every $\mathit{node}(a, f)$ which occurs in the tree, $\mathit{roots}.f = \mathit{moves}.a \in \mathit{List}(A)$, and the same will hold for each subtree in the forest f. However, since we cannot construct an infinite game-tree, we must weaken this requirement to $(f = \mathbf{nilf}) \vee (\mathit{roots}.f = \mathit{moves}.a)$. We specify this as the property M, where, for $t \in \mathit{Tree}(A)$,

$$\begin{array}{rll} M(t) & \equiv & \mathit{Telim}(t \quad , [x, y, hy](y = \mathbf{nilf}) \vee (\mathit{roots}.y = \mathit{moves}.x \wedge hy) \\ & & \qquad\qquad , \top \\ & & \qquad\qquad , [x, y, hx, hy]hx \wedge hy \\ & & \qquad) \\ & \in & U_1. \end{array}$$

The occurrence of hy in the first clause expresses that the ancestor-descendant relation holds recursively for all subtrees. There is also the corresponding property, FM, which expresses that, for $f \in \mathit{Forest}(A)$, each tree in f enjoys property M:

$$\begin{array}{rll} FM(f) & \equiv & \mathit{Felim}(f \quad , [x, y, hy](y = \mathbf{nilf}) \vee (\mathit{roots}.y = \mathit{moves}.x \wedge hy) \\ & & \qquad\qquad , \top \\ & & \qquad\qquad , [x, y, hx, hy]hx \wedge hy \\ & & \qquad) \\ & \in & U_1. \end{array}$$

```
                  ${FP(f) \equiv Felim(f,\ R,\ \top,\ [x,y,hx,hy]hx \wedge hy)}$
0.0               |[  $g \in \forall(A,\ [a]Set(Forest(A), [f]P(node(a,f))))$
0.1.0             ▷   |[  $l \in List(A)$
                      ▷      { induction on $l$, base case, nilf-intro }
0.1.1                    $\mathbf{nilf} \in Forest(A)$
                             { definition of $FP$ }
0.1.2                    $FP(\mathbf{nilf}) = \top$
                             { $\top$ always inhabited; type-equality, subset-intro, 0.1.1 }
0.1.3                    $\mathbf{nilf} \in Set(Forest(A), FP)$
                             { induction step }
0.1.4.0                  |[  $x \in A;\ y \in List(A);\ hy \in Set(Forest(A), FP)$
                         ▷       { function application, 0.0, 0.1.4.0 }
0.1.4.1                      $g.x \in Set(Forest(A), [f]P(node(x,f)))$
                                 { assumptions for subset-elim on $hy$ and $g.x$ }
0.1.4.2.0                    |[  $hy \in Forest(A);\ q \in FP(hy);\ gx \in Forest(A);\ r \in P(node(x,gx))$
                             ▷       { $node$-intro, :-intro, 0.1.4.0, 0.1.4.2.0 }
0.1.4.2.1                        $node(x,gx) : hy \in Forest(A)$
                                     { definition of $FP$ }
0.1.4.2.2                        $FP(node(x,gx) : hy) = P(node(x,gx)) \wedge FP(hy)$
                                     { pair-intro, 0.1.4.2.0, type-equality, 0.1.4.2.2 }
0.1.4.2.3                        $\langle r,q \rangle \in FP(node(x,gx) : hy)$
                                     { subset-intro, 0.1.4.2.1, 0.1.4.2.3 }
0.1.4.2.4                        $node(x,gx) : hy \in Set(Forest(A), FP)$
                             ]|
                                 { subset-elim, twice, 0.1.4.0, 0.1.4.1, 0.1.4.2 }
0.1.4.3                      $node(x, g.x) : hy \in Set(Forest(A), FP)$
                         ]|
                             { list-elim, 0.1.0, 0.1.3, 0.1.4 }
0.1.5                    $Listelim(l,\ \mathbf{nilf},\ [x,y,hy]node(x,g.x) : hy) \in Set(Forest(A), FP)$
                      ]|
                  ]|
```

Figure 18: derivation D4

From the symmetry of these expressions, it is easy to verify that

$$M(node(a, f)) = (f = \mathbf{nilf}) \vee (roots.f = moves.a \wedge FM(f))$$

and

$$FM(t : f) = M(t) \wedge FM(f).$$

We now give the function $gen \in \mathbb{N} \Rightarrow A \Rightarrow Forest(A)$, which we use to construct the game-tree. The function is defined to be:

$$\lambda([n]\mathbb{N}\text{-}elim(n,\ \lambda([a]\mathbf{nilf}),\ [x, hx]\lambda([a]mkf.hx.(moves.a))))$$

and is such that, for $n \in \mathbb{N}$ and $a \in A$, $gen.n.a$ is a forest of trees of level at most n such that $M(node(a, gen.n.a))$ holds. To ensure this latter property, figure 19 shows that $gen \in \mathbb{N} \Rightarrow \forall(A,\ [a]Set(Forest(A), [f]M(node(a, f))))$. The proof uses derivation D4, with P instantiated to M and FP instantiated to FM. We also use the property proven in D3, that for all $g \in A \Rightarrow Forest(A)$ and all $l \in List(A)$, $roots.(mkf.g.l) = l \in List(A)$.

Now that we can construct our game-tree, we turn to our last function, the minimax algorithm. The minimax algorithm is intended for a two-player game and assumes that there is some method of evaluating how good a given position is for a fixed player. Working on the assumption that this player will always try to move to a maximally good position, and that the other player will always move to a minimally good (for his opponent) position, minimax searches a game-tree by alternately selecting the maxima and minima of the levels of the game-tree. The algorithm is usually written as consisting of two mutually recursive functions, *maximise* and *minimise*:

$$\begin{aligned} maximise.(node(a, \mathbf{nilf})) &= a &(19)\\ maximise.(node(a, f)) &= max.(mapforest.minimise.f) &(20)\\ minimise.(node(a, \mathbf{nilf})) &= a &(21)\\ minimise.(node(a, f)) &= min.(mapforest.maximise.f) &(22) \end{aligned}$$

We assume given the functions $min \in List(A) \Rightarrow A$ and $max \in List(A) \Rightarrow A$, which select, respectively, a minimally good and a maximally good position from a list of positions. First, we construct a function, $alt \in Tree(A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow A$, which, given a tree and two functions, applies those functions alternately at each level of the tree. We define alt to be:

$$\begin{aligned} alt \equiv\ & \lambda([t]Telim(t\ \ , [x, y, hy]\lambda([p, q]\mathbf{if}\ y = \mathbf{nilf}\ \mathbf{then}\ x\ \mathbf{else}\ p.(hy.q.p))\\ &\qquad\qquad\qquad , \lambda([p, q]\mathbf{nil})\\ &\qquad\qquad\qquad , [x, y, hx, hy]\lambda([p, q](hx.p.q) :: (hy.p.q))\\ &\qquad\qquad\quad)\\ &\quad)\\ \in\ & Tree(A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow A. \end{aligned}$$

The swapping of the functions p and q in the first clause effects the alternate applications. The function is derived by Tree-elimination in the obvious way: the elimination hypothesis pertaining to trees is $(List(A) \Rightarrow A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow A$, and the elimination hypothesis pertaining

```
{$M(t) \equiv Telim(t,\ [x,y,hy](y = \mathbf{nilf}) \vee (roots.y = moves.x \wedge hy),\ \top,\ [x,y,hx,hy]hx \wedge hy)$}
{$FM(f) \equiv Felim(f,\ [x,y,hy](y = \mathbf{nilf}) \vee (roots.y = moves.x \wedge hy),\ \top,\ [x,y,hx,hy]hx \wedge hy)$}
{$gen \equiv \lambda([n]\mathbb{N}\text{-}elim(n,\ \lambda([a]\mathbf{nilf}),\ [x,hx]\lambda([a]mkf.hx.(moves.a))))$}
0.0        |[  $n \in \mathbb{N}$
           ▷       { induction on $n$, base case }
0.1.0          |[  $a \in A$
               ▷       { nilf-intro }
0.1.1              $\mathbf{nilf} \in Forest(A)$
                       { definition of $M$ }
0.1.2              $M(node(a,\mathbf{nilf})) = (\mathbf{nilf} = \mathbf{nilf}) \vee (roots.\mathbf{nilf} = moves.a \wedge FM(\mathbf{nilf}))$
                       { $\mathbf{nilf} = \mathbf{nilf}$, $\vee$-intro, subset-intro, 0.1.1, 0.1.2 }
0.1.3              $\mathbf{nilf} \in Set(Forest(A), [f]M(node(a,f)))$
               ]|
                   { $\lambda$-intro, 0.1 }
0.2            $\lambda([a]\mathbf{nilf}) \in \forall(A,\ [a]Set(Forest(A), [f]M(node(a,f))))$
                   { induction step }
0.3.0          |[  $x \in \mathbb{N};\ hx \in \forall(A,\ [a]Set(Forest(A), [f]M(node(a,f))))$
0.3.1.0        ▷   |[  $a \in A$
0.3.1.1            ▷   $moves.a \in List(A)$
                           { D4, definitions of $M$ and $FM$ }
0.3.1.2                $mkf.hx.(moves.a) \in Set(Forest(A), FM)$
                           { D3 }
0.3.1.3                $roots.(mkf.hx.(moves.a)) = moves.a \in List(A)$
                           { definition of $M$, subset-intro, 0.3.1.2, 0.3.1.3 }
0.3.1.4                $mkf.hx.(moves.a) \in Set(Forest(A), [f]M(node(a,f)))$
                   ]|
                       { $\lambda$-intro, 0.3.1 }
0.3.2              $\lambda([a]mkf.hx.(moves.a)) \in \forall(A,\ [a]Set(Forest(A), [f]M(node(a,f))))$
               ]|
                   { $\mathbb{N}$-elim, 0.0, 0.2, 0.3, definition of $gen$ }
0.4            $gen.n \in \forall(A,\ [a]Set(Forest(A), [f]M(node(a,f))))$
           ]|
```

Figure 19: derivation D5

to forests is $(List(A) \Rightarrow A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow List(A)$. Similarly, we can derive a function, $mapalt \in (List(A) \Rightarrow A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow List(A)$, which corresponds to alt, but is defined on forests, and effectively applies alt to each tree in a given forest:

$$\begin{aligned}
mapalt \equiv\ & \lambda([f]Felim(f \quad , [x, y, hy]\lambda([p, q]\mathbf{if}\ y = \mathbf{nilf}\ \mathbf{then}\ x\ \mathbf{else}\ p.(hy.q.p)) \\
& \qquad\qquad\qquad , \lambda([p, q]\mathbf{nil}) \\
& \qquad\qquad\qquad , [x, y, hx, hy]\lambda([p, q](hx.p.q) :: (hy.p.q)) \\
& \qquad\qquad\quad) \\
&) \\
\in\ & Forest(A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow (List(A) \Rightarrow A) \Rightarrow List(A)
\end{aligned}$$

The last step is to define $maximise \in Tree(A) \Rightarrow A$ to be $\lambda([t]alt.t.max.min)$ and define $minimise \in Tree(A) \Rightarrow A$ to be $\lambda([t]alt.t.min.max)$ and then prove that these functions satisfy equations (19)-(22) above. By using the computation rules, it is easily shown that:

$$mapalt.f.min.max = mapforest.minimise.f$$

$$mapalt.f.max.min = mapforest.maximise.f$$

from which, again by using the computation rules:

$$\begin{aligned}
maximise.(node(a, f)) &= \mathbf{if}\ f = \mathbf{nilf}\ \mathbf{then}\ a\ \mathbf{else}\ max.(mapalt.f.min.max) \\
&= \mathbf{if}\ f = \mathbf{nilf}\ \mathbf{then}\ a\ \mathbf{else}\ max.(mapforest.minimise.f)
\end{aligned}$$

$$\begin{aligned}
minimise.(node(a, f)) &= \mathbf{if}\ f = \mathbf{nilf}\ \mathbf{then}\ a\ \mathbf{else}\ min.(mapalt.f.max.min) \\
&= \mathbf{if}\ f = \mathbf{nilf}\ \mathbf{then}\ a\ \mathbf{else}\ min.(mapforest.maximise.f)
\end{aligned}$$

Finally, given $a \in A$ and $n \in \mathbb{N}$, the program

$$maximise.(node(a, gen.n.a))$$

finds the best move that can be made from a in n moves.

7 Conclusion

The world of programming languages seems to be split into two quite distinct and mutually 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 [45].

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.

Many would argue 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 — 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.

There are, moreover, many properties of a program that can be discovered neither at run-time nor at compile-time because of either theoretical or practical impossibility. We need only mention one — termination. The constructive theory of types that we have described here was originally developed unfettered by implementation ideologies or, indeed, by any concern for practical programming issues. Yet its introduction of the notion of dependent types was both a vital and an inevitable step; as a consequence, the notion of type is sufficiently enriched as to be equated with specification. And as a consequence, the type of a program is not a decidable property. The responsibility for ensuring that a program is well-typed devolves thus upon the professional programmer — Martin-Löf's theory of types is in our view an excellent exemplar of a formalism for *development-time* type checking.

On the other hand there remain drawbacks to the practical application of the theory that it would be dishonest of us not to mention. Two in particular concern (a) the mismatch between programs and proofs and (b) the introduction of well-founded recursion.

With regard to the former, we have already discussed the use of the subset type as a mechanism for eliminating computationally irrelevant information from proof objects. It is, however, a mechanism that in our view does not go far enough. Rather, it is the case that in many instances (for example, equalities and negations) the identification of propositions with types is far-fetched and awkward. In cases where the proof of a proposition contributes no computationally relevant information there is also no reason why classical reasoning should not be used. Current thinking is therefore towards a separation of propositions from types.

Related to the separation of propositions from types is the distinction between Gentzen-style proof derivations (the sort we have used here) and equational style reasoning. An argument that is aired nowadays is that Gentzen-style reasoning is better suited for machine implementations (witness the NuPRL system) than for human reasoning (in apparent contradiction to Gentzen's own claim that his was a "natural" system of logical deduction). Indeed it is often the case that equational-style derivations within the classical calculus are substantially more elegant than Gentzen-style derivations of the same propositions. The advantages of Gentzen-style derivations become apparent, however, in those cases where the structure of the proof directly reflects the structure of the program that is created. A proper separation of propositions from types will therefore separate those parts of program design that are directly reflected in the program structure from those parts that leave no visible trace in the program. Gentzen-style reasoning will continue to be effective in the former case but we are often more convinced by equational reasoning in the latter case.

The second drawback of Martin-Löf's theory concerns the strict requirement of totality for all functions defined within the theory. The inability to define non-terminating computations is not something that we regard as a major handicap to the practising programmer, although the introduction of partial functions is regarded by the NuPRL group as an important innovation

[15,16]. The drawback that we regard as more urgent is that there is no clean mechanism within the theory whereby (total) functions may be defined by well-founded recursion such as is used in, say, the development of quick-sort. There have been several attempts to introduce such a mechanism whilst maintaining the philosophy and elegance of Martin-Löf's formalism [54,50,58] but in our view the problem has still not been adequately resolved.

Computer programming has invigorated the study of formal systems not just because a proper formalisation is a prerequisite of any implementation but because good formalisms can be very effective in understanding and assisting the process of developing programs. Constructive type theory is a formalism that helps us to understand and to exploit the relationship between data and program structure; it is this aspect of the theory that we have chosen to emphasis here. A complaint that may be made is that we have been somewhat cavalier in our discussion of semantical and other foundational issues. For discussion of such aspects of the theory we refer the reader to Martin-Löf's own accounts [41,42,40] and to the work of Allen [1] and of Nordström et al. [53]. With reference to the introduction of new type structures into the theory we would particularly draw attention to the work of Mendler [44] and Constable and Mendler [14] where some of the limitations and pitfalls of the techniques that we have exemplified are amply discussed.

There is a number of other topics that we have not discussed, not least of which is implementations of the theory such as the NuPRL system [13,16], and the related implementation of "Constructions" [17]. Also not mentioned is the development and implementation of "logical frameworks" [28,22], a topic which can be said to owe its very existence to constructive type theory. Finally, the relationship between the work presented here and categorical accounts of type structures is one that we have just hinted at. We have not discussed it in depth because we ourselves are not capable of doing so at this point in time. Nevertheless it is a topic that we believe will receive particular attention in the future.

Acknowledgements

We would like to take this opportunity to express our gratitude to those who have made this paper possible. First and foremost, our thanks go to Per Martin-Löf, the author of the theory on which the paper is based, and to the members of the Göteborg Programming Methodology Group, Bengt Nordström, Kent Petersson and Jan Smith, for their pioneering efforts in bringing the theory to the attention of computing scientists. We have received valuable criticism and support from Stuart Anderson and from members of the Groningen Tuesday Afternoon Club. Thanks also go to Hilary Backhouse for her LaTeXpertise. G.R. Malcolm is supported by a grant from the Science and Engineering Research Council of Gt. Britain.

References

[1] S. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, September 1987.

[2] R.C. Backhouse. Notes on Martin-Löf's theory of types, parts 1 and 2. In *FACS FACTS*, British Computer Society, 1986.

[3] R.C. Backhouse. *On the Meaning and Construction of the Rules in Martin-Löf's Theory of Types.* Computing Science Notes CS 8606, Department of Mathematics and Computing Science, University of Groningen, 1986.

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

[5] R.S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.

[6] R.S. Boyer and J S. Moore. *MJRTY — A Fast Majority Vote Algorithm.* Technical Report ICSCA-CMP-32, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

[7] N.G. de Bruijn. A survey of the project automath. In J.P. Seldin and J.R. Hindley, editors, *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, Academic Press, 1980.

[8] P. Chisholm. Derivation of a parsing algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, 8:1–42, 1987.

[9] P. Chisholm. *Investigations into Martin-Löf Type Theory as a Programming Logic.* PhD thesis, Department of Computer Science, Heriot-Watt University, Edinburgh, July 1988.

[10] A. Church. *The Calculi of Lambda-Conversion.* Volume 6 of *Annals of Mathematical Studies*, Princeton University Press, Princeton, 1951.

[11] R. Cleaveland and P. Panangaden. *Type Theory and Concurrency.* Technical Report TR 85-714, Department of Computer Science, Cornell University, December 1985.

[12] R.L. Constable. Constructive mathematics as a programming logic 1: some principles of theory. *Annals of Discrete Mathematics*, 24:21–38, 1985.

[13] R.L. Constable, T.B. Knoblock, and J.L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1(3):285–326, 1985.

[14] R.L. Constable and N.P. Mendler. Recursive definitions in type theory. In *Proceedings of Logics of Programs Conference, LNCS 193*, pages 61–78, Springer-Verlag, 1985.

[15] R.L. Constable and S.F. Smith. Partial objects in constructive type theory. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 183–193, Computer Society Press of the IEEE, 1987.

[16] R.L. Constable, et al. *Implementing Mathematics in the Nuprl Proof Development System.* Prentice-Hall, 1986.

[17] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In *Proceedings of EUROCAL 85*, Linz, Austria, April 1985.

[18] H.B. Curry and R. Feys. *Combinatory Logic.* Volume 1, North-Holland, 1958.

[19] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming.* Academic Press, 1972.

[20] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[21] E.W. Dijkstra and W.H.J. Feijen. *Een Methode van Programmeren*. Academic Service, Den Haag, 1984. Now available as A Method of Programming, Addison-Wesley, Reading, Mass., 1988.

[22] P. Dybjer. *Inductively Defined Sets in Martin-Löf's Set Theory*. Technical Report, Department of Computer Science, University of Göteborg/Chalmers, April 1987.

[23] R. Dyckhoff. *Category Theory as an Extension of Martin-Löf Type Theory*. Technical Report CS/85/3, Department of Computational Science, University of St. Andrews, 1985.

[24] G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–213, North-Holland, Amsterdam, 1969.

[25] V. Glivenko. Sur quelques points de la logique de m. brouwer. *Bulletins de la classe des sciences*, 15:183–188, 1929.

[26] M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF*. Springer-Varlag, 1979.

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

[28] R. Harper, F.A. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings of the Second Annual Conference on Logic in Computer Science*, Cornell, Dec 1987.

[29] E.R.C. Hehner. *The Logic of Programming*. Prentice-Hall, 1984.

[30] C.A.R. Hoare. Notes on data structuring. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, Academic Press, 1972.

[31] W.A. Howard. The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490, Academic Press, 1980.

[32] J. Hughes. *Why Functional Programming Matters*. Technical Report, Department of Computer Science, University of Göteborg/Chalmers, 1984.

[33] M.A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[34] K. Jensen and N. Wirth. *PASCAL: user manual and report*. Springer-Verlag, 1975.

[35] S.C. Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.

[36] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:325–350, 1977.

[37] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic*. Volume 7 of *Studies in Advanced Mathematics*, Cambridge University Press, 1986.

[38] G.R. Malcolm and P. Chisholm. Polymorphism in Martin-Löf's type theory. Department of Mathematics and Computing Science, University of Groningen, 1988.

[39] P. Martin-Löf. Constructive mathematics and computer programming. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Computer Programming*, pages 167–184, Prentice-Hall, 1984.

[40] P. Martin-Löf. Constructive mathematics and computer programming. In L.J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *Logic, Methodology and Philosophy of Science, IV*, pages 153–175, North-Holland, 1982.

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

[42] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padova.

[43] L. Meertens. Algorithmics - towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334, North-Holland, 1986.

[44] N.P. Mendler. *Inductive Definitions in Type Theory*. PhD thesis, Cornell University, September 1987.

[45] R. Milner. The standard ml core language. *Polymorphism*, II(2), October 1985.

[46] R. Milner. A theory of type polymorphism in programming. *J. Comp. Syst. Scs.*, 17:348–375, 1977.

[47] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[48] J. Mitchell and G. Plotkin. Abstract types have existential types. In *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 37–51, 1985.

[49] B. Nordström. Multilevel functions in type theory. In N. Jones, editor, *Programs as Data Objects*, Springer-Verlag, LNCS 217, 1985.

[50] B. Nordström. *Terminating General Recursion*. Technical Report, Programming Methodology Group, University of Göteborg/Chalmers, September 1987.

[51] B. Nordström and K. Petersson. *The Semantics of Module Specifications in Martin-Löf's Type Theory*. Technical Report 36, Programming Methodology Group, University of Göteborg/Chalmers, October 1985.

[52] B. Nordström and K. Petersson. Types and specifications. In R.E. Mason, editor, *IFIP '83*, pages 915–920, Elsevier Science Publishers, 1983.

[53] B. Nordström, K. Petersson, and J. Smith. *An Introduction to Martin-Löf's Theory of Types*. Technical Report, Programming Methodology Group, University of Göteborg/Chalmers, 1986.

[54] L.C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[55] K. Petersson and D. Synek. *A Set Constructor for Trees in Intuitionistic Type Theory*. Technical Report, Department of Computer Science, University of Göteborg/Chalmers, August 1987.

[56] D. Prawitz. Proofs and the meaning and completeness of the logical constants. In J. Hintikka, I. Niiniluoto, and E. Saarinen, editors, *Essays on Mathematical and Philosophical Logic*, pages 25–40, Reidel, 1979.

[57] J.C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.

[58] E. Saaman and G.R. Malcolm. *Well-founded Recursion in Type Theory*. Computing Science Notes CS 8701, Department of Mathematics and Computer Science, University of Groningen, 1987.

[59] D. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[60] P. Schröder-Heister. A natural extension of natural deduction. *The Journal of Symbolic Logic*, 49(4), Dec 1984.

[61] J. Smith. *On a Nonconstructive Type Theory and Program Derivation.* Technical Report, Department of Computer Science, University of Göteborg/Chalmers, November 1985.

[62] J. Stoy. *Denotational Semantics.* The MIT Press, Cambridge, Mass, 1977.