A While-rule in Martin-Löf’s Theory of Types

R. BACKHOUSE* AND A. KHAMISS
Department of Computer Science, University of Essex, Colchester CO4 3SQ

The use of invariant properties and bound functions is a sound and well-documented methodology for the design of loop structures. We show how to formulate the methodology as a proposition in the intuitionistic theory of types developed by Per Martin-Löf. By proving the validity of this proposition we effectively obtain a method of writing while-statements in Type Theory. Two simple and well-known examples are given to illustrate the method.

Received July 1985

INTRODUCTION

Of those developments in theoretical computer science in the last decade or so that have had a significant impact on the very practical process of program construction, two of the most important are the introduction of type definitions into programming languages and the use of the principles of program verification to structure the program design process. Both developments have the same objective, to improve the reliability of programs by reducing the possibilities for error. In the words of C. A. R. Hoare: ‘high-level languages should be designed to extend the range of programming errors which logically cannot be made’.

The intuitionistic theory of types developed by Per Martin-Löf has a tremendous appeal to the present authors because it appears to combine within one elegant framework the conceptual advantages of types and program verification. In Martin-Löf’s theory specifications are identified with types and programs are identified with proofs. A consequence of this is that the formulation of inductive hypotheses in proofs of theorems is readily seen to be identical to the invention of invariants in the formation of loops in programs, and that strengthening an inductive hypothesis is the same process as adding extra information to a loop invariant.

Our own appreciation of the full ramifications of Martin-Löf’s theory is still incomplete, but it is already becoming clear to us that, as a vehicle for communication, the theory is at the level of an (albeit functional) assembly language. Although providing very powerful type-definition facilities, many high-level programming language features – even addition and multiplication – are absent from the system and quite cumbersome to define. Like the early machine-code and assembly language programmers we are therefore faced with a major task of bootstrapping the theory to a level at which it can be used in a practical programming environment.

The saving grace of Martin-Löf’s theory is that it is possible to express program design principles within the theory itself as derived rules. Smith has made a start on this by expressing course-of-values recursion on lists within the theory, and Paulson has gone further by examining the notion of well-founded orderings in the theory. We tackle a similar problem by expressing a while-rule (the use of invariants and bound functions in the development of looping structures) in type theory.

The importance and practicality of this rule are not in doubt; it is extremely well documented in a number of textbooks. Our contribution is to formulate the rule as a proposition and to prove the validity of that proposition in Martin-Löf’s theory. Equivalently, making the identification of propositions with specifications and proofs with programs, we specify the function of a while-statement and construct an implementation of that specification using the primitive looping mechanisms of type theory.

1. TYPES AND PROPOSITIONS, PROGRAMS AND PROOFS

In this section we outline the main features of Martin-Löf’s theory. Our own account is self-contained, but readers wishing further introductory material may consult Refs. 3, 5, 12 and 13.

In order to make the material readily accessible to computing scientists we have deliberately used our own notation, a notation that has been designed to be as close as possible to ML-notation. Our notation may be criticized for being verbose, but readability has been our primary concern; appendix A shows the correspondence between it and Martin-Löf’s own notation. Unfortunately, there is no standard notation, Martin-Löf himself having made significant changes to his notation quite recently. The two major groups active in the development of type theory, at Chalmers University of Technology and at Cornell University, also use different notations to each other and to Martin-Löf.

1.1 Type constructors

The primitive types in type theory include the set of natural numbers, denoted by N, and the enumeration types, for example (nil) and (red, yellow, blue). Such types are objects of a universe of types U. We write A: U meaning A is an object of the universe, or A is a type.

One of the most remarkable aspects of type theory is the identity it exhibits between types and propositions. Complex types are built from the primitive types using the operators ⇒ (implies), ∧ (and), ∨ (or), ∃ (for all) and ∃ (there exists) of the predicate calculus but the propositions so obtained are inseparable from the more familiar type constructors like functions, disjoint unions and Cartesian products. Fig. 1 summarises the correspondence.

The proposition A ⇒ B is identified with the type

* To whom correspondence should be addressed.
A → B of (total) functions from the type A to the type B. For example, the valid proposition A ⇒ A is the type of all functions from A to itself. The proposition is proved by exhibiting such a function, for example the identity function on A. In Fig. 1 we write

A: U ⊢ I_A: A ⇒ A

This can be read in two ways: (a) Assuming A is a type ("A: U ⊢") the identity function ("I_A") is a proof of ("⇒") the proposition A implies A; or (b) assuming A is a type the identity function is a total function from A into A.

The proposition A ∧ B is identified with the Cartesian product A × B of the types A and B. Thus objects of A ∧ B are pairs the first and second components of which are of types A and B, respectively. Thus the proposition A ∧ B ⇒ A is the type of all total functions from the Cartesian product of A and B into A. A proof of its validity is the function fst_A,B that projects a pair (a, b), on to its first component, a.

The proposition A ∨ B is identified with the disjoint union of the types A and B. Thus the function inl_A,B that injects an object of type A into the disjoint union A + B is a proof of A ⇒ A ∨ B.

The proposition (∀x:A) B(x) is identified with a function type in which the type of the result of the function is dependent on the argument a in A. This is indicated by the notation (a:A) → B(a) in Fig. 1. An example is the proposition (∀x:A) U(x) ⇒ A(x), which is a function that takes a type A as an argument and constructs a function from A into itself. The only object of this type is the polymorphic identity function, denoted by I. Similar examples would be the polymorphic projection and injection functions fst and inl which are objects of type (∀x:A)(∀y:B)(A ∧ B ⇒ A) and (∀x:A)(∀y:B)(A ⇒ A ∨ B), respectively.

The proposition (∃x:A) B(x) is identified with the Cartesian product (a:A) × B(a) in which the type of the second component depends on the object in the first component, that is with \{(x,y) ∈ (A × B) | (x:A) ∧ (y:B(x))\}. An example of such a proposition is (∃x:A) U, which may be read as ‘there is a non-empty type’ or ‘there is a valid proposition’. There are many objects of this type including (N, 0), (red, yellow, blue), (yellow) and ((∀x:A)(A ⇒ A), 1).

### 1.2 Proof rules

Martin-Löf's theory is formulated as a set of proof rules in the style of Gentzen's system of natural deduction. Four of the rules, in their simplest form, are shown in Fig. 2(a). The rules are divided into two sets, the introduction rules—which serve to introduce the propositional connectives into a formula—and the elimination rules—which serve to eliminate the connectives.

The introduction rule in Fig. 2(a) states that the conjunction A ∧ B may be inferred from individual proofs of A and of B. The elimination rule states that if the conjunction A ∧ B has been proved, and C can be proved assuming individually that A and B are provable, then C is true in general. The ⇒-introduction rule states that A ⇒ B follows if B can be proved assuming the truth of A. Finally the ⇒-elimination rule states that B follows from proofs of A ⇒ B and A. Note that assumptions are indicated by brackets (\{\}). Such assumptions are said to be discharged by the rule. The inference of a rule is the formula below the line, the premises are the formulae above the line.

Complex propositions may be proved by combining the inference rules into a proof tree. Fig. 3(a) exhibits a proof tree for the proposition \([A ∧ B ⇒ C \Rightarrow [A ⇒ (B ⇒ C)]\). In this proof we have begun by making three assumptions, (A ∧ B ⇒ C, A, and B, indicated by the numerals 1, 2 and 3. These assumptions are later discharged by three instances of the ⇒-introduction rule, indicated by attaching the appropriate numeral to the explanation of the step. Fig. 3(b) exhibits the same tree but in a linear style that we find more convenient to use. Note the use of indentation for the introduction and outdentation for the discharge of assumptions.

Martin-Löf's contribution has been to add proof objects to the rules, such objects being interpreted logically as summaries of the proof or computationally as programs achieving the stated proposition. Fig. 2(b) introduces proof objects into the rules of Fig. 2(a). (The
rules in Fig. 2(b) have been simplified slightly so as not to confuse the reader with too much information at this stage in the presentation.) The (new) ∧-introduction rule can be read in a logical sense as ‘if a is a proof of A and b is a proof of then the pair (a, b) is a proof of A ∧ B’. In a computational sense it takes on the meaning ‘if a is of type A and b is of type B then the pair (a, b) is of type A ∧ B’.

The ∧-elimination rule can be read in a logical sense as ‘if p is a proof of A ∧ B and assuming that a is a proof of A and b is a proof of B it is possible to construct a proof c of C, then letting a and b be the components of p in the expression c is a proof of C.’ Computationally it reads ‘if p is of type A ∧ B, and assuming that a is of type A and that b is of type B, c is of type C then let (a, b) ⇒ p in c is of type C’.

Note the expression c(a, b) in the premise of this rule indicates that c may depend on one or both of a and b. Note also that the effect of discharging the assumptions a: A and b: B is to bind the occurrences of a and b in the expression let (a, b) ⇒ p in c.

The ⇒-introduction rule can be read in a logical sense as ‘if ‘if p is a proof of B assuming that a is a proof of A then a. b is a proof of A ⇒ B’. In a computational sense it has the meaning ‘if b is of type B whenever a is of type A then a. b is a function mapping objects of type A into objects of type B’.

Finally, the ⇒-elimination states that ‘if f is a proof of A ⇒ B and a is a proof of A then (f a) is a proof of B’ or as ‘if f is a function mapping objects of type A into objects of type B and a has type A then f applied to a is of type B’.

Note that function application is denoted by juxtaposition as is conventional in the lambda calculus.10 Two other conventions of the lambda calculus that we adopt are that function application associates to the left, so that f a b means (f a) b, and that λ. f b means λ. f (b).

Introducing proof objects into Fig. 3 results in Fig. 4. In a computational sense what we have proved is that any function of a pair of arguments can be ‘curried’,” i.e. that if a b means (f a) b, and that λ. f g h means (λ. f (g h)).

Figure 3. Proof of \((A \land B) \Rightarrow C \Rightarrow (A \Rightarrow (B \Rightarrow C))\).

1.3 Further proof rules

There are many rules in type theory, and space does not allow us to give a complete discussion of them all. Fig. 5 includes most of the introduction and elimination rules in their simpler form, while Fig. 6 gives their complete form. Premises of the form ‘A type’ in these rules may be read simply as ‘A is a well-formed formula’ for the purposes of this discussion.

Note that occurrences of terms like b(x), B(x) and c(x, y) in these rules do not denote function applications. Instead they indicate that the given term is an expression composed of various signs which may or may not include the parenthesized signs. Thus B(x) signifies a term that

### Table: Abbreviated form of the introduction and elimination rules

<table>
<thead>
<tr>
<th>Introduction rules</th>
<th>Elimination rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>∧-I A B ( A \land B )</td>
<td>( \land-E ) ( A \land B \rightarrow C )</td>
</tr>
<tr>
<td>( \Rightarrow-I ) ( B \rightarrow A \Rightarrow B )</td>
<td>( \Rightarrow-E ) ( A \Rightarrow B \rightarrow A )</td>
</tr>
<tr>
<td>∨-I A B type ( A \lor B )</td>
<td>( \lor-E ) ( A \lor B \rightarrow \langle x: A (B(x)) \rangle )</td>
</tr>
<tr>
<td>( \exists-I ) ( a: A (B(a)) )</td>
<td>( \exists-E ) ( (\exists x: A (B(x)) \rightarrow C )</td>
</tr>
<tr>
<td>( \forall-I ) ( \langle x: A (B(x)) \rangle \rightarrow \langle x: A (B(x)) \rangle )</td>
<td>( \forall-E ) ( (\forall x: A (B(x)) \rightarrow a: A (B(a)) )</td>
</tr>
</tbody>
</table>

Figure 5. Abbreviated form of the introduction and elimination rules.
<table>
<thead>
<tr>
<th>Introduction rules</th>
<th>Elimination rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \land \mathbf{I} )</td>
<td>( \land \mathbf{E} )</td>
</tr>
<tr>
<td>( a: A ) ( b: B )</td>
<td>( p: A \land B )</td>
</tr>
<tr>
<td>( (a,b): A \land B )</td>
<td>( c(a,b): C((a,b)) )</td>
</tr>
<tr>
<td>( [x: A] )</td>
<td>let (a,b) ( \vDash ) ( p ) in ( c: C(p) )</td>
</tr>
<tr>
<td>( \Rightarrow \mathbf{I} )</td>
<td>( \Rightarrow \mathbf{E} )</td>
</tr>
<tr>
<td>( b(x): B )</td>
<td>( f: A \Rightarrow B )</td>
</tr>
<tr>
<td>( \lambda x. b: A \Rightarrow B )</td>
<td>( f a: B )</td>
</tr>
<tr>
<td>( \lor \mathbf{I} )</td>
<td>( \forall \mathbf{E} )</td>
</tr>
<tr>
<td>( a: A \lor B ) type</td>
<td>( p: A \lor B )</td>
</tr>
<tr>
<td>( A ) type</td>
<td>( c(a): C(\text{inl } a) )</td>
</tr>
<tr>
<td>( b: B )</td>
<td>( d(b): C(\text{inr } b) )</td>
</tr>
<tr>
<td>( \text{inl } a: A \lor B )</td>
<td>when ( p ) is</td>
</tr>
<tr>
<td>( \text{inr } b: A \lor B )</td>
<td>inl ( a ) then ( c(a) )</td>
</tr>
<tr>
<td>( \exists \mathbf{I} )</td>
<td>( \forall \mathbf{E} )</td>
</tr>
<tr>
<td>( a: A ) ( b(a): B(a) )</td>
<td>( p: (\exists x: A)B(x) )</td>
</tr>
<tr>
<td>( (a,b): (\exists x: A)B(x) )</td>
<td>( c(a,b): C((a,b)) )</td>
</tr>
<tr>
<td>( [x: A] )</td>
<td>let ( p = (a,b) ) in ( c: C(p) )</td>
</tr>
<tr>
<td>( \forall \mathbf{I} )</td>
<td>( \forall \mathbf{E} )</td>
</tr>
<tr>
<td>( b(x): B(x) )</td>
<td>( f: (\forall x: A)B(x) )</td>
</tr>
<tr>
<td>( \lambda x. b: (\forall x: A)B(x) )</td>
<td>( f a: B(a) )</td>
</tr>
<tr>
<td>( \mathbf{N}\mathbf{E} )</td>
<td>( n: N ) ( b: C(0) )</td>
</tr>
<tr>
<td>( n + 1: N )</td>
<td>( \text{i}(m, h): C(m + 1) )</td>
</tr>
</tbody>
</table>

Figure 6. Major introduction and elimination rules

may include the sign ‘\( \times \)’, e.g. \( x = 0 \), or may not include the sign ‘\( \times \)’ e.g. \( N \). This has two functions. The first is to indicate those expressions that should not be dependent on the variables introduced by the assumptions. (Compare the rules for \( \Rightarrow \)-introduction and \( \forall \)-introduction.) The second is to indicate the substitution of one sign for another. Thus in the \( \forall \)-elimination rule (Fig. 5) the occurrence of \( B(x) \) in the first premise and \( B(a) \) in the conclusion signifies that the sign ‘\( a \)’ is to be substituted for the sign ‘\( x \)’ everywhere the latter occurs in the expression \( B \).

The rules for \( \exists \) and \( \forall \) in Figs 5 and 6 are in fact generalizations of the \( \land \) and \( \Rightarrow \) rules discussed already. There are two \( \forall \)-introduction rules, namely, from a proof of \( A \) it is possible to infer \( A \lor B \) for any type \( B \), and from a proof of \( B \) it is possible to infer \( A \lor B \) for any type \( A \). The \( \forall \)-elimination rule corresponds to case analysis; if it has been established that just two cases \( A \) or \( B \) need to be considered, and in both cases the proposition \( C \) can be proved, then \( C \) is true in general.

A particular enumerated set is the empty set denoted by \( \{ \} \) or \( \emptyset \). This set is used to denote an absurdity or contradiction in type theory, since it would be absurd to construct an element of the empty set (Thus there is no introduction rule for the empty set.) The elimination rule for the empty set states that it is possible to infer any (well-formed) proposition from a contradiction.

The negation \( \neg A \) is modelled in type theory by the proposition \( A \Rightarrow \emptyset \), i.e. \( A \) is false if there is a function that constructs elements of the empty set from elements of \( A \). Care must be taken with the use of negation in type theory because laws such as the law of the excluded middle \( A \lor \neg A \) and the law of double negation \( \neg \neg A \Rightarrow A \) are not universally valid. We call a proposition \( A \) for which the law of the excluded middle is valid a decidable proposition.

The properties we deal with, like \( i < j \) where \( i \) and \( j \) are natural numbers, are all decidable, but an explicit proof of this fact must be given where the property is assumed. This figures in our formulation of the while-rule.

A derived rule that allows an abbreviation of the when-construct occurs when we have proofs of \( A \lor B \) and \( \neg B \). In these circumstances we have the formal rule

\[
\text{d: } A \lor B \quad c: \neg B
\]

when \( d \) is

- \( \text{inl } a \) then \( a \)
- \( \text{inr } b \) then \( z: A \)

but we abbreviate the when-construct to \( \text{outl } d \). Thus the rule becomes

\[
\text{d: } A \lor B \quad c: \neg B
\]

\( \text{outl } d: A \)

There is a similar combinator \( \text{outr } \) in the case that both \( A \lor B \) and \( \neg A \) are true.

1.4 Specifications

The set of natural numbers, \( N \), is also defined by introduction and elimination rules, the introduction rules being the familiar Peano axioms.
N-introduction 0: \( n \)  
\[ n: N \frac{n: N}{n + 1: N} \]

The N-elimination rule formalizes proof by induction:

\[ \frac{n: N \quad P(0) \quad [m: N, P(m)]}{P(n)} \]

The rule may be read as 'P(n) is true for a given natural number n if P(0) is true and P(m + 1) follows from the assumptions that m is a natural number and P(m) is true'.

Introducing additional objects into the rule, it may also be seen as defining primitive recursion.

\[ \frac{n: N \quad b: P(0) \quad [m: N, h: P(m)]}{i(m,h): P(m + 1)} \]

natrec \( h \ n \)
where \( h \ 0 \ \iff b \)
and \( h \ m + 1 \iff i(m, (h \ m)) \): \( P(n) \)

The abbreviations b, h and i have been used in the above rule for basis, hypothesis and induction step. The rule therefore states that if the given recursive definition of h is a proof of P(n) if b is a proof of P(0) and i is a proof of P(m + 1) whenever m is a natural number and h is a proof of P(m).

For example the addition of two natural numbers \( n \) and \( m \) can be defined inductively on \( n \) by the following:

\[ (\text{basis}) \quad m + 0 = m \]
\[ (\text{induction step}) \quad \text{if } m + n = q \text{ then } m + (n + 1) = q + 1 \]

Formulated using N-elimination we would thus obtain

\[ \text{plus} \in \lambda n. \lambda m. \text{natrec} \ m + \ n \]
where \( m + 0 = m \)
and \( m + p + 1 \iff (m + p) + 1 \)
\[ : N \Rightarrow (N \Rightarrow N) \]

In this way primitive recursive definitions of familiar arithmetic operators like multiplication (*) and division (div) may be defined.

Two more types that we need to discuss are the equality type and subtypes. The equality type is a family of types indexed by objects \( A \) of the universe \( U \). The proposition \( a = b \), read as 'a and b are equal objects of type A' is either the empty type, when \( a \) and \( b \) are unequal objects, or contains a single element denoted by \( e \), when \( a \) and \( b \) are indeed equal. Thus the statement

\[ e: a = b \]

may be read as 'a and b are provably equal in type A'.

Equality in type theory obeys the familiar laws of reflexivity, symmetry and transitivity. In addition several equalities can be identified involving objects constructed by the elimination rules. For complete details the reader is referred to Peterson.8,15,16

Using the equality type it is possible to define familiar relations, for example \( i \leq j \) on natural numbers

\[ i \leq j \iff (\exists k: N) (\text{plus} \ k \ i = j) \]

Note that a proof of \( i \leq j \) will be a pair consisting of a natural number and the object \( e \) of the equality type. For instance a proof of \( 0 \leq 1 \) is the pair \((1, e)\).

The wish to suppress uninteresting proof objects like \( e \) leads to the notion of subtypes, a notion not introduced in Martin-Löf’s original system but discussed by Backhouse,9 Constable1 and Petersson.15,16 The proposition \( (x: A) \ B(x) \), which is read as 'the set of objects of \( A \) satisfying \( B \) is non-empty', is identified with the type of all elements \( x \) of \( A \) for which \( B(x) \) can be proved. The rules governing subtypes given in Ref. 2 are similar to the rules for existential quantification except that the second component of any pair is omitted.

As an example of the use of subtypes the following is a tighter specification of the function plus defined earlier.

\[ +: \{f: N \Rightarrow (N \Rightarrow N) \mid (\forall n: N) (f \ n \ 0 = n)\} \]

This states that plus is a (curried) function on a pair of natural numbers having the property that \( 0 \) is a right identity.

It is always possible to recover an object of type \((\exists x: A) B(x)\) from an object of type \((x: A) \ B(x)\) whenever \( B \) is decidable. Specifically we have the theorem2

\[ \exists b. \lambda x. (x, \ \text{out}(b \ x)) : (\forall x: A) (B(x) \vee \neg B(x)) \Rightarrow [(x: A) \ B(x)] \Rightarrow (\exists x: A) B(x) \]

There are about 100 rules altogether describing type theory, and we have given less than a quarter. However, the rules given form the main core, the most important omissions being a formal statement of the rules governing equalities and the rules governing type formation. For a definitive statement of the complete set of rules used in the preparation of this paper excepting subtypes see Peterson.15,16 See Backhouse2 for a fuller discussion of the subtype rules.

With this basis it is now possible to give several examples of specifications in type theory. (Our examples are necessarily limited to problems involving the natural numbers since we have not discussed the type formation rules that permit the construction of lists, arrays, trees, etc.) In essence, most specifications we write take the form \((\forall x: I) (y: O \mid P(x, y))\). Such a specification is achieved by a function that maps arguments, \( x \), of the input type \( I \) into objects, \( y \), of the type \( O \) satisfying the required property \( P(x, y) \). For instance the specification of the integer square-root problem4 is the following

\[ (\forall i: N) \ {r: N \mid r^2 \leq i < (r + 1)^2} \]

The test whether \( i \) is a perfect square can be expressed as an instance of the law of the excluded middle.

\[ (\forall i: N) \ {r: N \mid r^2 = i} \lor \neg \{r: N \mid r^2 = i\} \]

The latter specification is achieved by a function that maps a natural number \( i \) into either the construction of a natural number \( r \) satisfying \( r^2 = i \) or a proof that such a construction is absurd, together with information represented by the constants \( \text{inl} \) and \( \text{inr} \) as to which alternative has been achieved.

The algorithm for integer remainder computation discussed later has the specification

\[ (\forall a: N) \ (\forall b: N \mid 0 < b) \]
\[ \{r: N \mid \{q: N \mid a = b \ast q + r \} \land r < b\} \]
This specification is achieved by a function having two arguments \(a\) and \(b\) that constructs a natural number \(r\) with the property that \(r\) is in the range \(0..b-1\) and the set of divisors \(q\) with the property that \(a = b*q + r\) is non-empty.

2. A WHILE-RULE

2.1 Formal statement

Informally the while-rule we consider can be stated as follows:

- The statement while \(- B\) do \(s\) can be proved correct with respect to preconditions \(P\) and postcondition \(Q\) if 
  (a) \(Q\) takes the form \(P \land B\).
  (b) \(P\) is maintained invariant by the loop body \(s\);
  (c) there is a bound function \(t\), which gives an upper bound on the number of iterations still to be performed, such that 
    (i) \(t \geq 0\)
    (ii) \(t\) is decreased by at least one each iteration.'

The formulation of this rule in type theory is almost straightforward, the only complication being the non-universality of the law of the excluded middle. Let \(O\) denote the type of the output values and, assuming that \(x:o\), let \(P(x):U\) denote the invariant property, \(t(x):N\) denote the bound function and \(B(x):U\) denote the termination property.

The requirement that the invariant property is also a precondition of the while-statement is to assert in type theory that the subtype of \(O\) consisting of objects \(x\) satisfying \(P\) is non-empty. I.e. we assume

\[
x_o : (x : O \mid P(x))
\]

(1)

Next, the function of the body of the loop to maintain the invariant property whilst decreasing \(t\) whenever the termination condition is not satisfied. To express this in type theory we require that \(B\) is decidable. In fact we impose the requirement that

\[
b : (\forall x : O) \{(B(x) \lor C(x))\}
\]

(2)

where \(C(x)\) is also of type \(U\) whenever \(x\) is of type \(O\). This is more general than the statement that \(B\) is decidable since we can always take \(C(x)\) to be \(- B(x)\), but our proof does not rely on such an identity. Then the property of the loop body \(s\) can be expressed as

\[
\forall x : (x : O \mid P(x) \land C(x)) \{y : O \mid P(y) \land t(y) < t(x)\}
\]

(3)

Property (3) can be read as Òs is a function mapping objects \(x\) in \(O\) satisfying the property \(C\) into objects \(y\) in \(O\) for which \(t(y) < t(x)\).Ó

The correctness of the while-statement is the claim that the conjunction of the three premises (1), (2) and (3) implies that there is a \(y\) in \(O\) that satisfies both the invariant property and the termination condition, i.e.

\[
\{y : O \mid P(y) \land B(y)\}
\]

(4)

Thus the formal statement of the while-rule is

\[(1) \land (2) \land (3) \Rightarrow (4)\]

By constructing a proof of this proposition we effectively obtain a program while\(x_o, b, s, t\) in type theory performing the same function as a while-statement in conventional programming languages.

2.2 The proof

The proof of the while-rule is a more-or-less conventional inductive proof on the value of the bound function. Assume (1), (2) and (3) and denote by \(Q\) the output specification (4).

The inductive hypothesis is that

\[
\{y : O \mid P(y) \land t(y) < n\} \Rightarrow Q
\]

where \(n\) is a natural number. Since

\[
t(x_o) < t(x_o) + 1
\]

it is straightforward to show that

\[
x_o : \{y : O \mid P(y) \land t(y) < (t(x_o) + 1)\}
\]

and hence that applying the inductive hypothesis to \(n = t(x_o) + 1\) and \(x_o\) establishes \(Q\).

The basis of the inductive proof is trivial since \(m < 0\) is a contradiction for all \(m\) in \(N\). In the inductive step we assume that \(y : \{y : O \mid P(y) \land t(y) < m + 1\} \Rightarrow Q\)

by a case analysis on \(B(y) \lor C(y)\). In the case \(B(y)\) the postcondition \(Q\) is established by \(y\) itself. In the case \(C(y)\) it is possible to apply the function \(s\) to \(y\) (see (3)) to obtain

\[
\{y : O \mid P(y) \land t(y) < m\}
\]

Thus applying the inductive hypothesis to \(s\) \(y\) establishes \(Q\).

The complete details of the proof are included in Appendix B. From that proof we note that the expression while \(x_o, b, s, t\) takes the following form:

\[
\begin{align*}
\text{natrec } w & \text{ t(x_o) + 1 } \\
\text{ where } w & 0 \iff \_x.z \\
\text{ and } w & (m + 1) \iff \_y.\text{when b y is } \\
& \text{ inl a then } y \\
& \text{ inr c then w m (s y)) x_o}
\end{align*}
\]

Using the equality rules for \(N\) it is possible to demonstrate that

\[
\text{while (x_o, b, s, t) = when b x_o = } \\
& \text{ inl a then } x_o \\
& \text{ inr c then while (s x_o, b, s, t),}
\]

an identity that is remarkably similar to

\[
\text{while } -b \text{ do s = if } -b \text{ then s; while } -b \text{ do s.}
\]

3. EXAMPLES

Having formally proved a while-rule it is now possible to construct type-theory programs in a conventional (imperative!) programming style. We illustrate this process on two well-known problems, remainder computation and the calculation of greatest common divisors. These problems were chosen as simple illustrations and because efficient solutions are not readily expressed using primitive recursion.

3.1. Remainder computation

Our first example is to compute the remainder \(r\) on dividing the given natural number \(a\) by the given, strictly positive natural number \(b\). As discussed earlier, the problem specification is therefore as follows.

\[
\forall a : N \forall b : (b : N \mid 0 < b)) \}\{r : N \mid q : N \mid a = b*q + r \land r < b\}
\]

32 THE COMPUTER JOURNAL, VOL. 30, NO. 1, 1987
Denoting \( b : \mathbb{N} \mid 0 < b \) by \( \mathbb{N}^+ \) and assuming \( a : \mathbb{N} \) and \( b : \mathbb{N}^+ \) the problem becomes that of showing that the set

\[
\{ r : \mathbb{N} \mid \{ q : \mathbb{N} \mid a = b \cdot q + r \} \land r < b \}
\]

is non-empty. To apply the while-rule we identify the output type \( O \) with \( \mathbb{N} \) and introduce the invariant property

\[
P(r) \equiv \{ q : \mathbb{N} \mid a = b \cdot q + r \}
\]

and the termination condition

\[
B(r) \equiv r < b.
\]

The non-termination condition is the obvious property

\[
C(r) \equiv b \leq r.
\]

We are now required to exhibit some value \( r_0 \) demonstrating that the invariant property can be established. Since \( a = b \cdot 0 + a \) the value \( a \) will do.

We are also required to demonstrate that, whenever the nontermination condition is satisfied, it is possible to maintain the invariant property whilst making progress towards the termination condition. In this case the bound function, \( t \), we use as a measure of progress is just the identity function so that \( t(r) = r \). Formally, therefore, we are required to construct a function \( s \) achieving the specification:

\[
(\forall r : \mathbb{N}) \{ q : \mathbb{N} \mid a = b \cdot q + r \} \land q \leq r \}
\]

\[
(\forall r : \mathbb{N}) \{ q : \mathbb{N} \mid a = b \cdot q + r \} \land r' < r
\]

Noting that

\[
a = b \cdot q + r \land b \leq r \Rightarrow a = b \cdot (q + 1) + (r - b)
\]

and

\[
0 < b \leq r \Rightarrow (r - b) < r
\]

it is clear that the function \( s = \lambda r. r - b \) has the desired property. Finally we are also required to show that \( t < \) is a decidable property. We assume that this is achieved by the function \( l \). Specifically,

\[
l(b) : (\forall r : \mathbb{N}) (r < b \lor \exists b \geq b)
\]

This completes the ingredients for the application of the while-rule and we conclude that the function

\[
\lambda a. \lambda b. \text{while} (a, l(b), \lambda r. r - b, r, r)
\]

achieves the original specification.

### 3.2. Greatest common divisors

Our second problem is the classic problem of determining the greatest common divisor of two positive integers \( a \) and \( b \).

A formal specification of this problem is based on the following definitions:

\[
\mathbb{N}^+ \equiv \{ x : \mathbb{N} \mid 0 < x \}
\]

\[
x \text{ divides } y \equiv \{ c : \mathbb{N} \mid y = c \cdot x \}
\]

\[
z \text{ is gcd } (x,y) \equiv z \text{ divides } x \land z \text{ divides } y \land
\]

\[
(\forall d : \mathbb{N}) (d \text{ divides } x \land d \text{ divides } y \Rightarrow d \leq z)
\]

The requirement to determine the greatest common divisor of \( a \) and \( b \) is expressed as a function with arguments \( a \) and \( b \) that returns a value \( z \) satisfying the relation \( z \text{ is gcd } (a,b) \):

\[
(\forall a : \mathbb{N}^+) (\forall b : \mathbb{N}^+) \{ z : \mathbb{N} \mid z \text{ is gcd } (a,b) \}
\]

Assuming that \( a \) and \( b \) are both in \( \mathbb{N}^+ \) the problem becomes that of showing that the set

\[
\{ z : \mathbb{N} \mid z \text{ is gcd } (a,b) \}
\]

is non-empty. It is as straightforward to prove in type theory as it is in classical mathematics that \( a \text{ is gcd } (a,a) \), so we apply the while-rule to the related problem:

\[
\{ (x,y) : \mathbb{N} \land \mathbb{N} \mid x = y \land (\forall z : \mathbb{N}) (z \text{ is gcd } (x,y) \Rightarrow z \text{ is gcd } (a,b)) \}
\]

The output type \( O \) is therefore taken to be \( \mathbb{N} \land \mathbb{N} \), the invariant property \( P \) is defined as

\[
P(x,y) \equiv (\forall z : \mathbb{N}) (z \text{ is gcd } (x,y) \Rightarrow z \text{ is gcd } (a,b))
\]

and the termination condition \( B \) is

\[
B(x,y) \equiv x = y,
\]

the nontermination property \( C \) being \(- (x = y)\).

Use of the while-rule requires us to exhibit initial values \( (x_0, y_0) \) satisfying the property \( P \). Clearly the pair \((a,b)\) suffices for this purpose. We are also required to exhibit a function \( s \) with argument a pair \((x,y)\) in \( \mathbb{N} \land \mathbb{N} \) that makes progress (as measured by some bound function \( t(x,y) \)) to the termination condition \( x = y \) whilst maintaining the property \( P \). The solution is the well-known one described by Dijkstra. We take \( x + y \) as the bound function and consider two cases, the first when \( x < y \) and the second when \( x > y \). In the first case we prove that

\[
z \text{ is gcd } (x,y-x) \Rightarrow z \text{ is gcd } (x,y)
\]

and \( x + (y-x) < x + y \)

and in the second case we prove that

\[
z \text{ is gcd } (x-y,y) \Rightarrow z \text{ is gcd } (x,y)
\]

and \( (x-y)+y < x+y \)

Thus, assuming that \( Is \) \((x,y)\) is of type \( x < y \lor y < x \) whenever the pair \((x,y)\) is of type \( (x,y) : \mathbb{N} \land \mathbb{N} \mid -(x = y) \), the two cases may be combined by the \( \lor \)-elimination rule to form the pair \((x',y')\) defined by the expression

\[
\text{when } Is \text{ (x,y) is}
\]

\[
\text{inl a then } \text{(x,y-x)}
\]

\[
\text{inr b then } \text{(x-y,y)}
\]

and having the properties \( P(x',y') \) and \( x'+y' < x+y \).

Putting these facts together using the while-rule we obtain the formal statement

\[
\text{while(}
\]

\[
(a,b),
\]

\[
\text{eq,}
\]

\[
\lambda (x,y). \text{when } Is \text{ (x,y) is}
\]

\[
\text{inl a then } \text{(x,y-x)}
\]

\[
\text{inr b then } \text{(x-y,y)}
\]

\[
\lambda (x,y). x+y
\]

\[
\text{)}
\]

\[
= \{ (x,y) : \mathbb{N} \land \mathbb{N} \mid x = y \land (\forall z : \mathbb{N}) (z \text{ is gcd } (x,y) \Rightarrow z \text{ is gcd } (a,b)) \}
\]

\[
\text{where eq: (V(x,y): N \land (x = y \lor -(x = y))).}
\]

THE COMPUTER JOURNAL, VOL. 30, NO. 1, 1987  33
CONCLUSION

Martin-Löf’s theory of types, viewed as a programming language, is a typed functional language without assignments or other imperative features. Our proof, therefore, of a while-rule – which is conventionally associated with imperative programming – may be hard for some readers to reconcile with the aims of functional programming, and they may be tempted to explain it away as just an instance of course-of-values recursion. But that would be to miss the point of the exercise. Our own view of functional versus imperative programming is very simple: it is clear that there is a great deal to be learned from the technique of functional programming, but it is also clear that a great deal has been learned already from the techniques of imperative programming. One of the attractions to us of Martin-Löf’s theory is that it does not constrain the programmer to one particular style of program development but, as we have attempted to show, permits different styles to be developed within the theory itself (so long as the style is mathematically sound, of course).

In our proof, the variant function is assumed to return a natural number. The proof can be generalised to an arbitrary well-founded ordering, but this would have added unnecessary complexity to the presentation. Readers interested in such a generalisation should consult Paulson’s report.14

The proof of the while-rule has been mechanically checked using the implementation of type theory given by Petersson15 to which has been added an implementation of the subtype rules.8 Full details can be obtained from the authors.

Acknowledgement

This work was supported by a grant from the Science and Engineering Research Council.

REFERENCES

12. B. Nordström and J. Smith, Propositions and Specifications of Programs in Martin-Löf’s Theory of Types. Department of Computer Science, University of Göteborg, Chalmers University of Technology, Sweden.
APPENDIX A: CORRESPONDENCE BETWEEN OUR AND MARTIN-LOF'S NOTATION

<table>
<thead>
<tr>
<th>Our notation</th>
<th>Martin-Löf's notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\exists x : A)B</td>
<td>(\exists x \in A)B</td>
</tr>
<tr>
<td>(a,b)</td>
<td>(a,b)</td>
</tr>
<tr>
<td>let (a,b) \iff p in c</td>
<td>(E_a b) (p,c)</td>
</tr>
<tr>
<td>A \lor B</td>
<td>A + B</td>
</tr>
<tr>
<td>inl a</td>
<td>i(a)</td>
</tr>
<tr>
<td>inr b</td>
<td>j(b)</td>
</tr>
<tr>
<td>when p is</td>
<td>(D a,b) (p,c,d)</td>
</tr>
<tr>
<td>inl a then c</td>
<td></td>
</tr>
<tr>
<td>inr b then d</td>
<td></td>
</tr>
<tr>
<td>(\forall x : A)B</td>
<td>(\forall x \in A)B</td>
</tr>
<tr>
<td>\alpha a \beta f a</td>
<td>(\alpha\beta a f a)</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>n+1</td>
<td>n'</td>
</tr>
<tr>
<td>natrec h n</td>
<td>(R m, h) (n, b, i)</td>
</tr>
<tr>
<td>where h 0 \iff b</td>
<td></td>
</tr>
<tr>
<td>and h m+1 \iff i</td>
<td></td>
</tr>
</tbody>
</table>

APPENDIX B: DETAILED PROOF OF THE WHILE-RULE

Note: Q is used as an abbreviation for \{y : O \land P(y) \land B(y)\}

1. b : (\forall x : O)(B(x) \lor C(x))
2. s : (\forall x : O)(B(x) \land C(x)) \land t(x) < t(x)
3. x_n : (x : O \land P(x))
   \% Prove by induction on n that \{y : O \land P(y) \land t(y) < n\} \implies Q
   \% Basis
   4. x : (y : O \land P(y) \land t(y) < 0)
   5. \emptyset (4, prop. of \land)
   6. z : Q (5, \emptyset-elim)
   7. \alpha x.z : (y : O \land P(y) \land t(y) < 0) \implies Q (4,6, \land-intr)
   \% Induction step
   \% *** see below ***
   8. t(x_n) + 1 : N
   9. natrec w t(x_n) + 1
      where w 0 \iff 2x.z
      and w (m+1) \iff \alpha y. when b y is
      inl a then y
      inr c then w m (s y)
     : (x : O \land P(x) \land t(x) < t(x_n) + 1) \implies Q (7,8,7.23,N-elim)
   10. x_n : (x : O \land P(x) \land t(x) < t(x_n) + 1) (Prop. of \land, 3 and sub-intro)
   11. (natrec w t(x_n) + 1...) x_n : Q (9,10, \land-intr)
   \% Induction step%

Ass. 7.1 m : N, w : (y : O \land P(y) \land t(y) < m) \implies Q
Ass. 7.2 y : (y : O \land P(y) \land t(y) < m + 1)
Ass. 7.3 y' : O, q : P(y') \land t(y') < m + 1
   \% Case analysis%
   7.4 b y' : B(y') \lor C(y') (1,7.3, \lor-elim)
Ass. 7.5 a : B(y')
   7.6 (fst q, a) : P(y') \land B(y') (7,3,7.5, \land-elim, \land-intro)
   7.7 y' : Q (7,3,7.6, sub-intro)
Ass. 7.8 c : C(y')
   7.9 (fst q, c) : P(y') \land C(y') (7,3,7.8, \land-elim, \land-intro)
   7.10 y' : (y : O \land P(y) \land C(y)) (7,3,7.9, sub-intro)
   7.11 s y' : (y : O \land P(y) \land t(y) < t(y')) (2,7.10, \land-intr)
   7.12 s y' : (y : O \land P(y)) \land t(y) < t(y') (7.11, Prop of subtypes)
Ass. 7.14 y' : (y : O \land P(y))
   7.15 1 : t(y') < t(y')
   7.16 \Gamma (snd q, 1) : t(y') < m
      (for some \Gamma, details of which are not needed, from 7.3,7.15, Prop. of \land)
   7.17 y' : (y : O \land P(y)) \land t(y) < m (7.14,7.16, sub-intro)
7.18 \( y^*: \{ y: \mathbb{O} \mid P(y) \wedge t(y) < m \} \)  (7.17 Prop. of subtypes)
7.19 \( w y^*: Q \)  (7.1, 7.18, \( \Rightarrow - \)elim)
7.20 \( w (s y')': Q \)  (7.12, 7.19, subt-elim)
7.21 when \( b y' \) is
   \[\text{inl} a \text{ then } y'\]
   \[\text{inr} c \text{ then } w (s y')': Q \]  (7.3, 7.7, 7.20, \( \lor - \)elim)
7.22 when \( b y \) is
   \[\text{inl} a \text{ then } y\]
   \[\text{inr} c \text{ then } w (s y): Q \]  (7.2, 7.21, subt-elim)
7.23 \( \lambda y. \) when \( b y \) is
   \[\text{inl} a \text{ then } y\]
   \[\text{inr} c \text{ then } w (s y)\]
\( \{ y: \mathbb{O} \mid P(y) \wedge t(y) < m + 1 \} \Rightarrow Q \)  (7.22, \( \Rightarrow - \)intr)