

# Terminating General Recursion \*

*Bengt Nordström*

Department of Computer Science  
University of Göteborg/Chalmers  
S-412 96 Göteborg  
Sweden

September 1987

## Abstract

In Martin-Löf's type theory, general recursion is not available. The only iterating constructs are primitive recursion over natural numbers and other inductive sets. The paper describes a way to allow a general recursion operator in type theory (extended with propositions). A proof rule for the new operator is presented. The addition of the new operator will not destroy the property that all well-typed programs terminate. An advantage of the new program construct is that it is possible to separate the termination proof of the program from the proof of other properties.

*C.R. Categories:* D.2.1, D.2.4, D.3.1, F.3.1, F.3.3.

*Key Words:* recursion, well-founded induction, programming logic, fixed point, termination proof.

## 1 Introduction

Martin-Löf's type theory [4] can be seen as a programming logic for a functional programming language. There is a formal system in which it is possible to express not only programs and their specifications but also derivations of programs. Versions of type theory have been implemented in Göteborg [10], Cornell [2] and Cambridge [9]. Using these systems it is possible to use the computer to check the correctness of program derivations.

Type theory relies heavily on the identification of sets, propositions and specifications.

The judgement

$$a \in A$$

can be read as:

---

\*This was published in BIT, Vol. 28, pp. 605-619, 1988

1.  $a$  is an element in the set  $A$
2.  $a$  is a construction (proof element) for the proposition  $A$
3.  $a$  is a solution of the problem  $A$
4.  $a$  is a program with type  $A$
5.  $a$  is a program which satisfies the specification  $A$
6.  $a$  is an implementation of the abstract data type specification  $A$ .

The last reading was shown in [5]. Typ theory is a theory for total correctness, that  $a \in A$  means that the program  $a$  terminates with a value in  $A$ . Compared with other functional languages type theory has a very rich type structure in that the type system can be used to completely express the task of the program. For instance, the type of a program solving Fermat's last theorem is

$$\{n \in \mathbf{N} \mid n > 2 \ \& \ (\exists x, y, z \in \mathbf{N})x^n + y^n = z^n\}$$

If the type structure is very strong, the program forming operations may seem a little weak, since general recursion is not available, only primitive recursion. From a metamathematical point of view, this is not a serious problem. We know that primitive recursion together with higher order functions give us a way to express all provably (in Peano's arithmetic) total functions. This, however, gives no relief to a programmer who really wants to write down a program! From a programming methodological point of view, it forces the programmer to prove the termination of the program at the same time as the program is derived. It is often convenient to be able to separate the termination proof from the rest of the correctness proof. Another serious problem is that it forces the programmer to in some sense estimate the number of iterations the program will make. This information is not always available, we will see an example of this later.

The purpose of this paper is to show that it is possible to extend type theory with an operator for general recursion, while still not giving up the requirement on termination.

## 2 The Identification of Propositions, Sets and Specifications

The major ideas behind the identification of sets, propositions and specifications were described in a previous article in BIT [7]. Here, only the main ideas will be sketched.

A proposition is identified with the set of its proof objects and a specification is identified with the set of all programs satisfying the specification. Martin-Löf's type theory is basically a theory about sets in which it is possible to interpret a

logic. The following identifications are done: The conjunction  $A \& B$  is identified with the Cartesian product  $A \times B$ , since a proof of  $A \& B$  consists of a proof of  $A$  and a proof of  $B$  (in the same way as an element in  $A \times B$  consists of an element in  $A$  and an element in  $B$ ). The disjunction  $A \vee B$  is identified with the disjoint union  $A + B$ , since a proof of  $A \vee B$  consists of a proof of  $A$  or a proof of  $B$ . The implication  $A \supset B$  is identified with the function set  $A \rightarrow B$  since a proof of  $A \supset B$  consists of a method which to an arbitrary proof of  $A$  yields a proof of  $B$ . Similarly, the proposition  $\exists x \in A. B(x)$  is interpreted as  $\Sigma x \in A. B(x)$ , the disjoint union of a family of sets. The reason is that a proof of  $\exists x \in A. B(x)$  consists of two things: an element  $a$  in  $A$  and a proof of  $B(a)$  in the same way as elements in  $\Sigma x \in A. B(x)$  consists of pairs  $\langle a, b \rangle$ , where  $a$  is an element in  $A$  and  $b$  is an element in  $B(a)$ . Finally, the proposition  $\forall x \in A. B(x)$  is interpreted as the set  $\Pi x \in A. B(x)$ , the Cartesian product of a family of sets. A proof of  $\forall x \in A. B(x)$  consists of a method which to an arbitrary element  $x$  in  $A$  yields a proof of  $B(x)$ . Likewise, the elements in  $\Pi x \in A. B(x)$  are functions which maps an element  $x$  in  $A$  to an element in  $B(x)$ . So these functions are more general than the functions in  $A \rightarrow B$  in that the type of the output depends on the value of the input.

The basic sets are  $N$  (the set of natural numbers),  $\emptyset$  (the empty set, used to interpret  $\perp$ , the absurdity),  $\top$  (the one element set, used to interpret the proposition which is always true), **Bool** and other enumeration sets. There is also the set  $a =_A b$ , which is non-empty exactly when  $a$  and  $b$  are equal elements in  $A$ . The set  $Wx \in A. B(x)$  is used for representing a large class of inductively defined sets and the set **U** contains (codings of) sets as elements.

To look at programs as proofs and identifying specifications with propositions gives good insights in basic problems concerning specification languages and programming languages. A more practical reason for this identification is that it cuts down the size of a programming logic, since one proof rule can be given different interpretations. For instance, if we identify proof objects and programs, the following rule for the Cartesian product <sup>1</sup>

$$\frac{p \in A \times B \quad e(x, y) \in C(\langle x, y \rangle) [x \in A, y \in B]}{\text{split}(p, e) \in C(p)}$$

also expresses the elimination rule for conjunction

---

<sup>1</sup>As is usual in natural deduction systems, the proof rule is written by listing the premises over the horizontal line and the conclusion under the line. The premises and the conclusion may in general depend on assumptions, only assumptions which are discharged by the rule are shown. The rule above should be read in the following way. If  $p \in A \times B$  and if  $e(x, y) \in C(\langle x, y \rangle)$  under the assumptions that  $x \in A$  and  $y \in B$  then  $\text{split}(p, e) \in C(p)$ . The program  $\text{split}(p, e)$  is computed by first computing the value of  $p$ . If the value is a pair  $\langle a, b \rangle$  then the value of  $\text{split}(p, e)$  is the value of  $e(a, b)$ . So  $\text{split}$  is used to express pattern matching over pairs. A more readable syntax for the  $\text{split}$ -expression would be

$$\text{cases } p \text{ of } \langle x, y \rangle : e' \text{ endcases}$$

where all free occurrences of the variables  $x$  and  $y$  become bound in the expression  $e'$ .

$$\frac{A \& B \text{ true} \quad C \text{ true } [A \text{ true}, B \text{ true}]}{C \text{ true}}$$

The latter being obtained from the former by reading the specifications  $A$ ,  $B$  and  $C$  as propositions, thereby ignoring the elements (programs, proof objects) in the rule. If we read  $A$  and  $B$  as specifications and  $C$  as a proposition we obtain the rule

$$\frac{p \in A \times B \quad C(\langle x, y \rangle) \text{ true } [x \in A, y \in B]}{C(p) \text{ true}}$$

So we have seen how one rule expresses at least three rules.

### 3 Programs are not Proofs

But there are problems with looking at programs as proofs. The first problem has to do with the level of formalization. If we want our programs to be executed on a computer they have to be completely formalized. So if we identify programs and proofs this mean that the whole derivation of the program (including the correctness proof) has to be formalized. No one would like to work in such a strait jacket! Informal reasoning about programs will always be important.

The first step towards making the distinction between proofs and programs was already taken in the 1979 version of Martin-Löf's type theory [3] where the constant  $\text{id}$  is the (only) element in  $a =_A b$  if  $a = b \in A$ . The program  $\text{id}$  doesn't have enough structure to represent the proof of  $a = b \in A$ . Nobody would accept  $\text{id}$  as a proof that  $a^n + b^n = c^n$  for some natural numbers  $a$ ,  $b$ ,  $c$  and  $n > 2$ . So the distinction between programs and proofs is clear here.

The next step towards this distinction is the subset type former which was introduced in [6] to remove some computationally irrelevant details from the programs (proof objects). The proof of an existential statement  $\exists x \in A. B(x)$  is a pair  $\langle a, b \rangle$ , where  $a \in A$  and  $b \in B(a)$ . But in most cases where we want to read the existential statement as a specification of a program we are not interested in the second component. It is just there as some kind of (formal) comment that the first component has the property  $B$ . As an example, consider a specification of the sorting problem:

$$\forall x \in \text{ListN}. \exists y \in \text{ListN}. \text{Op}(y, x)$$

where  $\text{Op}(y, x)$  is a predicate which is true if  $y$  is an ordered permutation of  $x$ . If we solve this problem and get a program  $s$  then  $s$  is a function which when applied to a list  $l$  yields a pair  $\langle a, b \rangle$ , the first component  $a$  being an ordered permutation of  $l$  and the second a proof object of  $\text{Op}(a, l)$ . So to get a sorting algorithm we have to construct a new program

$$\text{sort} =_{\text{def}} \lambda x. \text{fst}(\text{apply}(s, x))$$

whose only purpose is to cut away the “comment” in the second component.

The suggested solution to this is to use a subset

$$\{x \in A \mid B(x)\}$$

which is the set of all elements  $a$  in  $A$  for which  $B(a)$  is true. So this subset is like the set  $\Sigma x \in A. B(x)$  except that the second component (the proof object) in each element has been removed. It is now possible to give a correct specification of the sorting problem:

$$\Pi x \in \text{ListN}. \{y \in \text{ListN} \mid \text{Op}(y, x)\}$$

This is a correct specification in the sense that the programs satisfying it are exactly the set of all sorting programs, i.e. programs which when applied to a list outputs an ordered permutation of it.

Another problem with looking at programs as proofs is related to the difference in the repetitive constructs in programs and proofs. In programming languages, there are often potentially nonterminating constructs (like **while**-loops and **repeat**-loops in imperative languages programs and general recursion in functional languages). The repetitive constructs in proofs, on the other hand, are usually based on some kind of structural induction which by its very nature always terminate. The typical example being the rule for mathematical induction:

$$\frac{p \in \mathbf{N} \quad C(0) \text{ true} \quad C(\text{succ}(x)) \text{ true} \ [x \in \mathbf{N}, C(x) \text{ true}]}{C(p) \text{ true}}$$

So programmers often have to provide a separate proof of termination of their programs. Mathematicians seldom prove that their proofs terminate! If they do, they express it as a justification of the correctness of the induction principle they use.

Let’s look at an example where this difference of view becomes important. The problem is to write a program whose input is a function  $f \in \mathbf{N} \rightarrow \mathbf{N}$  which has a root, and the output is a root of the function. A root of a function  $f$  is a number  $x$  such that  $f(x) = 0$ . Using type theory with subsets we have the following specification

$$\Pi f \in \{g \in \mathbf{N} \rightarrow \mathbf{N} \mid \exists x \in \mathbf{N}. g(x) =_{\mathbf{N}} 0\}. \{y \in \mathbf{N} \mid f(y) =_{\mathbf{N}} 0\}.$$

But this problem doesn’t seem possible to solve in type theory! The reason is that we cannot prove the termination of such a program without having more information about the proof of  $\exists x \in \mathbf{N}. g(x) =_{\mathbf{N}} 0$ . This information is available if we use the  $\Sigma$ -type instead of the subset type:

$$(\Pi f \in \Sigma g \in \mathbf{N} \rightarrow \mathbf{N}. \exists x \in \mathbf{N}. g(x) =_{\mathbf{N}} 0) \{y \in \mathbf{N} \mid f(y) =_{\mathbf{N}} 0\}$$

but then we get a totally different problem which, moreover, is totally uninteresting. The input of this problem is a triple  $\langle g, a, b \rangle$  where  $g \in \mathbf{N} \rightarrow \mathbf{N}$ ,  $a$  is a root of  $g$  and  $b$  is a proof object of this fact, i.e.  $b \in g(a) =_{\mathbf{N}} 0$ . So this problem is solved by returning the second component of the input.

To cope with situations like this, type theory (with propositions and subsets) can be extended with a program construct for general recursion without giving up the idea that all well-typed programs terminate. Before I suggest one way of doing this it is necessary to briefly describe the syntax of type theory and give a short description of the rule for mathematical induction.

## 4 The Syntax of Type Theory

Expressions in type theory are built up from constants and variables using application and abstraction. An expression which cannot be applied to an argument is called *saturated*. For instance, in the Natural Induction rule below,  $p$ ,  $\mathbf{N}$ ,  $d$ ,  $C(0)$ ,  $e(x, y)$ ,  $C(\text{succ}(x))$ ,  $x$ ,  $y$ ,  $\text{natrec}(p, d, e)$ ,  $C(p)$  are saturated. The expression  $C$  is unsaturated, it expects one (saturated) argument, the expression  $e$  expects two saturated arguments and, finally, the primitive constant  $\text{natrec}$  expects two saturated arguments and one unsaturated argument which expects two saturated arguments.

If  $x$  is a variable and  $e$  is an expression then  $x.e$  will denote the abstraction of  $e$  with respect to  $x$ . Sometimes, applications  $f(a)$  will be written as  $fa$  and repeated applications  $f(a)(b)$  will be written  $f(a, b)$  and repeated abstractions  $x.(y.e)$  will be written  $x.y.e$ . For instance,  $\lambda x.e$  is shorthand for  $\lambda(x.e)$ ,  $\lambda$  is a constant which yields a saturated expression when applied to an abstraction. In the case that  $f$  is a saturated expression, then  $f(a)$  will stand for  $\text{apply}(f, a)$ .

## 5 Primitive Recursion and Mathematical Induction

Consider the rule for natural induction in type theory:

Natural Induction

$$\frac{p \in \mathbf{N} \quad d \in C(0) \quad e(x, y) \in C(\text{succ}(x)) \ [x \in \mathbf{N}, y \in C(x)]}{\text{natrec}(p, d, e) \in C(p)}$$

The rule can be read in the following way: We may draw the conclusion  $\text{natrec}(p, d, e) \in C(p)$  if  $p \in \mathbf{N}$ ,  $d \in C(0)$  and if  $e(x, y) \in C(\text{succ}(x))$  under the assumptions that  $x \in \mathbf{N}$  and  $y \in C(x)$ .

Using currying, we can slightly rewrite the rule as:

$$\frac{p \in \mathbf{N} \quad d \in C(0) \quad e(x) \in C(x) \rightarrow C(\text{succ}(x)) \ [x \in \mathbf{N}]}{\text{natrec}(p, d, e) \in C(p)}$$

So the problem  $C(p)$  is solved by the program  $\text{natrec}(p, d, e)$  if  $d$  solves the problem  $C(0)$  and  $e(x)$  is a “step-function” taking a solution of the problem  $C(x)$  to a solution of the problem  $C(\text{succ}(x))$ . The justification of the rule is based on

the semantics of type theory and the computation rule for the primitive recursion operator `natrec`:

If the value of  $p$  is  $0$  then the value of `natrec`( $p, d, e$ ) is the value of  $d$  which solves the problem  $C(0)$ .

If the value of  $p$  is `succ`( $0$ ) then the value of `natrec`( $p, d, e$ ) is the value of  $e(0)(d)$  which solves the problem  $C(\text{succ}(0))$  since  $e(0) \in C(0) \rightarrow C(\text{succ}(0))$  and  $d \in C(0)$ .

If the value of  $p$  is `succ`(`succ`( $0$ )) then the value of `natrec`( $p, d, e$ ) is the value of  $e(\text{succ}(0))(e(0)(d))$  which solves the problem  $C(\text{succ}(0))$  since  $e(\text{succ}(0)) \in C(\text{succ}(0)) \rightarrow C(\text{succ}(\text{succ}(0)))$  and  $e(0)(d) \in C(\text{succ}(0))$ .

In general, if the value of  $p$  is `succ`( $a$ ), then the value of `natrec`( $p, d, e$ ) is the value of  $e(a)(\text{natrec}(a, d, e))$  which solves the problem  $C(\text{succ}(a))$  since  $e(a) \in C(a) \rightarrow C(\text{succ}(a))$  and `natrec`( $a, d, e$ )  $\in C(a)$ .

## 6 Course-of-values Recursion and Complete Induction

In course-of-values recursion we have a step-function  $e(x)$  which takes a solution of all problems  $C(0), C(\text{succ}(0)), \dots, C(x)$  to a solution of the problem  $C(\text{succ}(x))$ . How can we express this? We want to have a function  $e(x)$  which as argument takes a list or a tuple  $\langle g_0, g_1, \dots, g_x \rangle$ , where  $g_i \in C(i)$ . A convenient way to express this is that the argument of  $e(x)$  is a function  $g$  such that  $g(i) \in C(i)$  for  $i \leq x$ . This is an element in a Cartesian product of a family of sets, i.e.

$$e(x) \in \left( \prod_{i \leq x} C(i) \right) \rightarrow C(\text{succ}(x))$$

where  $\prod_{i \leq x} C(i)$  is the set of functions  $b$  such that  $b(i) \in C(i)$  for  $i \leq x$ .

We can also express the requirement on the step-function  $e(x)$  as:

$$e(x, y) \in C(\text{succ}(x)) \quad [x \in \mathbf{N}, y(z) \in C(z) \quad [z \leq x \text{ true}]].$$

So we obtain the following rule:

Course-of-values Induction 1

$$\frac{p \in \mathbf{N} \quad d \in C(0) \quad e(x, y) \in C(\text{succ}(x)) \quad [x \in \mathbf{N}, y(z) \in C(z) \quad [z \leq x \text{ true}]]}{\text{covrec}(p, d, e) \in C(p)}$$

where `covrec` is a new primitive constant which is computed in the following way:

The value of `covrec`( $p, d, e$ ) is obtained by first computing the value of  $p$ . If the result is *zero* then the value of `covrec`( $p, d, e$ ) is the value of  $d$ . If the result is `succ`( $a$ ), then the value of `covrec`( $p, d, e$ ) is the value of  $e(a, z.\text{covrec}(z, d, e))$ , where  $z.e$  is the notation for the abstraction of  $e$  with respect to the variable  $z$ .

So if the value of  $p$  is  $0$  then the value of `covrec`( $p, d, e$ ) is the value of  $d$  which solves the problem  $C(0)$ . If the value of  $p$  is `succ`( $0$ ), then the value of

$\text{covrec}(p, d, e)$  is the value of  $e(0, z.\text{covrec}(z, d, e))$ . But  $e(0, y) \in C(\text{succ}(0))$  if  $y$  is a function such that  $y(z) \in C(z)$ , for  $z \leq 0$ . But  $z.\text{covrec}(z, d, e)$  is such a function since  $\text{covrec}(0, d, e) \in C(0)$ .

If the value of  $p$  is  $\text{succ}(\text{succ}(0))$  then the value of  $\text{covrec}(p, d, e)$  is the value of  $e(\text{succ}(0), z.\text{covrec}(z, d, e))$  but  $e(\text{succ}(0), y) \in C(\text{succ}(\text{succ}(0)))$  if  $y$  is a function such that  $y(z) \in C(z)$  for  $z \leq \text{succ}(0)$ . And  $z.\text{covrec}(z, d, e)$  is such a function since  $\text{covrec}(0, d, e) \in C(0)$  and  $\text{covrec}(\text{succ}(0), d, e) \in C(\text{succ}(0))$ .

We can simplify the rule for course-of-values induction if we instead have a step function  $e(x)$  which takes a solution of all problems strictly smaller than  $x$  to a solution of  $C(x)$ . We can then drop the second premise:

Course-of-values Induction 2

$$\frac{p \in \mathbf{N} \quad e(x, y) \in C(x) [x \in \mathbf{N}, y(z) \in C(z) [z < x \text{ true}]]}{\text{rec}'(p, e) \in C(p)}$$

where the value of  $\text{rec}'(p, e)$  is computed by computing the value of  $e(p, z.\text{rec}'(z, e))$ .

We get the rule for complete induction if we take away some of the constructions in the previous rule:

Complete Induction

$$\frac{p \in \mathbf{N} \quad C(x) \text{ true} [x \in \mathbf{N}, C(z) \text{ true} [z < x \text{ true}]]}{C(p) \text{ true}}$$

In the following, the notation  $a < b$  will be used instead of  $a < b \text{ true}$ .

## 7 General Recursion and Well-founded Induction

There is nothing in the rule for course-of-values induction which is particular to the set of natural numbers. The reason the rule works is that  $\mathbf{N}$  is well-ordered by  $<$ . We can generalize the rule to an arbitrary set  $A$  which is well-ordered by a relation  $\prec_A$ . The proposition  $\text{Wellordered}(A, \prec_A)$  will be used to express that  $A$  is well-ordered by  $\prec_A$ . The exact formulation of this is postponed to the section about the set of accessible elements. The computation rule becomes a little simpler if we change the order of the arguments:

Recursion rule

$$\frac{\text{Wellordered}(A, \prec_A) \quad p \in A \quad e(x, y) \in C(x) [x \in A, y(z) \in C(z) [z \prec_A x]]}{\text{rec}(e, p) \in C(p)}$$

with the following computation rule for  $\text{rec}$ :

The value of  $\text{rec}(e, p)$  is the value of  $e(p, \text{rec}(e))$ .

Notice that we get a binary fix-point operator by defining

$$Y_2(e, p) =_{\text{def}} \text{rec}(x.y.e(y, x), p)$$

because then  $Y_2(e)$  is a fix-point of  $e$ , i.e.  $e(Y_2(e)) = Y_2(e)$ , i.e.  $e(Y_2(e))(p) = Y_2(e)(p)$  since

$$\begin{aligned}
Y_2(e)(p) &\equiv Y_2(e, p) \\
&\equiv \text{rec}(x.y.e(y, x), p) \\
&\Rightarrow (x.y.e(y, x))(p, \text{rec}(x.y.e(y, x))) \\
&\equiv e(y, x) [x := p, y := \text{rec}(x.y.e(y, x))] \\
&\equiv e(\text{rec}(x.y.e(y, x)), p) \\
&\equiv e(Y_2(e), p)
\end{aligned}$$

where

$\Rightarrow$  stands for “computes in one step to”

$e[x := a, y := b]$  stands for the expression obtained from  $e$  by simultaneous substitution of the variable  $x$  with  $a$  and the variable  $y$  with  $b$ .

So we get the fix-point operator by swapping the arguments of the first argument to  $\text{rec}$ . If we try to formulate the previous rule using the  $Y_2$ -combinator directly we get the following

Wrong recursion rule

$$\frac{\text{Wellordered}(A, \prec_A) \quad p \in A \quad e(y, x) \in C(x) [y(z) \in C(z) [z \prec_A x], x \in A]}{Y_2(e, p) \in C(p)}$$

which doesn't make sense since the first argument of  $e$  depends on the second.

## 8 A Simple Example: The Termination of Quicksort

The scheme which will be used for solving recursion equations is that if

$$f(z) = e(z, f)$$

is a recursion equation then it can be solved by defining

$$f =_{\text{def}} \text{rec}(e)$$

where we in  $e$  have abstracted the two variables  $z$  and  $f$ . This is correct provided that the requirements on the parameters hold as expressed by the recursion rule. In the simple case that the family  $C(x)$  of sets over  $A$  does not depend on  $x$ , the requirements are that in the equation  $f(z) = e(z, f)$ ,  $f$  is a function from  $A$  to  $C$ ,  $z$  is an element in  $A$  and  $f$  must only be applied to arguments smaller than  $z$  on the right hand side of the equation. These requirements are exactly those which are used by programmers of functional languages in informal termination proofs.

Let's look at a termination proof of quicksort. The recursion equations for quicksort are:

$$\begin{aligned} \text{quick}(\text{nil}) &= \text{nil} \\ \text{quick}(\text{cons}(a, s)) &= \text{quick}(\text{le}(s, a)) \text{appendcons}(a, \text{quick}(\text{gt}(s, a))) \end{aligned}$$

where  $\text{nil}$  is the empty list and  $\text{cons}(a, s)$  is the list with  $a$  as the first element and the list  $s$  as the rest,  $\text{le}(s, a)$  is the list obtained from  $s$  by taking away all elements which are greater than  $a$  and  $\text{gt}(s, a)$  is the list where all elements of  $s$  strictly smaller than  $a$  have been removed. To solve the equations we have to first rewrite them using the `listcases`-expression:<sup>2</sup>

$$\text{quick}(p) = \text{listcases}(p, \text{nil}, \text{a.s.}(\text{quick}(\text{le}(s, a)) \text{appendcons}(a, \text{quick}(\text{gt}(s, a))))))$$

This equation can be solved by making the definition:

$$\text{quick} =_{\text{def}} \text{rec } x . y . \text{listcases}(x, \text{nil}, \text{a.s.}(y(\text{le}(s, a)) \text{appendcons}(a, y(\text{gt}(s, a))))))$$

In order to show that  $\text{quick}$  terminates it is enough to show that<sup>3</sup>

$$\text{quick} \in \text{List}N \rightarrow \text{List}N$$

In order to show this we have to show that

$$\text{listcases}(x, \text{nil}, \text{a.s.}(y(\text{le}(s, a)) \text{appendcons}(a, y(\text{gt}(s, a)))))) \in \text{List}N$$

under the assumptions that  $x \in \text{List}N$ ,  $y(z) \in \text{List}N$  [ $z \in \text{List}N$ ,  $z \prec x$ ] for some well founded relation  $\prec$ .

This can be shown by induction over  $x$ . The base case is trivial. If  $x = \text{cons}(a, s)$  we have to show that

$$y(\text{le}(s, a)) \text{appendcons}(a, y(\text{gt}(s, a))) \in \text{List}N$$

---

<sup>2</sup>The primitive constant `listcases` is used to express pattern-matching over lists. The value of `listcases(p, d, e)` is computed by first computing the value of  $p$ . If the value of  $p$  is `nil`, then the value of `listcases(p, d, e)` is the value of  $d$ . If the value of  $p$  is `cons(a, s)` then the value of `listcases(p, d, e)` is the value of  $e(a, s)$ . An alternative syntax for the `listcases`-expression would be

$$\text{cases } p \text{ of } \text{nil} : d, \text{cons}(x, y) : e' \text{ endcases}$$

where all free occurrences of the variables  $x$  and  $y$  becomes bound in the expression  $e'$ .

<sup>3</sup>The meaning of  $a \in A$  is that the computation of  $a$  *terminates* with a value in  $A$ .

which holds if

$$\begin{cases} y(\text{le}(s, a)) \in \text{List}N \\ y(\text{gt}(s, a)) \in \text{List}N \end{cases}$$

since the concatenation operator *append* takes two lists to a list. But this follows from the induction assumption if we can find a well-ordering  $\prec$  on  $\text{List}N$  such that

$$\text{le}(s, a) \prec \text{cons}(a, s)$$

and

$$\text{gt}(s, a) \prec \text{cons}(a, s)$$

but this holds if we define  $\prec$  to be

$$a \prec b =_{\text{def}} \#a <_{\mathbf{N}} \#b$$

where  $\#a$  is the number of elements in the list  $a$  and  $<_{\mathbf{N}}$  is the usual order on natural numbers.

The relation  $\prec$  defined in this way is well-founded (i.e.  $\prec$  well-orders  $\text{List}N$ ) since it is the inverse image of the well-founded relation  $<_{\mathbf{N}}$  on  $\mathbf{N}$ . That  $<_{\mathbf{N}}$  is well-founded comes from the fact that it is the transitive closure of the well-founded relation  $<_{\mathbf{N}}$  defined by  $a <_1 b =_{\text{def}} b =_{\mathbf{N}} a$ . To prove these facts formally it is necessary to extend the formal system of type theory with more rules about set equality. One suggestion how to do this is presented in [11]<sup>4</sup> which also presents other applications of the recursion rule and the set of accessible elements presented in the next chapter.

This termination proof should be compared with Smith's [12]. He proves the termination completely within type theory by translating course-of-values recursion over lists to primitive recursion over the length of the list.

## 9 The Set of Accessible Elements

Let  $\prec$  be a binary relation on a set  $A$ , i.e.  $\prec(x, y)$  is a proposition for  $x, y \in A$ . The notation  $x \prec y$  and  $y \succ x$  will be used instead of  $\prec(x, y)$  and it will sometimes be read as  $x$  is smaller than  $y$  even if the relation doesn't have to be an ordering relation. The set  $\mathbf{Acc}(A, \prec)$  of accessible element of  $\prec$  in  $A$  is the set of elements  $a \prec A$  such that there is no infinite descending sequence  $a \succ a_0 \succ a_1 \succ a_2 \dots$ . The set  $\mathbf{Acc}(A, \prec)$  is called the well-founded part of  $\prec$  by Aczel in [1]. The set  $A$  is well ordered by  $\prec$  if  $A = \mathbf{Acc}(A, \prec)$ .

A constructively more useful description of the set  $\mathbf{Acc}(A, \prec)$  is the following:

The element  $a \in A$  is accessible if all elements smaller than  $a$  are accessible. In particular, if  $a$  is an initial element, i.e. if  $\neg(x \prec a)$ , for  $x \in A$ , then  $a$  is accessible.

---

<sup>4</sup>This is an example that the relation "a refers to b" is not a well founded relation on the set of papers.

This can be formalized by the following introduction rule for  $\mathbf{Acc}(A, \prec)$ :

**Acc-introduction**

$$\frac{a \in A \quad y \in \mathbf{Acc}(A, \prec) [y \in A, y \prec a]}{a \in \mathbf{Acc}(A, \prec)}$$

The other rules are:

**Acc-formation**

$$\frac{A \text{ set} \quad x \prec y \text{ prop } [x \in A, y \in A]}{\mathbf{Acc}(A, \prec) \text{ set}}$$

**Acc-elimination**

$$\frac{p \in \mathbf{Acc}(A, \prec) \quad e(x, y) \in C(x) [x \in \mathbf{Acc}(A, \prec), y(z) \in C(z)[z \in A, z \prec x]]}{\text{rec}(e, p) \in C(p)}$$

**Acc-equality**

$$\frac{p \in \mathbf{Acc}(A, \prec) \quad e(x, y) \in C(x) [x \in \mathbf{Acc}(A, \prec), y(z) \in C(z)[z \in A, z \prec x]]}{\text{rec}(e, p) = e(p, \text{rec}(e)) \in C(p)}$$

The elimination rule collapses to the recursion rule above if  $A$  is well-ordered by  $\prec$ , due to the fact that  $A$  is well-ordered if it is equal to the set  $\mathbf{Acc}(A, \prec)$ .

An alternative formulation of this set is to start with a family  $B(x)$  of sets over  $A$ . Intuitively, the set  $B(x)$  is the set of immediate predecessors from  $x$ , it is related to the relation  $\prec$  in the following way:

$$B(x) \equiv \{y \in A \mid y \prec x\}.$$

We get the following rules:

**Acc'-formation**

$$\frac{A \text{ set} \quad B(x) \text{ set } [x \in A]}{\mathbf{Acc}'(A, B) \text{ set}}$$

**Acc'-introduction**

$$\frac{a \in A \quad y \in \mathbf{Acc}'(A, B) [y \in B(a)]}{a \in \mathbf{Acc}'(A, B)}$$

**Acc'-elimination**

$$\frac{p \in \mathbf{Acc}'(A, B) \quad e(x, y) \in C(x) [x \in \mathbf{Acc}'(A, B), y(z) \in C(z)[z \in B(x)]]}{\text{rec}(e, p) \in C(p)}$$

We notice that  $\mathbf{Acc}'(A, B)$  is a solution to the equation

$$\mathbf{Acc}'(A, B) = \{a \in A \mid B(a) \subset \mathbf{Acc}'(A, B)\}$$

So  $a$  is an element in  $\mathbf{Acc}'(A, B)$  iff all elements in  $B(a)$  are elements in  $\mathbf{Acc}'(A, B)$ .

## 10 The Termination of a Recursive max-function

Consider the following equation:

$$f(n, k) = \begin{array}{l} \text{if } n \geq k \text{ then } n \\ \text{else } f(f(n + 1, k), k) \end{array}$$

The problem is to show that this function terminates for all natural numbers. The example is a version of McCarthy's 91-function.

The termination will be proven by proving something stronger, namely that if

$$m(n, k) \stackrel{\text{def}}{=} \text{rec}(n.f.\text{if } n \geq k \text{ then } n \\ \text{else } f(f(n + 1))) (n)$$

then  $m(n, k) \in \text{Max}(n, k)$ , for  $n, k \in \mathbf{N}$ . The set  $\text{Max}(n, k)$  is the set containing the maximal value of  $n$  and  $k$ . By applying the computation rules for  $\text{rec}$  we see that  $m(a, b)$  is computed in the same way as the function  $f(a, b)$  does, interpreting equality as rewriting.

To prove termination, assume  $n, k \in \mathbf{N}$ . We want to show that  $m(n, k) \in \text{Max}(n, k)$  by using the recursion rule, so we have to find a well-ordering on  $\mathbf{N}$  and prove that

$$\text{if } n \geq k \text{ then } n \text{ else } f(f(n + 1)) \in \text{Max}(k, n) \quad (1)$$

under the assumptions that  $n \in \mathbf{N}$  and  $f(z) \in \text{Max}(k, z)$  [ $z \prec n$ ] for some well-ordering  $\prec$ . The well-ordering we will use is the reverse ordering relation on natural numbers making all numbers greater than  $k$  initial:

$$z \prec n \stackrel{\text{def}}{=} n < z \leq k$$

To prove (1) we consider two cases. If  $n \geq k$  then  $n \in \text{Max}(k, n)$ . If  $n < k$ , then we must show that

$$f(f(n + 1)) \in \text{Max}(k, n).$$

Since  $n + 1 \prec n$  we can use the induction assumption to conclude that  $f(n + 1) \in \text{Max}(k, n + 1)$ , and hence that  $f(n + 1) = k$ , since  $n + 1 \leq k$ . But then  $f(n + 1) \prec n$  and we can apply the induction assumption again to obtain  $f(f(n + 1)) \in \text{Max}(k, n)$ .

Notice that this termination proof is obtained simultaneously as we prove that  $m$  computes the maximal value of its arguments.

## 11 Remark

If we extend the program forming operations in type theory with the  $\text{rec}$ -operator we can replace all operators for primitive recursion with operators for pattern-matching. We would then have a language which is much more similar to other functional languages.

**Theorem 1** *All iterating constructs in type theory can be reduced to pattern matching and the general recursion operator `rec`.*

**Proof outline:** For instance, if we define <sup>5</sup>

$$\text{natrec}(p, d, e) =_{\text{def}} \text{rec}(x.y.\text{natcases}(x, d, z.e(z, y(z))), p)$$

The value of `natrec`( $p, d, e$ ) is then the value of

$$\begin{aligned} & \text{natcases}(x, d, z.e(z, y(z))) [x := p, y := \text{rec}(x'.y'.\text{natcases}(x', d, z.e(z, y'(z))))] \\ \equiv & \text{natcases}(p, d, z.e(z, \text{rec}(x'.y'.\text{natcases}(x', d, z.e(z, y'(z))), z))) \\ \equiv & \text{natcases}(p, d, z.e(z, \text{natrec}(z, d, e))) \end{aligned} \quad (2)$$

If the value of  $p$  is `0` then the value of (2) is the value of  $d$ . If the value of  $p$  is `succ`( $a$ ), then the value of (2) is the value of  $e(a, \text{natrec}(a, de))$ . So we have shown that `natrec` as defined above is computed in the same way as the traditional primitive recursion operator in type theory.

The `listrec`-operator and the `wrec`-operator can be defined similarly.  $\square$

## 12 Acknowledgements

I want to thank Peter Dybjer, Per Martin-Löf, Kent Petersson and Jan Smith for many discussions about recursion and induction.

## References

- [1] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–, North-Holland Publishing Company, 1977.
- [2] R. L. Constable et. al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [3] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175, North-Holland, 1982.

---

<sup>5</sup>The primitive constant `natcases` is used to express pattern-matching over natural numbers. The value of `natcases`( $p, d, e$ ) is computed by first computing the value of  $p$ . If the value of  $p$  is `0`, then the value of `natcases`( $p, d, e$ ) is the value of  $d$ . If the value of  $p$  is `succ`( $a$ ) then the value of `natcases`( $p, d, e$ ) is the value of  $e(a)$ . A more readable syntax for the `natcases`-expression would be

$$\text{cases } p \text{ of } 0 : d, \text{succ}(x) : e' \text{ endcases}$$

where all free occurrences of the variable  $x$  becomes bound in the expression  $e'$ .

- [4] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
- [5] Bengt Nordström and Kent Petersson. *The Semantics of Module Specifications in Martin-Löf's Type Theory*. PMG Memo ??, Chalmers University of Technology, S-412 96 Göteborg, 1987.
- [6] Bengt Nordström and Kent Petersson. Types and specifications. In R. E. A. Mason, editor, *Proceedings of IFIP 83*, pages 915–920, Elsevier Science Publishers, October 1983.
- [7] Bengt Nordström and Jan Smith. Propositions, types and specifications in Martin-Löf's type theory. *BIT*, 24(3):288–301, October 1984.
- [8] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, (2):325–355, 1986.
- [9] Lawrence C. Paulson. *Natural Deduction Proof as Higher-Order Resolution*. technical report 82, Universtiy of Cambridge Computer Laboratory, Cambridge, 1985.
- [10] Kent Petersson. *A Programming System for Type Theory*. PMG Memo 21, Chalmers University of Technology, S-412 96 Göteborg, 1982.
- [11] E. Saaman and G. Malcolm. *Well-founded Recursion in Type theory*. Technical Report, Subfaculteit Wiskunde en Informatica, 1987.
- [12] Jan Smith. The identification of propositions and types in Martin-Löf's type theory. In *Foundations of Computation Theory, Proceedings of the Conference*, pages 445–456, 1983.