Programming in Constructive Set Theory: Some Examples

Bengt Nordström
Laboratory for Programming Methodology
Informationsbehandling
University of Göteborg and
Chalmers University of Technology
S-412 96 Göteborg, Sweden

Abstract

Per Martin-Löf's Constructive Set Theory is a mathematical language with computation rules. It is primarily designed to be a language for mathematical reasoning. The language has a very simple semantics and its rules have a simple structure. Since it is a language for constructive mathematics, it is possible to execute the proof (the construction of a proposition) as a program.

The language can be seen as a programming language without assignments and other side effects. Compared to traditional functional languages it has a very rich type structure in that the type of an expression can completely specify the task of the expression. A sorting algorithm, for instance, can be conventionally specified to have a type

$$\text{sort: List(A)} \rightarrow \text{List(A)}$$

which is type-correct if sort is any function taking a list as argument and producing a list as result. It is, however, also possible to specify that sort is a function taking a list as argument and producing a sorted permutation of its input as result, i.e.

$$\text{sort:}(\Pi x \in \text{List(A)})\ (\Sigma y \in \text{List(A)})\ (\text{Perm}(x,y) \times \text{Sorted}(y))$$

The type (or the task) of the program can be read as the proposition

$$(\forall x \in \text{List(A)})\ (\exists y \in \text{List(A)}\ (\text{Perm}(x,y)\ \&\ \text{Sorted}(y))$$

which is read "for all lists x, there is a sorted permutation y of x". We can prove that this proposition is true, using the rules of the language to construct a program for the task. If the proposition were not true, it would be impossible to find a program for it and we would have had an impossible task. The types of Constructive Set Theory can be seen as a specification language for the programs, but of course there is only one language, avoiding the complexity of mixing a programming language with a logical language.

The similarity (or rather: identity) between a mathematical proof of a given proposition and a program for a given task suggests that programming should be similar to the mathematicians activity of finding proofs. We have illustrated this with an example of how a proof of the Euclidean division theorem yields a program to compute the quotient and the remainder between two natural numbers.

The paper contains a description of the language. Since all programs in the language terminate, the proof rules and the semantics are simple. We give some examples of programming with lists and reasoning about the programs. We also define the Ackermann function.

Introduction

Constructive Set Theory is a mathematical language with computation rules developed by Per Martin-Löf [ML1, ML2]. The inference rules of the language explain how to form judgements from known judgements. A judgement of the form

$$a \in A$$

may be read in several ways:

1) a is an element of the set A.
2) a is a material proof (construction, manifestation of the proposition A.
3) a is a program for the task A.
4) a is a solution to the problem A.

In this context, we are most interested in the third reading, but we will also use the first two. The different readings imply that a proposition A is seen as the set of all proofs of A and that the specification of the task of a program is similar to a proposition. The programmer's activity of finding a program is therefore similar to the mathematicians activity of finding a proof of a proposition. Similar ideas have been expressed by Takasu[T] and Goto [G] who derive programs mechanically from constructive proofs.

There are two fundamental forms of judgements in Constructive Set Theory, "a ∈ A" and "A set".

This research was sponsored by Styrelsen för Teknisk utveckling and Naturvetenskapliga Forskningsrådet

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1981 ACM 0-89791-060-5/81-10/0141 $00.75

The first means that a is an element of A and the second that A is a set. If we are not interested in the construction a for A, i.e. we are only interested in the assertion that there is a construction for A, i.e. that A is true, then we will write

A true

instead of $a \in A$.

Another fundamental form of judgement is "$a = b \in A$" which means that a and b are equal elements of A.

The description of the language will proceed in the following way. There are different ways of constructing new sets from given sets. We call these rules of set formation. For each new set we will explain how to construct the canonical elements of the set. A canonical element is an element which has itself as value. We will call these rules introduction rules. The next rule is an elimination rule which from an arbitrary element p of the set explains how to construct a program for the task S(p), where S is a property (family of sets) over the set in question. This rule can be seen as a control structure in the language. Since the control structure is a non-canonical form of expression, there is an equality rule which is used to find the value of an expression of that form. The value of an expression p is a canonical expression which is equal to p.

We will start by explaining sets which are common as types in programming languages: finite sets, the natural numbers, function sets, cartesian products, disjoint unions and lists. We will then continue to describe two set forming operations which correspond to quantifiers in predicate logic, and finally we will explain well-orderings with transfinite induction which are essential for implementing tree like data structures (lists, natural numbers, abstract syntax trees etc.).

How expressions are formed

Expressions are used to express sets and elements. There are two different ways of writing an expression, either by using the concrete syntax or the proper syntax of the language. The proper syntax of an expression resembles what McCarthy [McC] calls the abstract syntax in that all parts of an expression are essential for its meaning, and that there is only one kind of separators and terminators which makes it easy to decompose the expression into its parts. It is different from the abstract syntax in that the proper syntax of an expression explains what variables become bound in different parts of the expression. The proper syntax is easier to use when talking about expressions (e.g. in proof rules, editors, interpreters), while the concrete syntax is easier to read and write since it allows an expression to be written in a more traditional way. The following table contains examples of expressions written in the two ways.

Concrete syntax	Proper syntax
a+b*c	plus(a,times(b,c))
$\sum_{i=1}^{n} e$,	sum(1,n,(i)e)
[sum,i=1,n]e	sum(1,n,(i)e)
$\lambda x.e$	$\lambda((x)e)$
$(\forall x \in A)B$	$\forall((x)B)$

Proper expressions

A proper expression is either a variable, an application or an abstraction.

An application is of the form

$$e(e_1,e_2,\ldots,e_n) \quad , n \geq 0,$$

where e is an operator and $e_1,e_2,\ldots,e_n$ are its operands. The operator and the operands are always expressions.

An abstraction is of the form

$$(x_1,x_2,\ldots x_n)e \quad , n \geq 1$$

where $x_1,\ldots x_n$ are variables and e is an expression.

If e is an expression depending on $x_1,x_2,\ldots,x_n$, then the abstraction $(x_1,\ldots x_n)e$ is an expression which does not depend on $x_1,\ldots x_n$. For instance, the expression $(x)x^2$ does not depend on x, it is the expression which squares an integer.

An expression e depends on the variable x when x is free in e. A variable is free in an expression in the following cases.

- The variable x is free in the expression x.
- The variable x is free in the expression $e(e_1,\ldots,e_n)$ if x is free in one of the expressions $e,e_1,\ldots,e_n$.
- The variable x is free in the expression $(x_1,\ldots,x_n)e$ if x is free in e and distinct from $x_1,\ldots,x_n$.

If g is an abstraction $(x_1,\ldots,x_n)e$ then the application $g(e_1,\ldots,e_n)$ is the expression e with $e_1,\ldots,e_n$ substituted for $x_1,\ldots,x_n$ in e, i.e.

$$((x_1,\ldots,x_n)e)(e_1,\ldots,e_n) \equiv e(e_1,\ldots,e_n/x_1,\ldots,x_n)$$

so that

$$((x_1,\ldots,x_n)e)(x_1,\ldots,x_n) \equiv e$$

Finite sets

If $i_1, i_2, \ldots, i_n$, $n \geq 0$ are identifiers then $\{i_1, \ldots, i_n\}$ is a set. The canonical elements of $\{i_1, \ldots, i_n\}$ are i_1 and i_2 and ... and i_n. The control structure associated with $\{i_1, \ldots, i_n\}$ is the case expression

case p of
 i_1: e_1,
 i_2: e_2,
 ⋮
endcase

which is a construction (program) for S(p) if S is a property[1] over $\{i_1,\ldots,i_n\}$, $p \in \{i_1,\ldots,i_n\}$ and $e_1 \in S(i_1)$, ..., $e_n \in S(i_n)$. The case expression is a non-canonical form of expression; its

1) That S is a property over a set A means that S(a) is a proposition (set) if $a \in A$. This means that S is an abstracted expression, which applied to a member of A yields a proposition (set). S can therefore be seen as a family of sets indexed by elements in A.

value is computed by the following rule:

1. Compute p. Since $p \in \{i_1, \ldots, i_n\}$, its value is of the form i_j, $j \leq n$.
2. The value of the case-expression is the value of e_j.

For instance, if Boolean ≡ {true, false}, then the definition

```
not(x)  ≡ case x of
            true: false,
            false: true
          endcase
```

corresponds to the informal definition

$$\begin{cases} \text{not(true)} = \text{false}, \\ \text{not(false)} = \text{true}. \end{cases}$$

We will use the convention of writing

<u>if</u> p <u>then</u> e <u>else</u> f

instead of <u>case</u> p <u>of</u> true: e, false: f <u>endcase</u>

The proof rules for the finite sets can be given by:

FF $\quad \{i_1, \ldots, i_n\}$ set

FI $\quad i_1 \in \{i_1, \ldots, i_n\} \quad \ldots \quad i_n \in \{i_1, \ldots, i_n\}$

FE $\quad \dfrac{p \in \{i_1, \ldots, i_n\} \quad e_1 \in S(i_1) \quad \ldots \quad e_n \in S(i_n)}{\text{case}(p, e_1, \ldots, e_n) \in S(p)}$

$$\frac{e_1 \in S(i_1) \quad \ldots \quad e_n \in S(i_n)}{\text{case}(i_j, e_1, \ldots, e_n) = e_j \in S(i_j)}$$

<u>Natural numbers</u>

N is a set. The canonical elements of N are 0 and succ(a) for $a \in N$. The control structure associated with the natural numbers is the rec-expression

```
rec p of
    0: d,
    succ(x): from z to b
endrec
```

which is a construction (program) for S(p) if $p \in N$, d is a construction for S(0) and b is a construction for S(succ(x)) under the assumption that $x \in N$ and that z is a construction for S(x). S is a property over N.

The proper syntax of the rec-expression is

rec(p,d,(x,z)b).

The rec-expression is a non-canonical form of expression; its value is computed by the following rule:

1. Compute p. Since $p \in N$, its canonical form is either 0 or succ(q), where $q \in N$.
2. If p's value is 0, then the value of the rec-expression is the value of d.
3. If p's value is succ(q) then find the value of b(q,r/x,z) where r is the expression

```
rec q of
    0: d
    succ(x): from z to b
endrec
```

For instance, the definition

```
fac(x) ≡ rec x of
           0: 1
           succ(y): from z to z*succ(y)
         endrec
```

corresponds to the informal definition:

$$\begin{cases} \text{fac}(0) = 1 \\ \text{fac}(y+1) = \text{fac}(y)*(y+1) \end{cases}$$

The proof rules for the natural numbers can be given by:

NF $\quad$ N set

NI $\quad 0 \in N \qquad \dfrac{a \in N}{\text{succ}(a) \in N}$

NE $\quad \dfrac{p \in N \quad d \in S(0) \quad \begin{matrix}(x \in N,\ z \in S(x)) \\ e(x,z) \in S(\text{succ}(x))\end{matrix}}{\text{rec}(p,d,e) \in S(p)}$

N= $\quad \dfrac{q \in N \quad d \in S(0) \quad \begin{matrix}(x \in N,\ z \in S(x)) \\ e(x,z) \in S(\text{succ}(x))\end{matrix}}{\begin{matrix}\text{rec}(0,d,e) = d \in S(0) \\ \text{rec}(\text{succ}(q),d,e) = e(q,\text{rec}(q,d,e)) \in S(\text{succ}(q))\end{matrix}}$

where the scheme

$$\frac{\begin{matrix}(P) \\ Q\end{matrix}}{R}$$

means that the conclusion R does not depend on the assumption P. The scheme

$$\begin{matrix} P \\ \overline{Q} \\ R \end{matrix}$$

is an abbreviation for the two schemes

$$\frac{P}{Q} \quad \frac{P}{R}.$$

A special case of the NE-rule is the following

$$\frac{p \in N \qquad S(0)\ \text{true} \qquad \begin{matrix}(x \in N,\ S(x)\ \text{true}) \\ S(succ(x))\ \text{true}\end{matrix}}{S(p)\ \text{true}}$$

which is the rule of mathematical induction: if we have proved that S(0) is true and that S(x+1) is true under the assumption that S(x) is true, then we may conclude that S(p) is true for an arbitrary natural number p.

Function set (a special case of the cartesian product between a family of sets)

If A and B are sets, then $A \to B$ (with proper syntax $\to (A,B)$) is a set. The canonical elements of $A \to B$ are of the form $\lambda x.b$ (with proper syntax $\lambda((x)b)$) where $b \in B$ under the assumption that $x \in A$. The control structure associated with $A \to B$ is functional application

apply(p,a)

which is a construction (program) for B if $a \in A$ and $p \in A \to B$.

Functional application is a non-canonical form of expression, its value is computed by the following rule.

1. Compute p. Since $p \in A \to B$ its value is of the form $\lambda x.b$.
2. Compute b(a/x), i.e. substitute a for x in b and find the canonical value of the resulting expression.
3. The value of apply(p,a) is the value of b(a/x)

Reading "$\lambda x.b \in A \to B$" as "$\lambda x.b$ is a material proof of the proposition $A \to B$" yields: "We may conclude that the proposition $A \to B$ is true if we have a method which constructs a proof of B from an arbitrary proof of A." This is the interpretation of $A \supset B$ (A implies B) in constructive mathematics.

In a more familiar notation, $A \to B$ is the function space B^A consisting of all functions mapping a member of A to a member of B.

The proof rules for $A \to B$ can be given by:

$$\to F \quad \frac{A\ \text{set} \quad B\ \text{set}}{A \to B\ \text{set}}$$

$$\to I \quad \frac{\begin{matrix}(x \in A) \\ e(x) \in B\end{matrix}}{\lambda(e) \in A \to B}$$

$$\to E \quad \frac{a \in A \quad p \in A \to B}{apply(p,a) \in B}$$

$$\to = \quad \frac{\begin{matrix}(x \in A) \\ e(x) \in B\end{matrix} \quad a \in A}{apply(\lambda(e),a) = e(a) \in B}$$

Ignoring the constructions in the introduction and elimination rules yields

$$\to I \quad \frac{\begin{matrix}(A\ \text{true}) \\ B\ \text{true}\end{matrix}}{A \quad B\ \text{true}}$$

$$\to E \quad \frac{A\ \text{true} \quad A \to B\ \text{true}}{B\ \text{true}}$$

which are the rules for implication in Gentzen's system of natural deduction.

Cartesian product of two sets (a special case of the disjoint union between a family of sets)

If A and B are sets, then $A \times B$ (with proper syntax $\times(A,B)$) is a set. The canonical elements of $A \times B$ are of the form (a,b) (with proper syntax pair(a,b)) where $a \in A$ and $b \in B$. The control structure associated with the cartesian product is the split-expression

split p into (x,y) in h

which is a construction (program) for S(p) if $p \in A \times B$ and e is a construction for S((x,y)) under the assumptions that $x \in A$, $y \in B$. The proper syntax for the split expression is split(p,(x,y)h).

The split expression is a non-canonical form of expression, its value is computed by the following rule

1. Compute p. Since $p \in A \times B$, its canonical form is a pair (a,b).
2. Compute h(a,b/x,y), i.e. find the canonical value of e after having substituted a for x and b for y.
3. The value of h(a,b/x,y) is the value of the split expression.

For instance, the definition

length(x) ≡ split x into (y,z) in sqrt(y×y+z×z)

corresponds to the informal definition

length((y,z)) ≡ sqrt(y×y+z×z).

Reading "$(a,b) \in A \times B$" as "(a,b) is a material proof of the proposition $A \times B$" together with its premises yields: "We may conclude that the proposition $A \times B$ is true if we have a proof of A and a proof of B." This is the interpretation of A & B (A and B) in constructive mathematics.

The proof rules for $A \times B$ can be given by:

$$\times F \quad \frac{A\ \text{set} \quad B\ \text{set}}{A \times B\ \text{set}}$$

$$\times I \quad \frac{a \in A \quad b \in B}{(a,b) \in A \times B}$$

$$\times E \quad \frac{p \in A \times B \quad \begin{matrix}(x \in A,\ y \in B) \\ e(x,y) \in S((x,y))\end{matrix}}{split(p,e) \in S(p)}$$

$$\times = \frac{a \in A \quad b \in B \quad \begin{matrix}(x \in A,\ y \in B)\\ e(x,y) \in S((x,y))\end{matrix}}{split((a,b),e) = e(a,b) \in S((a,b))}$$

By using p ↓ 1 for split(p,(x,y)x) and p ↓ 2 for split(p,(x,y)y) we get as special cases of the elimination rule:

$$\times E' \quad \frac{p \in A \times B}{p \downarrow 1 \in A} \qquad \frac{p \in A \times B}{p \downarrow 2 \in B}$$

Ignoring the constructions in xI and xE' yields

$$\times I \quad \frac{A \text{ true} \quad B \text{ true}}{A \times B \text{ true}}$$

$$\times E \quad \frac{A \times B \text{ true}}{A \text{ true}} \qquad \frac{A \times B \text{ true}}{B \text{ true}}$$

which are the rules for conjunction in Gentzen's system of natural deduction.

Disjoint union

If A and B are sets, then A+B (with proper syntax +(A,B)) is a set. The canonical elements of this set are of the forms i|a and j|b (with proper syntax i(a) and j(b)) where a ∈ A and b ∈ B. The control structure associated with the disjoint union set is the <u>when</u> - <u>expression</u>

```
when p is
       i|x: e,
       j|y: f
endwhen
```

which is a construction (program) for S(p) if p ∈ A+B and e is a construction for S(i|x) under the assumption that x ∈ A and f is a construction for S(j|y) under the assumption that y ∈ B. The proper syntax for the when-expression is when(p,(x)e,(y)f).

The when-expression is a non-canonical form of expression, its value is computed by the following rule:

1. Compute p. Since p ∈ A+B its value is either of the form i|a, where a ∈ A or of the form j|b, where b ∈ B.

2. If p's value is i|a, then in that case the value of the when-expression is the value of e(a/x).

3. If p's value is j|b, then in that case the value of the when-expression is f(b/y).

Reading "i|a ∈ A+B" and "j|b ∈ A+B" as "i|a and j|b is a material proof of A+B" yields "We may conclude that the proposition A+B is true if we have a proof of A or a proof of B." This is the interpretation of A ∨ B (A or B) in constructive mathematics.

The proof rules for A+B can be given by:

$$+F \quad \frac{A \text{ set} \quad B \text{ set}}{A+B \text{ set}}$$

$$+I \quad \frac{a \in A}{i|a \in A+B} \qquad \frac{b \in B}{j|b \in A+B}$$

$$+E \quad \frac{p \in A+B \quad \begin{matrix}(x \in A)\\ c(x) \in S(i|x)\end{matrix} \quad \begin{matrix}(y \in B)\\ d(y) \in S(j|y)\end{matrix}}{when(p,c,d) \in S(p)}$$

$$+= \quad \frac{a \in A \quad b \in B \quad \begin{matrix}(x \in A)\\ c(x) \in S(i|x)\end{matrix} \quad \begin{matrix}(y \in B)\\ d(y) \in S(j|y)\end{matrix}}{\begin{matrix}when\ (i|a,c,d) = c(a) \in S'(i|a)\\ when\ (j|b,c,d) = d(b) \in S(j|b)\end{matrix}}$$

Ignoring the constructions in +I and +E yields

$$+I \quad \frac{A \text{ true}}{A+B \text{ true}} \qquad \frac{B \text{ true}}{A+B \text{ true}}$$

$$+E \quad \frac{A+B \text{ true} \quad \begin{matrix}(A \text{ true})\\ S \text{ true}\end{matrix} \quad \begin{matrix}(B \text{ true})\\ S \text{ true}\end{matrix}}{S \text{ true}}$$

which are the rules for disjunction in Gentzen's system of natural deduction.

Lists

If A is a set , then List(A) is a set. The canonical elements of List(A) are nil and a;s where a ∈ A and s ∈ List(A). The proper syntax of a;s is ;(a,s). The control structure associated with List(A) is the listrec-expression

```
listrec p of
       nil:d,
       x;y: from z to e
endlistrec
```

which is a construction (program) for S(p) if p ∈ List(A), d is a construction for S(nil) and e is a construction for S(x;y) under the assumptions that x ∈ A, y ∈ List(A) and z is a construction for S(y). The proper syntax of the listrec-expression is

listrec(p,d,(x,y,z)e).

The list-ind expression is a non-canonical form of expression, its value is computed by the following rule:

1. Compute p. Since p ∈ List(A) its value is either nil or a;s where a ∈ A, s ∈ List(A).

2. If p's value is nil then the value of the

listrec-expression is the value of d.

3. If p's value is a;s, then the value of the listrec-expression is the value of e(a,s,r/x,y,z), r is the expression

```
listrec s of
    nil: d,
    x;y: from z to e
endlistrec
```

For instance, the definition

```
concat(x,y) ≡ listrec x of
                  nil: y
                  a;s: from p to a;p
              end
```

corresponds to the informal definition

$$\begin{cases} concat(nil,y) = y \\ concat(a;s,y) = a.concat(s,y). \end{cases}$$

The proof rules for List(A) can be given by:

$$LF \quad \frac{A\ set}{List(A)\ set}$$

$$LI \quad nil \in List(A) \qquad \frac{a \in A \qquad s \in List(A)}{a;s \in List(A)}$$

$$LE \quad \frac{p \in List(A) \qquad d \in S(nil) \qquad \begin{matrix}(x \in A, y \in List(A), \\ z \in S(y))\\ e(x,y,z) \in S(x;y)\end{matrix}}{listrec(p,d,e) \in S(p)}$$

$$L{=} \quad \frac{a \in A \qquad s \in List(A) \qquad d \in S(nil) \qquad \begin{matrix}\begin{pmatrix} x \in A \\ y \in List(A) \\ z \in S(y)\end{pmatrix}\\ e(x,y,z) \in S(x;y)\end{matrix}}{\begin{matrix} listrec(nil,d,e) = d \in S(nil) \\ listrec(a;s,d,e) = e(a,s,listrec(s,d,e)) \in S(a;s)\end{matrix}}$$

A special case of the list elimination rule is

$$\frac{p \in List(A) \qquad S(nil)\ true \qquad \begin{matrix}(x \in A, y \in List(A), \quad S(y)\ true)\\ S(x;y)\ true\end{matrix}}{S(p)\ true}$$

which is the rule for induction on lists: if we can prove that S(nil) is true and that S(x;y) is true under the assumption that S(y) is true then we may comclude that S(p) is true for an arbitrary p.

The product of a family of sets

If A is a set and if B(x) is a set under the assumption that $x \in A$ (i.e. B(x) is a family of sets indexed by $x \in A$, or B is a property of elements of A) then $(\prod x \in A)B(x)$ is a set. The canonical elements of $(\prod x \in A)B(x)$ are of the form $\lambda x.b$ (with proper syntax $\lambda((x)b)$) where b(x) is an element of B(x) under the assumption that $x \in A$. The control structure associated with $(\prod x \in A)B(x)$ is functional application

apply(p,a)

which is a construction (program) for B(a) if $a \in A$ and $p \in (\prod x \in A)B(x)$.

The computation rule for functional application has been given earlier in the description of $A \rightarrow B$.

Reading "$\lambda x.b \in (\prod x \in A)B(x)$" as "$\lambda x.b$ is a material proof of the proposition $(\prod x \in A)B(x)$" yields: "We may conclude that the proposition $(\prod x \in A)B(x)$ is true if we have proof of B(x) for an arbitrary element $x \in A$". This is the interpretation of $(\forall x \in A)B(x)$ in constructive mathematics.

If B(x) does not depend on x in $(\prod x \in A)B(x)$ then we get the set $A \rightarrow B$ previously described.

If A is a finite set, for instance $A = \{i_1, i_2, \ldots, i_n\}$ then $(\prod x \in A)B(x)$ corresponds exactly to the record type

```
record
    i1: T1,
    i2: T2
    ⋮
    in: Tn
end
```

provided $B(i_1) = T_1, B(i_2) = T_2, \ldots, B(i_n) = T_n$.

The proof rules for $(\prod x \in A)B(x)$ can be given by:

$$\prod F \quad \frac{A\ set \qquad \begin{matrix}(x \in A)\\ B(x)\ set\end{matrix}}{\prod(A,B)\ set}$$

$$\prod I \quad \frac{\begin{matrix}(x \in A)\\ b(x) \in B(x)\end{matrix}}{\lambda(b) \in \prod(A,B)}$$

$$\prod E \quad \frac{p \in \prod(A,B) \qquad a \in A}{apply(p,a) \in B(a)}$$

$$\prod {=} \quad \frac{\begin{matrix}(x \in A)\\ b(x) \in B(x)\end{matrix} \qquad a \in A}{apply(\lambda(b), a) = b(a) \in B(a)}$$

Ignoring some of the constructions in the introduction and elimination rules gives

$$\prod I \frac{\begin{matrix}(x \in A)\\ B(x) \text{ true}\end{matrix}}{(\prod x \in A)B(x) \text{ true}}$$

$$\prod E \frac{(\prod x \in A)B(x) \text{ true} \quad a \in A}{B(a) \text{ true}}$$

which should be compared with the rules for the universal quantifier in Gentzen's system of natural deduction:

$$\forall I \frac{B(x) \text{ true}}{(\forall x)B(x) \text{ true}}$$

$$\forall E \frac{(\forall x)B(x)}{B(a)}$$

The disjoint union of a family of sets

If A is a set and B(x) is a set under the assumption that x ∈ A, then (∑ x ∈ A)(B(x) (with proper syntax ∑(A,B))is a set. The canonical elements of (∑ x ∈ A)B(x) are of the form (a,b) (with proper syntax pair(a,b)) where a ∈ A and b ∈ B(a). The control structure associated with the disjoint union set is the split-expression

split p into (x,y) in h

which is a construction (program) for S(p) if p ∈ (∑ x ∈ A)B(x) and h is a construction for S((x,y)) under the assumptions that x ∈ A and y ∈ B(x). The proper syntax for the split-expression is split(p,(x,y)h).

The computation rule for the split-expression has been given earlier in the description of A × B.

Reading "(a,b) ∈ (∑ x ∈ A)B(x)" as "(a,b) is a material proof of the proposition (∑ x ∈ A)B(x)" yields: "We may conclude that the proposition (∑ x ∈ A)B(x) is true if we have an element x ∈ A and a proof of B(x)." This is the interpretation of (∃ x ∈ A)B(x) in constructive mathematics.

The proof rules for (∑x ∈ A)B(x) can be given by:

$$\sum F \frac{A \text{ set} \quad \begin{matrix}(x \in A)\\ B(x) \text{ set}\end{matrix}}{\sum(A,B) \text{ set}}$$

$$\sum I \frac{a \in A \quad b \in B(a)}{(a,b) \in \sum(A,B)}$$

$$\sum E \frac{p \in \sum(A,B) \quad \begin{matrix}(x \in A,\ y \in B(x))\\ e(x,y) \in S((x,y))\end{matrix}}{\text{split}(p,e) \in S(p)}$$

$$\sum = \frac{a \in A \quad b \in B(a) \quad \begin{matrix}(x \in A,\ y \in B(x))\\ e(x,y) \in S((x,y))\end{matrix}}{\text{split}((a,b),e) = e(a,b) \in S((a,b))}$$

Ignoring some of the constructions in ∑I and ∑E yields

$$\sum I \frac{a \in A \quad B(a) \text{ true}}{(\sum x \in A)B(x) \text{ true}}$$

$$\sum E \frac{(\sum x \in A)B(x) \text{ true} \quad \begin{matrix}(x \in A,\ B(x) \text{ true})\\ S \text{ true}\end{matrix}}{S \text{ true}}$$

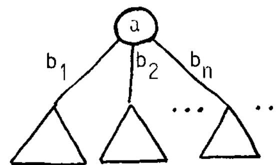
which should be compared with the rules for the existential quantifier in Gentzen's system of natural deduction

$$\exists I \frac{B(a) \text{ true}}{(\exists x)B(x) \text{ true}}$$

$$\exists E \frac{(\exists x)B(x) \text{ true} \quad \begin{matrix}(B(x) \text{ true})\\ S \text{ true}\end{matrix}}{S \text{ true}}$$

Wellorderings

If A is a set and B(x) is a set under the assumption that x ∈ A, then (Wx ∈ A)B(x) (with proper syntax W(A,B)) is a set. The canonical elements of (Wx ∈ A)B(x) are of the form sup(a,f) where a ∈ A and f(y) ∈ (Wx ∈ A)B(x) under the assumption that y ∈ B(a). We can look upon "sup(a,f)" as a tree with a root labeled "a" and branches labeled "b_1", "b_2", ... where b_1 ∈ B(a), b_2 ∈ B(a), ...



The branches lead to subtrees which are build up in the same way. The control structure associated with (Wx ∈ A)B(x) is transfinite recursion

transrec p of
  sup(x,u):from v to d(x,u,v)
endtransrec

which is a construction (program) for S(p) if $p \in (Wx \in A)B(x)$ and $d(x,u,v)$ is a construction for $S(sup(x,u))$ under the assumptions that $x \in A$, $u(z) \in (Wx \in A)B(x)$ for $z \in B(x)$ and $v(z) \in S(u(z))$ for $z \in B(x)$. The proper syntax for transfinite recursion is transrec(p,d).

The computation rule for transfinite recursion is:

1. Evaluate p. Since $p \in (Wx \in A)B(x)$, p's value is of the form "sup(a,f)".

2. The value of the transfinite recursion is then the value of d(a,f, (z)transrec(f(z),d)).

The rules for the wellorderings can be given by:

$$\text{WF}\ \frac{A\ \text{set} \quad \begin{matrix}(x \in A)\\ B(x)\ \text{set}\end{matrix}}{W(A,B)\ \text{set}}$$

$$\text{WI}\ \frac{a \in A \quad \begin{matrix}(y \in B(a))\\ f(y) \in W(A,B)\end{matrix}}{sup(a,f) \in W(A,B)}$$

$$\text{WE}\ \frac{p \in W(A,B) \quad \begin{matrix}\begin{pmatrix}(x \in A, u(z) \in W(A,B)(z \in B(x)),\\ v(z) \in S(u(z))(z \in B(x))\end{pmatrix}\\ d(x,u,v) \in S(sup(x,u))\end{matrix}}{transrec(p,d) \in S(p)}$$

$$W =\ \frac{a \in A \quad \begin{matrix}(y \in B(a))\\ f(y) \in W(A,B)\end{matrix} \quad \begin{matrix}\begin{pmatrix}x \in A,\\ u(z) \in W(A,B)\ (z \in B(x)),\\ v(z) \in S(u(z))(z \in B(x))\end{pmatrix}\\ d(x,u,v) \in S(sup(x,u))\end{matrix}}{\begin{matrix}transrec(sup(a,f),d)\\ = d(a,f,(z)transrec(f(z),d)) \in S(sup(a,f))\end{matrix}}$$

The equality proposition

The proposition which corresponds to the judgement $a = b \in A$ is Eq(A,a,b). So if we have a proof of the proposition Eq(A,a,b) then we may make the judgement $a = b \in A$ and if we have made the judgement $a = b \in A$ then we may conclude there is an element in the set Eq(A,a,b). We call this element e. To summarize:

$$\text{EqI}\ \frac{a = b \in A}{e \in Eq(A,a,b)}$$

$$\text{EqE}\ \frac{p \in Eq(A,a,b)}{a = b \in A}$$

$$\text{Eq=}\ \frac{p \in Eq(A,a,b)}{p = e \in Eq(A,a,b)}$$

A simple example of the derivation of a program

The task is to find a proof of the Euclidean division theorem (called E)

$$(\forall a,b \in N)([b>0] \rightarrow \exists(q,r \in N)([r<b]\&[a=b*q+r]))$$

which says that for all natural numbers a and b where b>0, there exists natural numbers q and r such that r<b and a=b*q+r.

We have used the following definitions:

$(\forall a,b \in N)B \equiv (\forall a \in N)(\forall b \in N)B$

$[a>b] \equiv [b<a]$

$[b<a] \equiv (\exists x \in N)[b+succ(x)=a]$

$[a=b] \equiv Eq(N,a,b)$

$(\exists a,b \in N)B \equiv (\exists a \in N)(\exists b \in N)B$

The proof will give us a program for the theorem where

$$c = \lambda a.\lambda b.\lambda o.(q(a,b),(r(a,b),(u(a,b,o),v(a,b,o)))) \in E$$

where a and b are natural numbers,

o	is a construction for [b>0],
q(a,b)	is the quotient between a and b,
r(a,b)	is the remainder,
u(a,b,o)	is a construction for r<b and
v(a,b,o)	is a construction for [b+succ(x)=a].

Proof: Assume $a,\ b \in N,\ o \in [b>0]$.
Let us do an induction over a.

Define $G(x) \equiv (\exists q,r \in N)([r<b]\&[x=b*q+r])$

Basis: $(0,(0,.)) \in G(0)$

since [0<b]&[0=b*0+0] is true.

("." denotes a construction which we do not care about.)

Induction step:

Assume that $z \in G(x)$ and define $q(z) \equiv z{\downarrow}1$, $r(z) \equiv z{\downarrow}2{\downarrow}1$. We want to find a construction for G(succ(x)). From the assumption, by repeated $\sum$-elimination, we get $z{\downarrow}2{\downarrow}2{\downarrow}1 \in [r(z)<b]$ which means that we can find a construction $p(z,b) \in [r(z)+1<b \vee [r(z)+1=b]$.

We can now make an $\vee$-elimination:
Assume that [r(z)+1<b] is true.

$$z{\downarrow}2{\downarrow}2{\downarrow}2 \in [x=b*q(z)+r(z)]$$

$$x = b*q(z)+r(z) \in N \qquad \text{(EqE)}$$

$$x+1 = b*q(z)+(r(z)+1) \in N \qquad \text{(Arithmetics)}$$

$$(q(z),(r(z)+1,.)) \in G(x+1) \qquad (\textstyle\sum E)$$

Assume that [r(z)+1=b] is true.

```
z↓2↓2↓2 ∈ [x = b*q(z)+r(z)]
          x = b*q(z)+r(z) ∈ N     (EqE)
        x+1 = b*q(z)+(r(z)+1) ∈ N
        x+1 = b*(q(z)+1)+0 ∈ N    (Arithmetics)
 (q(z)+1,(r(z),.)) ∈ G(x+1)
```

We can conclude that

```
when p(z,b) is
   i|x:(q(z),(r(z)+1,.))
   j|x:(q(z)+1,(r(z),.))
endwhen  is a construction for  G(x+1)
                                by + elimination
```

N-elimination gives us:

```
rec a of
      0:(0,(0,.))
   succ(x):from z to
           when p(z,b) is
              i|x:(q(z),(r(z)+1,.))
              j|x:(q(z)+1,(r(z),.))
           endwhen
end is a construction for G(a)
```

Three $\prod$-introductions yield that

```
λa.λb.λo. rec a of
               0:(0,(0,.))
          succ(x):from z to
                  when p(z,b) is
                     i|x:(q(z),(r(z)+1,.))
                     j|x:(q(z)+1,(r(z),.))
                  endwhen
          end
```

is a construction for the theorem, where p(z,b) is a construction for [r(z)+1<b]v[r(z)+1=b]

and q(z) ≡ z↓1

and r(z) ≡ z↓2↓1

What remains to be done is to find a proof p of [r(z)+1<b]v[r(z)+1=b]. It is not difficult to prove that for each proposition P built up by conjunctions and disjunctions of arithmetic equalities and inequalities there is an expression p' ∈ Bool which is such that

p'= true ∈ Bool iff p is true

We can now use the fact that there is a boolean function a<b (without brackets) which is defined such that

a<b = true ∈ Bool iff [a<b] is true.

to obtain the following program

```
λa.λb.λo. rec a of
               0:(0,(0,.))
     succ(x):from z to
               if r(z)+1<b then (q(z),(r(z)+1,.))
                           else (q(z)+1,(r(z),.))
     end
```

which is a construction for the theorem E.

Programming with lists

The definition of concatenation between lists was given earlier,

```
concat(x,y) ≡ listrec x of
                  nil: y
                  a;s: from p to a.p
              end
```

In order to give a flavour of how to reason about programs we will give a formal proof that concatenation is associative. Instead of writing concat(x,y) we will write x◇y, and instead of nil we will write ∅ .

Theorem: (x◇y)◇z = x◇(y◇z) ∈ List(A)
if x,y,z ∈ List(A), where A is a type.

Informal proof:

We make a list-induction on x, and abbreviate Eq(List(A),x,y) to x = y

Basis: By List= (∅◇y)◇z = y◇z, and
(∅◇(y◇z) = y◇z.

Induction step:

Assume that (s◇y)◇z = s◇(y◇z).

```
(a.s◇y)◇z = (a.(s◇y))◇z       by List=
          = a.((s◇y)◇z)       by List=
          = a.(s◇(y◇(y◇z))    by the induc-
                               tion assumption.
(a.s◇(y◇z) = a.(a◇(y◇z))      by List=
```

A formal proof

We define L(A) ≡ List(A). The formal proof of the theorem is the following proof tree:

```
                                               [(s◇y)◇z = s◇(y◇z)]
                  P1                                  P2
x∈L(A)   [(∅◇y)◇z = ∅◇(y◇z)]   [(a.s◇y)◇z = a.s◇(y◇z)LE
──────────────────────────────────────────────────────────────────
                      [(x◇y)◇z = x◇(y◇z)]
                      (x◇y)◇ z = x◇(y◇z) ∈ L(A)
```

where P_1 stands for the proof

$$\cfrac{\cfrac{\cfrac{\cfrac{y\in L(A)}{\emptyset\diamond y=y}\,LE \quad \cfrac{(x\in L(A))}{x\diamond z=x\diamond z}\,R}{(\emptyset\diamond y)\diamond z=y\diamond z}\,S \qquad \cfrac{\cfrac{(y,z\in L(A)}{y\diamond z=y\diamond z}\,Refl \quad \cfrac{(x\in L(A))}{x=\emptyset\diamond x}\,L}{y\diamond z=\emptyset\diamond(y\diamond z)}\,S}{(\emptyset\diamond y)\diamond z=\emptyset\diamond(y\diamond z)}\,Trans}{[(\emptyset\diamond y)\diamond z=\emptyset\diamond(y\diamond z)]}\,EqI$$

and P_2 stands for the proof

$$\cfrac{\cfrac{\begin{matrix}P_{21}\\(a.s\diamond y)\diamond z=a.(a\diamond(y\diamond z))\end{matrix} \qquad \cfrac{\cfrac{y,z\in L(A)}{y\diamond z=y\diamond z} \quad \cfrac{a\in A,s,x\in L(A)}{a.(s\diamond x)=(a.s)\diamond x}\,L=}{a.(s\diamond(y\diamond z))=a.s\diamond(y\diamond z)}\,S}{(a.s\diamond y)\diamond z=a.s\diamond(y\diamond z)}\,Trans}{[(a.s\diamond y)\diamond z=a.s\diamond(y\diamond z)]}\,EqI$$

where P_{21} is the proof

$$\cfrac{\begin{matrix}P_{211}\\(a.s\diamond y)\diamond z=a.((s\diamond y)\diamond z)\end{matrix} \qquad \cfrac{\cfrac{(x\in L(A))}{a.x=a.x}\,R \quad \cfrac{([(s\diamond y)\diamond z=s\diamond(y\diamond z)])}{(s\diamond y)\diamond z=s\diamond(y\diamond z)}\,IdE}{a.((s\diamond y)\diamond z)=a.(s\diamond(y\diamond z))}\,S}{(a.s\diamond y)\diamond z=a.(s\diamond(y\diamond z))}$$

where P_{211} is the proof

$$\cfrac{\cfrac{\cfrac{x,z\in L(A)}{x\diamond z=x\diamond z}\,R \quad \cfrac{a\in A,\ s,y\in L(A)}{a.s\diamond y=a.(s\diamond y)}\,=}{(a.s\diamond y)\diamond z=a.(s\diamond y)\diamond z} \qquad \cfrac{\cfrac{s,y\in L(A)}{s\diamond y=s\diamond y}\,R \quad \cfrac{a\in A,x\in L(A)}{a.x\diamond z=a.(x\diamond z)}\,=}{a.(s\diamond y)\diamond z=a.((s\diamond y)\diamond z)}}{(a.s\diamond y)\diamond z=a.((s\diamond y)\diamond z)}\,T$$

This completes the formal proof. It is not our suggestion that programmers should always give formal computer checked proofs of their programs. Formal proofs can however be used to optimize programs in the way shown by Goad [Gd].

The Ackermann-function

The Ackermann function is the classical example of a recursive function which is not primitive recursive, i.e. a function which cannot be defined using composition of functions and the schemata

$$f(x_1,\dots,x_n,0)=d(x_1,\dots,x_n)$$
$$f(x_1,\dots,x_n,x+1)=e(x_1,\dots,x_n,x,f(x_1,\dots,x_n,x))$$

which corresponds to the form rec in Constructive Set Theory.

This exercise will show how to express the Ackermann function in Constructive Set Theory.

Consider the following definitions

$f_0(x,y) \equiv y+x \equiv$ rec y of
0: x
succ(n): from z to succ(z) end

$f_1(x,y) \equiv x\cdot y \equiv$ rec y of
0: 0
succ(n): from z to x+z end

$f_2(x,y) \equiv x^y \equiv$ rec y of
0: 1
succ(n): from z to x*z end

$f_3(x,y) \equiv x\uparrow\uparrow y \equiv$ rec y of
0: 1
succ(n): from z to x^z end

These functions are the first in a series of functions f_1, f_2,... where $f_{n+1}(x,y)$ is the result of

$$f_n(x,\dots f_n(x,x)\dots)) \qquad \text{if } y>0$$

where f_n is applied y-1 times. The function $A(n,x,y) \equiv f_n(x,y)$ is the original Ackerman function and is defined by the following equalities:

$$\begin{cases} A(0,x,0)=x \\ A(0,x,y+1)=A(0,x,y)+1 \\ A(1,x,0)=0 \\ A(n+2,x,0)=1 \\ A(n+1,x,y+1)=A(n,x,A(n+1,x,y)) \end{cases}$$

We assume for simplicity that $y>0$. The first two equalities define $f_0(x,y)=x+y$ and the fifth equality defines f_{n+1} in terms of f_n.

If we define the function $do(n,f,x)\in N(n\in N,\ f\in N\times N\in N)$ such that

$$do(n,f,x) = apply(f,(\dots apply(f,(x,x))\dots))))$$

with n applications, then we can define the Ackermann function after having noted the equalities

```
A(0,x,y) = x+y
A(n+1,x,0) = if iszero(n)  then 0 else 1
A(n+1,x,y+1) = do(y,λ(y,z). A(n,y,z),x)
```

where we have used the abbreviation λ(y,z).e for λx.split x into (y,z) in e. For f(n) defined by

```
f(n) ≡ λ(x,y). A(n,x,y)
```

the following equalities hold

```
f(0) = λ(x,y).x+y
f(n+1) = λ(x,y). if y > 0 then do (y-1,f(n),x)
                 else if iszero(n)
                      then 0
                      else 1
```

This means that we can define the Ackermann function by

```
A(n,x,y) ≡ apply(rec n of
                  0:λ(x',y').x'+y'
                  succ(n):from z to
                          λ(x',y').
                           if y'> 0
                           then do(y'-1,z,x')
                           else if iszero(n)
                                then 0 else 1,
              (x,y))
```

What remains to be done is to define the function do which is such that

$$\begin{cases} do(0,f,x) = x \in N \\ do(n+1,f,x) = apply(f,(x,do(n,f,x))) \end{cases}$$

These equalities are solved by putting

```
do(n,f,x) ≡ rec n of
            0: x
            succ(n): from z to
                          apply(f,(x,z))
            end
```

Finally, we prove the following theorem.

Theorem: The function A(n,x,y) as defined above is a solution to the Ackermann equations.

Proof: We prove the equalities by substituting A into the three equations

i) $A(0,x,0) = x+0 = x$

ii) $A(0,x,y+1) = x+(y+1) = (x+y)+1 = A(0,x,y)+1$

These two cases have been proven by N-equality, $\prod$-elimination and elementary properties of +.

```
iii) If f(n)≡ λ(x,y).A(n,x,y) then
  f(n)=rec(n,
           λ(x,y).x+y,
           (n,z) λ(x,y).
              if y>0
               then do(y-1,z,x)
               else rec(n,0,1)
           )
  and
  A(1,x,0)=apply(λ(x,y).
                   if y>0
                    then do(y-1,f(0),x)
                    else rec(0,0,1),
                 (x,0))            by N-equality
         =rec(0,0,1)               by Π-equality
         =0                        by N-equality

iv)
  A(n+2,x,0)=apply(λ(x,y).
                     if y>0
                      then do(y-1,f(n+1),x)
                      else rec(n+1,0,1),
                   (x,0))          by N-equality
          =rec(n+1,0,1)            by Π-equality
          =0                       by N-equality

v)
  A(n+1,x,y+1)=apply( λ(x,y).
                        if y>0
                         then do(y-1,f(n),x)
                         else rec(n,0,1),
                      (x,y+1))     by N-equality
            =do(y,f(n),x)          by Π-equality
```

Now since y>0, y = m+1 ∈ N for some m. This means that

```
A(n+1,x,y+1) = do(m+1,f(n),x) ∈ N by substitution
             = apply(f(n),(x,do(m,f(n),x) ∈ N
                                   by N-equality
             = apply(f(n),(x,A(n+1,x,m+1) ∈ N
                                   by N = and ∏=
             = A(n,x,A(n+1,x,y)) ∈ N by substi-
                                          tution
```

The summation operator and a more general iterator

We will show how to define the traditional summation operator $\sum_{i=1}^{h} e$ where e is an expression depending on i. We will use the proper syntax

```
sum(1,h,e)
```

where e is an abstraction, i.e. we could also write

```
sum(1,h,(i)g)
```

if g is an expression depending on the variable i. The definition is straightforward when we have observed the facts that

$$(1)\quad \sum_{i=1}^{h} e(i) = \sum_{i=0}^{h-1} e(i+1)$$

$$(2)\quad \sum_{i=0}^{n+1} e(i) = \sum_{i=0}^{n} e(i) + e(n+1)$$

$$(3)\quad \sum_{i=0}^{0} e(i) = e(0)$$

The first fact implies that we should do a primitive recursion over h-1:

```
sum(1,h,e) ≡ rec h-1 of
                 0: e(1)
                 succ(x): from p to p+e(succ(x)+1)
                 end
```

This definition is an instance of a general scheme for applying a function repeatedly over an abstracted expression. First we need some definitions for functional application and composition.

Instead of f(a) we will write a→f and use the convention that a→b→c≡(a→b)→c. For functional composition

fog(x) ≡ f(g(x))

we also use the notation

g|f ≡ fog

We could define the multiple composition f|f|... |f with n compositions by the equalities

$$\begin{cases} f^0 = (x)x \\ f^{n+1} = f^n|f \end{cases}$$

i.e.

```
fⁿ===↑(f,n) ≡ (x)rec(n of
                  0:x,
                  succ(x): from p to p|f
                 endrec
```

We use a===b to mean "the concrete syntax for the proper expression b is a".

If we can find a construct $|_{i=1}^{n} f(i)$ which evaluates to

f(1) f(2)...f(n)

under the assumption that f(i)(x) ∈ A,(i ∈ N,x ∈ A) we obtain the summation expression
e(1)+...+e(n) by $0 \to |_{i=1}^{n}(x)x+e(i)$

For $|_{i=1}^{h} f(i)$ we use the proper syntax |(1,h,f) and we note the following equalities

$$|_{i=1}^{h} f(i) = |_{i=1}^{h-1+1} f(i+1+1)$$

$$|_{i=1}^{0} f(i) = (x)x$$

$$|_{i=1}^{n+1} f(i) = (|_{i=1}^{n} f(i))|f(n+1)$$

which gives the following definition

```
|ʰᵢ₌₁f(i)===|(1,h,f) ≡ (x)rec h-1+1 of
                          0: x,
                          succ(n): from p to
                                   p|f(n+1)
                         endrec
```

Iterators for lists

Composition iterator for lists

The definition of $|_{i=1}^{n} f(i)$ which is defined such that

$$|_{i=1}^{n} f(i) = f(1)|f(2)|...|f(n)$$

if 1,n∈Nat, f(i)(x) ∈ A for i ∈ N, x ∈ A

can be generalized to lists by the following construct

for i in s do $f(i) = f(s_1)|f(s_2)|...|f(s_n)$

if $s = s_1.s_2. ...s_n$.nil

We make the following definition

```
for i in s do f(i)od===for(i,s,f) ≡
                      (x)list-ind s in
                               nil:  x
                               a.t: from p to
                                        f(a)|p
                      end
```

For example to sum all elements in a list l of integers we write

0→for i in l do(x)x+i od

and to check if the number of elements in each element in a list of lists are less than 40 we write

true→for i in l do (x)(x and # i < 40)od

The following theorem is a link between the list iterator and the composition iterator.

<u>Theorem</u>: If $i,n \in Nat$, $x \in A, f(i)(x) \in A$ then

$$|_{i=1}^{n} f(i) = \underline{for}\ i\ \underline{in}\ 1...n\ do\ f(i) \in A \to A$$

where we have used the definitions

$$i..j \equiv nil \to |_{k=i}^{j}(x)cons(i+j-k,x)$$

$$cons(x,y) \equiv x;y$$

so, for instance

$$1...5 = 1;2;3;4;5;nil$$

<u>Proof</u>: Set $F(j,m) \equiv \underline{for}\ i\ in\ j...m\ \underline{do}\ f(i)$

We prove the theorem by an induction over n

I The theorem is true for n = 0 since

$$LHS = |_{i=1}^{0} f(i) = (x)x$$

$$RHS = for\ i\ in\ 1...0\ do\ f(i)$$
$$= for\ i\ in\ nil\ do\ f(i) = (x)x$$

II Assume that $|_{i=1}^{m} f(i) = F(1,m)$

We want to show that $|_{i=1}^{m+1} f(i) = F(1,m+1)$

$$|_{i=1}^{m+1} f(i) = |_{i=1}^{m} f(i)|f(m+1) \text{ by N } =$$

$= F(1,m)|f(m+1)$ by the induction assumption

$= F(1,m+1)$ by the following lemma.

<u>Lemma</u>: $F(j,m)|f(m+1) = F(j,m+1)$

<u>Proof</u>: We know that $F(j,m+1) = f(j)|F(j+1,m+1)$ since

$F(j,m+1) = \underline{for}\ i\ \underline{in}\ j...m+1\ do\ f(i)$

$= \underline{for}\ i\ \underline{in}\ nil \to |_{i=j}^{m+1}(x)cons(j+m+1-i,x)\ \underline{do}\ f(i)$

$= \underline{for}\ i\ \underline{in}\ cons(j,nil) \to |_{i=j+1}^{m+1}(x)cons(j+m+2-i,x))\ \underline{do}\ f(i)$

$= f(j)|F(j+1,m+1)$

We prove the lemma by an induction over m-j.

I The lemma is true for m-j = 0 (follows from the first line in the proof.)

II Assume that $F(j,m)|f(m+1) = F(j,m+1)$ holds when m-j = n. We want to show that the lemma is true when m-j = n+1

$LHS = F(j,m)|f(m+1) = (f(j)|F(j+1,m)|f(m+1)$

$= f(j)|(F(j+1,m)|f(m+1))$ by associativity of $|$.

$= f(j)|F(j+1,m+1)$ by the induction hypothesis

The composition iterator for lists resembles the reduce operator in APL.

Implementation

We have implemented an interpreter for Constructive Set Theory written in Lisp. It runs under Unix on a Vax computer.

Acknowledgements

I would like to thank Per Martin-Löf för many stimulating discussions and also the members of the laboratory for contributing to a nice working environment.

References

[ML1] Per Martin-Löf: "An Intuitionistic Theory of Types: Predicative Part", Logic Colloquium '73, ed. Rose, Shepherdson, North-Holland, Amsterdam, 1975, pp. 73-118.

[ML2] Per Martin-Löf: "Constructive Mathematics and Computer programming", Dept of Math, University of Stockholm, 113 85 Stockholm, Sweden, read at the 6:th International Congress for Logic, Methodology of Science, Hannover 1979.

[T] Takasu: Proofs and Programs, The Third IBM Symposium on Mathematical Foundation of Computer Science, Aug. 1978.

[G] Goto: Program Synthesis from Natural Deduction Proofs, IJCAI 1979, Tokyo.

[McC] John McCarthy: "A formal description of a subset of Algol 60", in Formal Language Description Languages, ed. Steel, North-Holland.

[P] Dag Prawitz: "Natural Deduction, a Proof Theoretical Study", Almqvist & Wiksell, Stockholm 1965.

[Gd] Christopher Goad: "Computational Uses of the Manipulation of Formal Proofs". (Thesis). Standard Report CS-80-819, 1980.