A Set Constructor for Inductive Sets in Martin-Löf's Type Theory

Kent Petersson* Dan Synek*

1 Introduction

An important construction in programming languages is the definition facility for inductive data types [Hoa75]. One way to understand this construction is as a very general type constructor for trees. Trees are used for many purposes in computer science, one important example is syntax trees for representing phrases in languages. It is therefore vital that such a type constructor, together with its proof rules, should be available in a programming logic such as Martin-Löf's intuitionistic type theory [ML82, ML84]. In this paper, we will define a set constructor that one could use for defining many inductive data types.

The rest of the paper is organized as follows: We first explain the wellorder set constructor W introduced by Martin-Löf in [ML82]. This set constructor is the least solution of a particular parameterized fixed point set equation. And since the parameters of this equation are closely related to the different parts of a single ML datatype definition [Mil84], we continue to discuss the correspondence between the wellorder constructor in Type Theory and the datatype constructor in ML.

The wellorder constructor is not sufficient when we want to define a family of mutually dependent inductive sets. We therefore introduce a set constructor for such sets. This set constructor, Tree, is explained in the following section and its formal rules are given.

*Programming Methodology Group, Dept. of Computer Sciences, Univ. of Göteborg and Chalmers, S-412 96 Göteborg, Sweden.

In section 3, we show that all wellorders could be defined by the tree set constructor and then we discuss a slight variant of the tree constructor. Finally some examples are presented.

1.1 An introduction to the wellorder set constructor

With the wellorder set constructor we can construct many different sets of trees and to characterize a particular set we must provide information about two things,

- the different ways the trees may be formed, and
- for each way to form a tree which parts it consists of.

To provide this information, the wellorder set constructor W has two arguments:

1. The *constructor set* B.
2. The *index family* C.

Given a constructor set B and an index family C over B, we can form a wellorder $\mathsf{W}(B, C)$ (two other notations are $(\mathsf{W}x : B)\,C(x)$ and $\mathsf{W}_{x:B}C(x)$).

One way of understanding the wellorder set constructor is to see it as a solution of an inductive definition of a particular form. So instead of solving a fixed point set equation every time we want to introduce a new set, as we do in domain theory [Sco82] we look at one particular equation of a general form and parameterize it with the information that differ between different inductive definitions. Since we just have one equation, we do not need to introduce the notions of fixed point set constructor, set operator and strictly positive set operator into the theory. Dybjer has furthermore shown [Dyb87] that all sets defined by a fixed point set operator could, in an extensional version of type theory, be represented by a wellorder.

The set $\mathsf{W}(B, C)$ is a solution to the equation

$$X \cong \sum_{y:B} C(y) \to X \tag{1}$$

In other words, isomorphic to the least fixed point of the strictly positive set operator

$$(X) \sum_{y:B} C(y) \to X$$

So $\mathsf{W}(B,C)$ is a fixed point to a *particular* (strictly positive) set operator.

If we have an element b_1 in the set B, that is, if we have a particular form we want the tree to have, and if we have a function c_1 from $C(b_1)$ to $\mathsf{W}(B,C)$, that is if we have a collection of subtrees, we may form the tree $\mathsf{sup}(b_1,c_1)$. We visualize this element in figure 1.

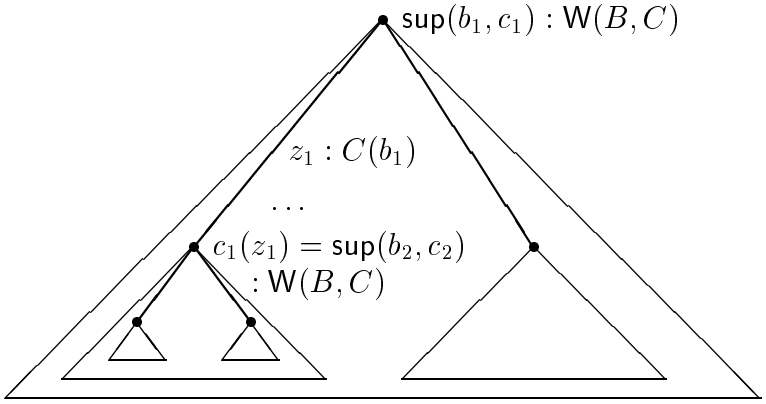

Figure 1: An element of a wellorder

We now give the formal rules for the wellorder set constructor and then an informal comparison between sets introduced using the wellorder set constructor and ML datatypes.

W - formation

$$\frac{B : \text{ set} \qquad C(y) : \text{ set } [y : B]}{\mathsf{W}(B,C) : \text{ set}}$$

W - introduction

$$\frac{b : B \qquad c(z) : \mathsf{W}(B,C)\ [z : C(b)]}{\mathsf{sup}(b,c) : \mathsf{W}(B,C)}$$

The non-canonical constant wrec is computed according to the computation rule:

$$\mathsf{wrec}(\mathsf{sup}(b,c),f) \ \rightarrow \ f(b,c,(x)\mathsf{wrec}(c(x),f))$$

The computation rule for wrec justifies the elimination and equality rules:

W - elimination

$$\frac{\begin{array}{l} D(x):\ \text{set}\ \ [x:\mathsf{W}(B,C)] \\ w:\mathsf{W}(B,C) \\ f(y,z,u):D(\mathsf{sup}(y,z)) \\ \qquad [y:B, z(v):\mathsf{W}(B,C)\ [v:C(y)], u(v):D(z(v))\ [v:C(y)]] \end{array}}{\mathsf{wrec}(w,f):D(w)}$$

W - equality

$$\frac{\begin{array}{l} D(x):\ \text{set}\ \ [x:\mathsf{W}(B,C)] \\ b:B \\ c(y):\mathsf{W}(B,C)\ \ [y:C(b)] \\ f(y,z,u):D(\mathsf{sup}(y,z)) \\ \qquad [y:B, z(v):\mathsf{W}(B,C)\ [v:C(y)], u(v):D(z(v))\ [v:C(y)]] \end{array}}{\mathsf{wrec}(\mathsf{sup}(b,c),f) = f(b,c,(x)\mathsf{wrec}(c(x),f):D(\mathsf{sup}(b,c))}$$

Let us see how we can represent the set introduced by an ML datatype definition of the form[1]

$$\textbf{datatype}\ A = b_1\ \textbf{of}\ C_1\ \ |\ \cdots\ |\ b_n\ \textbf{of}\ C_n$$

First, we define a set B containing the constructor names $b_1,\ldots,b_n$. Then we would like to construct a family of sets C over B in such a way that $C(b_i)$ expresses the information in the ML type C_i. Let us for a moment restrict the form of C_i to a cartesian product $\underbrace{A*\cdots*A}_{n}$. But instead of using a cartesian product where the elements are decomposed by selectors (projection functions), we use a function from a set of selector names to A. That is,

[1]Note the similarity between the ML definition

$$A = b_1\ \textbf{of}\ \{c_{b_1 1},\ldots,c_{b_1 n_{b_1}}\}\to A\ |\ \cdots\ |\ b_m\ \textbf{of}\ \{c_{b_m 1},\ldots,c_{b_m n_{b_m}}\}\to A$$

and the equation

$$A\ \ \cong\ \sum_{y:\{b_1,\ldots,b_m\}} \{c_{y1},\ldots,c_{yn_y}\}\to A$$

where $\sum$ is the *disjoint sum* of a family of sets and | is ML:s "sum" operator.

instead of A^n we use $\{c_1, \ldots, c_n\} \to A$. We can now, for each constructor, express the information in the ML type C_i as the set $C(b_i) = \{c_{b_i 1}, \ldots, c_{b_i n_{b_i}}\}$. Then we can construct the wellorder $\mathsf{W}(B, C)$, which is a set in Type Theory that represents the ML type.

Let us give some examples. An ML datatype definition such as

$$\textbf{datatype } \mathit{BinTree} = \mathit{leaf} \mid \mathit{node} \textbf{ of } \mathit{BinTree} * \mathit{BinTree}$$

will be represented as follows. The set of constructors is $\{\mathit{leaf}, \mathit{node}\}$. There is no selector for the constructor *leaf* because it has no components, and the set of selectors for *node* is a two element set, for example $\{\mathit{left}, \mathit{right}\}$ where the two elements give names to the two components. Putting this together, we can see that the wellorder[2]

$$\mathit{BinTree} \equiv \mathsf{W}(\{\mathit{leaf}, \mathit{node}\}, (x)\mathsf{case}(x, \mathit{leaf} : \{\}, \mathit{node} : \{\mathit{left}, \mathit{right}\}))$$

represents the ML-type above. The element $\mathsf{sup}(\mathit{leaf}, (x)\mathsf{case}_{\{\}}(x))$ represents the ML object *leaf* and $\mathsf{sup}(\mathit{node}, (x)\mathsf{case}(x, \mathit{left} : \hat{t_1}, \mathit{right} : \hat{t_2}))$ represents $\mathit{node}(t_1, t_2)$ if $\hat{t_1}$ represents t_1 and $\hat{t_2}$ represents t_2. Notice that we need an extensional version of type theory to insure that all elements of the form $\mathsf{sup}(\mathit{leaf}, z)$, where $z(x) : \mathit{BinTree}\,[x : \{\}]$, are equal to $\mathsf{sup}(\mathit{leaf}, (x)\mathsf{case}_{\{\}}(x))$.

It comes perhaps as a surprising fact that the wellorder set constructor is not constrained to datatypes of the limited form above. We can also allow constant sets to be used in the cartesian product C_i. This is made possible by viewing the constant part of C_i as part of the constructor. Let us see how this works by adding a natural number to the nodes in the example above.

$$\textbf{datatype } \mathit{BinTree} = \mathit{leaf} \mid \mathit{node} \textbf{ of } \mathsf{N} * \mathit{BinTree} * \mathit{BinTree}$$

The set of constructors now becomes $\{\mathit{leaf}\} + \mathsf{N}$, the selector set $B(\mathsf{inl}(\mathit{leaf}))$ is the empty set and $B(\mathsf{inr}(n))$ is the set $\{\mathit{left}, \mathit{right}\}$. Let us give an ML-like definition to show what we have done:

$$\textbf{datatype } \mathit{BinTree} = \mathit{leaf} \mid \mathit{node}\ N \textbf{ of } \mathit{BinTree} * \mathit{BinTree}$$

[2]In order to define many families of sets we need a set of sets, a universe. When we use it here, we will identify sets and elements of the universe and also use a slightly changed syntax for case expressions.

The intuition is that we have a different node constructor for each natural number. This is possible for the wellordering since the constructor set may be infinite. The result is the wellorder

$$\mathsf{W}(\{\mathit{leaf}\} + \mathsf{N}, (x)\mathsf{when}(x, (y)\{\}, (y)\{\mathit{left}, \mathit{right}\}))$$

which represents the ML datatype. The ML-value *leaf* is represented by $\mathsf{sup}(\mathsf{inl}(\mathit{leaf}), \mathsf{case}_{\{\}})$ and $\mathit{node}(n, t_1, t_2)$ is represented by $\mathsf{sup}(\mathsf{inr}(n), (x)\mathsf{case}(x, \hat{t_1}, \hat{t_2}))$, if $\hat{t_1}$ represents t_1 and $\hat{t_2}$ represents t_2. The method outlined above can also be extended to definitions where C_i is of the form $K \to T$ and where K is a constant.

As an example of a function from an inductive set, we define the function that adds all numbers in the binary trees we defined above. In ML such a function could be defined as

$$\begin{array}{lll} \textbf{fun}\ f(\mathit{leaf}) & = 0 \\ \mid \quad f(\mathit{node}(n, t_1, t_2)) & = n + f(t_1) + f(t_2); \end{array}$$

and in Type Theory it could be defined as

$$f(w) = \mathsf{wrec}(w, (x, y, z)\mathsf{when}(x, (u)0, (u)u + z(\mathit{left}) + z(\mathit{right})))$$

2 The Tree Set Constructor

When trying to mimic mutually recursive inductive definitions with the wellorder set constructor one encounters a problem since it is not always the case that all components of an element belong to the same set as the element itself. For example if we want to represent the types defined in ML by:

$$\begin{array}{lll} \textbf{datatype}\ \mathit{Odd} & = \mathit{sO}\ \textbf{of}\ \mathit{Even} \\ \textbf{and} & \mathit{Even} = \mathit{zeroE} \mid \mathit{sE}\ \textbf{of}\ \mathit{Odd}; \end{array}$$

we can not do this directly using wellorders, we have to introduce a more complicated set constructor.

The constructor should produce a family of sets instead of one set as the wellorder set constructor did. In order to do this, we introduce a *name set*, which is a set of names of the mutually defined sets in the inductive definition.

A suitable choice of name set for the example above would be $\{Odd, Even\}$. Instead of having one set of constructors B and one index family C over B, as we had in the wellorder case, we now have one constructor set and one index set for each element in A. The constructors form a family of sets B, where $B(x)$ is a set for each x in A and the index set form a family of sets C where $C(x, y)$ is a set for each x in A and y in $B(x)$. Furthermore, since the parts of a tree now may come from different sets, we introduce a function d which provides information about this; $d(x, y, z)$ is an element of A if $x : A$, $y : B(x)$ and $z : C(x, y)$. We call this element the *component set name*.

The family of sets $\mathsf{Tree}(A, B, C, d)$ is a representation of the family of sets introduced by a collection of inductive definitions, for example an ML datatype definition. It could also be seen as a solution to the equation

$$\mathcal{T} \cong (x) \sum_{y:B(x)} \prod_{z:C(x,y)} \mathcal{T}(d(x, y, z))$$

where $\mathcal{T}$ is a family of sets over A and $x : A$. This equation could be interpreted as a possibly infinite collection of ordinary set equations, one for each $a : A$.

$$\begin{aligned} \mathcal{T}(a_1) &\cong \sum_{y:B(a_1)} \prod_{z:C(a_1,y)} \mathcal{T}(d(a_1, y, z)) \\ \mathcal{T}(a_2) &\cong \sum_{y:B(a_2)} \prod_{z:C(a_2,y)} \mathcal{T}(d(a_2, y, z)) \\ &\vdots \end{aligned}$$

Or, if we want to express the tree set constructor as the least fixed point of a set function operator.

$$\mathsf{Tree}(A, B, C, d) \cong \mathsf{FIX}((\mathcal{T})(x) \sum_{y:B(x)} \prod_{z:C(x,y)} \mathcal{T}(d(x, y, z)))$$

Comparing this equation with the equation for the wellorder set constructor, we can se that it is a generalization in that the non-dependent function set, "$\rightarrow$", has become a set of dependent functions, "$\prod$". This is a natural generalization since we are now defining a family of sets instead of just one set and every instance of the family could be defined in terms of every one of the other instances. It is the function d that expresses this relation.

2.1 Rules

The formation rule for the set of trees is:

$$\frac{\begin{array}{l} A : \text{ set} \\ B(x) : \text{ set } \ [x : A] \\ C(x,y) : \text{ set } \ [x : A, y : B(x)] \\ d(x,y,z) : A \ \ [x : A, y : B(x), z : C(x,y)] \\ a : A \end{array}}{\mathsf{Tree}(A,B,C,d,a) : \text{ set}}$$

The different parts have the following intuitive meaning:

- A, the name set, is a set of names for the mutually dependent sets.
- $B(x)$, the constructor set, is a set of names for the clauses defining the set x.
- $C(x,y)$, the index set, is a set of names for selectors of the parts in the clause y in the definition of x.
- $d(x,y,z)$, the component set name, is the name of the set corresponding to the selector z in clause y in the definition of x.
- a determines a particular instance of the family of sets.

Understood as a set of syntax-trees generated by a grammar, the different parts have the following intuitive meaning:

- A is a set of non-terminals.
- $B(x)$ is a set of names for the alternatives defining the non-terminal x.
- $C(x,y)$, is a set of names for positions in the sequence of non-terminals in the clause y in the definition of x.
- $d(x,y,z)$, is the name of the non-terminal corresponding to the position z in clause y in the definition of x.
- a is the startsymbol.

In order to reduce the notational complexity, we write $\mathcal{T}(a)$ instead of $\mathsf{Tree}(A, B, C, d, a)$ in the rest of the paper.

Introduction rule:

$$\frac{\begin{array}{l} a : A \\ b : B(a) \\ c(z) : \mathcal{T}(d(a,b,z)) \quad [z : C(a,b)] \end{array}}{\mathsf{tree}(a,b,c) : \mathcal{T}(a)}$$

Intuitively:

- a is the name of one of the mutually dependent sets.
- b is one of the constructors of the set a.
- c is a function from $C(a,b)$ to a tree. This function defines the different parts of the element.

The element $\mathsf{tree}(a,b,c)$ in the set $\mathcal{T}(a)$ corresponds to the tree in figure 2, where $C(a,b) = \{z_1, \ldots, z_n, \ldots\}$ and $c(z_i) : \mathcal{T}(d(a,b,z_i))$.

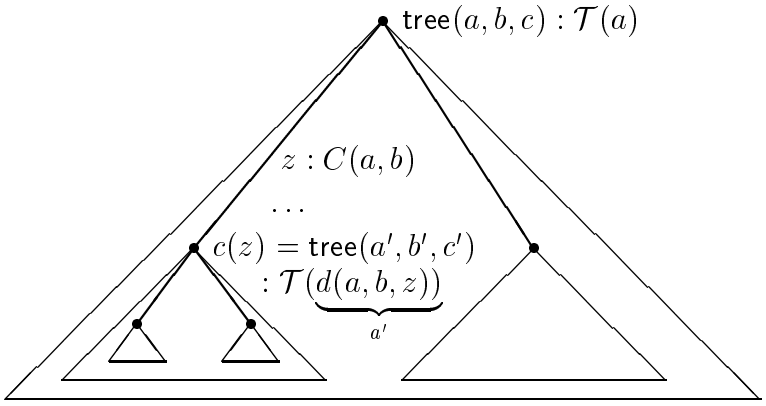

Figure 2: An element of a tree

Elimination rule:

$$\frac{\begin{array}{l} D(x,t):\ \mathrm{set}\ \ [x:A, t:\mathcal{T}(x)] \\ a:A \\ t:\mathcal{T}(a) \\ f(x,y,z,u):D(x,\mathsf{tree}(x,y,z)) \\ \qquad [x:A, y:B(x), z(v):\mathcal{T}(d(x,y,v))\,[v:C(x,y)], \\ \qquad\ u(v):D(d(x,y,v),z(v))\,[v:C(x,y)]] \end{array}}{\mathsf{treerec}(t,f):D(a,t)}$$

The non-canonical constant treerec has the following computation rule:

$$\mathsf{treerec}(\mathsf{tree}(a,b,c),f)\ \rightarrow\ f(a,b,c,(x)\mathsf{treerec}(c(x),f))$$

This is reflected in the equality rule:

$$\frac{\begin{array}{l} D(x,t):\ \mathrm{set}\ \ [x:A, t:\mathcal{T}(x)] \\ a:A \\ b:B(a) \\ c(z):\mathcal{T}(d(a,b,z))\ \ [z:C(a,b)] \\ f(x,y,z,u):D(x,\mathsf{tree}(x,y,z)) \\ \qquad [x:A, y:B(x), z(v):\mathcal{T}(d(x,y,v))\,[v:C(x,y)], \\ \qquad\ u(v):D(d(x,y,v),z(v))\,[v:C(x,y)]] \end{array}}{\mathsf{treerec}(\mathsf{tree}(a,b,c),f)=f(a,b,c,(x)\mathsf{treerec}(c(x),f)):D(a,\mathsf{tree}(a,b,c))}$$

3 Relation to the Wellorder Set Constructor

A wellorder set $\mathsf{W}(B,C)$ can be seen as an instance of a Tree set. We get the wellorders by defining a family of trees over a set with only one element. If we make the definitions:

$$\begin{array}{rcl} \mathsf{W}(B,C) & = & \mathsf{Tree}(\{\mathsf{e}\},(x)B,(x,y)C(y),(x,y,z)\mathsf{e},\mathsf{e}) \\ \mathsf{sup}(b,c) & = & \mathsf{tree}(\mathsf{e},b,c) \\ \mathsf{wrec}(t,f) & = & \mathsf{treerec}(t,(x,y,z,u)f(y,z,u)) \end{array}$$

where $\{\mathsf{e}\}$ is the set consisting of the element e. Then we can derive the rules for wellorders from the rules for trees as follows:

Formation rule If we assume that the premisses of the wellorder formation rule hold, that is, if we assume

$$B : \text{ set}$$
$$C(y) : \text{ set} \quad [y : B]$$

we can infer

$$\{\mathsf{e}\} : \text{ set}$$
$$((x)B)(x) : \text{ set} \quad [x : \{\mathsf{e}\}]$$
$$((x, y)C(y))(x, y) : \text{ set} \quad [x : \{\mathsf{e}\}, y : B]$$
$$((x, y, z)\mathsf{e})(x, y, z) : \{\mathsf{e}\} \quad [x : \{\mathsf{e}\}, y : B, z : C(y)]$$
$$\mathsf{e} : \{\mathsf{e}\}$$

and then, by the Tree-formation rule, get

$$\mathsf{Tree}(\{\mathsf{e}\}, (x)B, (x, y)C(y), (x, y, z)\mathsf{e}, \mathsf{e}) : \text{ set}$$

which is the same as

$$\mathsf{W}(B, C) : \text{ set}$$

and also the conclusion of the formation rule. So we have proved that the formation rule holds for the definition of wellorders in terms of trees.

Introduction rule Assume

$$b : B$$
$$c(z) : \mathsf{W}(B, C) \quad [z : C(b)]$$

From the last assumption we get

$$c(z) : \mathsf{Tree}(\{\mathsf{e}\}, (x)B, (x, y)C(y), (x, y, z)\mathsf{e}, \mathsf{e}) \quad [z : C(b)]$$

It then follows that

$\mathsf{e} : \{\mathsf{e}\}$
$b : ((x)B)(\mathsf{e})$
$c(z) : \mathsf{Tree}(\{\mathsf{e}\}, (x)B, (x,y)C(y), (x,y,z)\mathsf{e}, ((x,y,z)\mathsf{e}))(\mathsf{e}, b, z) \quad [z : ((x,y)C(y))(b)]$

and, from the Tree-introduction rule,

$\mathsf{tree}(\mathsf{e}, b, c) : \mathsf{Tree}(\{\mathsf{e}\}, (x)B, (x,y)C(y), (x,y,z)\mathsf{e}, \mathsf{e})$

which is the same as

$\mathsf{sup}(b, c) : \mathsf{W}(B, C)$

The elimination and equality rules could be proved in the same way.

4 A Derived Version of the Tree Set Constructor

In this section we give a short description of an internal definition of Tree, called Tree_I, It is defined using the W set constructor. In [Syn89] we give the formal definition and prove that it is correct.

The construction of Tree_I is done in two steps. First we define a set constructor Ap, such that for all elements in $\mathsf{Tree}(A, B, C, d, a)$ there is a corresponding element in the approximation $\mathsf{Ap}(A, B, C)$. The set $\mathsf{Ap}(A, B, C)$ is an instance of the well order set constructor and does not simulate $\mathsf{Tree}(A, B, C, d, a)$ fully, i.e. there are elements in $\mathsf{Ap}(A, B, C)$ which have no correspondence in $\mathsf{Tree}(A, B, C, d, a)$. We then define a property $\mathsf{Ok}(A, C, d, a, t)$, where t is an element in $\mathsf{Ap}(A, B, C)$, which is true iff there is a corresponding element in $\mathsf{Tree}(A, B, C, d, a)$.

The set which simulates $\mathsf{Tree}(A, B, C, d, a)$ is the set of all x in $\mathsf{Ap}(A, B, C)$ such that
$\mathsf{Ok}(A, C, d, a, t)$ is true. Since we do not have subsets in pure type theory we define Tree_I as a disjoint union of $\mathsf{Ok}(A, B, C, d, a)$ over $\mathsf{Ap}(A, B, C)$.

$$\mathsf{Tree}_I(A, B, C, d, a) \equiv \sum_{t:\mathsf{Ap}(A,B,C)} \mathsf{Ok}(A, C, d, a, t)$$

Instead of viewing the rules for Tree as primitive we can now prove them for Tree_I. The proof object of the introduction rule is the internal version of tree and the proof object of the elimination rule is the internal version of treerec.

5 A Variant of the Tree Set Constructor

We will in this section introduce a slight variant of the tree set constructor. Instead of having information in the element about what instance of the family a particular element belongs to, we move this information to the recursion operator. We call the new set constructor Tree', the new element constructor tree' and the new recursion operator $\mathsf{treerec}'$. The formation rule for Tree' is exactly the same as for Tree, but the other rules are slightly modified.

Introduction rule:

$$\frac{\begin{array}{l} a : A \\ b : B(a) \\ c(z) : \mathsf{Tree}'(A, B, C, d, d(a, b, z)) \quad [z : C(a, b)] \end{array}}{\mathsf{tree}'(b, c) : \mathsf{Tree}'(A, B, C, d, a)}$$

Elimination rule:

$$\frac{\begin{array}{l} D(x, t) : \text{ set } \quad [x : A, t : \mathsf{Tree}'(A, B, C, d, x)] \\ a : A \\ t : \mathsf{Tree}'(A, B, C, d, a) \\ f(x, y, z, u) : D(x, \mathsf{tree}'(y, z)) \\ \qquad [x : A, y : B(x), z(v) : \mathsf{Tree}'(A, B, C, d, d(x, y, v))\,[v : C(x, y)], \\ \qquad\ u(v) : D(d(x, y, v), z(v))\,[v : C(x, y)]] \end{array}}{\mathsf{treerec}'(d, a, t, f) : D(a, t)}$$

We leave the formulation of the equality rule to the reader. Notice that we in the first version of the tree sets can view the constructor tree as a family of constructors, one for each $a : A$. In the variant we have one constructor for the whole family, but instead we get a family of recursion operators, one for each a in A.

6 Examples of Different Tree Sets

6.1 Even and odd numbers

Consider the following datatype definition in ML:

$$\begin{array}{l} \textbf{datatype } Odd \;=\; sO \textbf{ of } Even \\ \textbf{and} \qquad\; Even = zeroE \mid sE \textbf{ of } Odd; \end{array}$$

and the corresponding grammar:

$$\begin{array}{lcl} \text{<odd>} & ::= & s_O(\text{<even>}) \\ \text{<even>} & ::= & 0_E \;\mid\; s_E(\text{<odd>}) \end{array}$$

If we want to define a set with elements corresponding to the phrases defined by this grammar (and if we consider <odd> as startsymbol), we can define $OddNrs \;=\; \mathsf{Tree}(A, B, C, d, a)$ where:

$$\begin{array}{rcl} A & = & \{Odd, Even\} \\ \\ a & = & Odd \\ \\ B(Odd) & = & \{s_O\} \\ B(Even) & = & \{zero_E, s_E\} \\ i.e.\ B & = & (x)\mathsf{case}(x, Odd : \{s_O\}, Even : \{zero_E, s_E\}) \\ \\ C(Odd, s_O) & = & \{pred_O\} \\ C(Even, zero_E) & = & \{\} \\ C(Even, s_E) & = & \{pred_E\} \\ i.e.\ C & = & (x, y)\mathsf{case}(x, \\ & & \qquad Odd : \{pred_O\}, \\ & & \qquad Even : \mathsf{case}(y, zero_E : \{\}, s_E : \{pred_E\})) \\ \\ d(Odd, s_O, pred_O) & = & Even \\ d(Even, s_E, pred_E) & = & Odd \\ i.e.\ d & = & (x, y, z)\mathsf{case}(x, \\ & & \qquad Odd : Even, \\ & & \qquad Even : \mathsf{case}(y, zero_E : \mathsf{case}_{\{\}}(z), s_E : Odd)) \end{array}$$

The element $s_E(s_O(zero_E))$ is represented by

$$2_E \;=\; \mathsf{tree}(Even, s_E, (x)\mathsf{tree}(Odd, s_O, (x)\mathsf{tree}(Even, zero_E, (x)\mathsf{case}_{\{\}}(x))))$$

and $s_O(s_E(s_O(0_E)))$ is represented by

$$3_O \quad = \quad \mathsf{tree}(Odd, s_O, (x)2_E)$$

We get the set of even numbers by just changing the "startsymbol"

$$EvenNrs \quad = \quad \mathsf{Tree}(A, B, C, d, Even)$$

and we can define a mapping from even or odd numbers to ordinary natural numbers by:

$$\begin{array}{lcl} tonat(w) & = & \mathsf{treerec}(w, \\ & & \quad (x, y, z, u)\, \mathsf{case}(x, \\ & & \quad\quad Odd : \mathsf{succ}(u(pred_O)), \\ & & \quad\quad Even : \mathsf{case}(y, \\ & & \quad\quad\quad zero_E : \mathsf{0}, \\ & & \quad\quad\quad s_E : \mathsf{succ}(u(pred_E))))) \end{array}$$

$$tonat(w) \quad : \quad \mathsf{N} \quad [v : \{Odd, Even\}, w : \mathsf{Tree}(A, B, C, d, v)]$$

6.2 An infinite family of sets

In ML, and all other programming languages with some facility to define mutually inductive types, one can only introduce finitely many new datatypes. A family of sets in type theory, on the other hand, could range over infinite sets and the tree set constructor therefore could introduce families with infinitely many instances. In this section we will give an example where the name set is infinite.

The problem is to define a set $\mathsf{Array}(A, n)$, whose elements are lists with exactly n elements from the set A. If we make a generalization of ML's datatype construction to dependent types this type could be defined as:

$$\begin{array}{lcl} \mathsf{Array}(E, 0) & = & \mathsf{empty} \\ \mathsf{Array}(E, s(n)) & = & \mathsf{add\ of}\ E * \mathsf{Array}(E, n)) \end{array}$$

The corresponding definition with the tree set constructor is:

$$\mathsf{Array}(E, n) \quad = \quad \mathsf{Tree}'(\mathsf{N}, B, C, d, n)$$

where

$$\begin{array}{lcl} B(n) & = & \mathsf{natrec}(n, \{\mathsf{nil}\}, (x,y)E) \\ C(n,x) & = & \mathsf{natrec}(n, \{\}, (x,y)\{\mathsf{tail}\}) \\ d(n,x,y) & = & \mathsf{natrec}(n, \mathsf{case}_{\{\}}(y), (z,u)z) \end{array}$$

We can then define:

$$\begin{array}{lcl} \mathsf{empty} & = & \mathsf{tree}'(\mathsf{nil}, \mathsf{case}_{\{\}}) \\ \mathsf{add}(e,l) & = & \mathsf{tree}'(e,l) \end{array}$$

as the elements. Notice that we in this example have used the variant of the tree constructor we introduced in section 5.

7 Related Work

Several different approaches to the introduction of inductive sets in type theory have already been suggested:

- Using the wellorder set constructor in type theory.
- Introducing a fixed point set constructor [CM85, Dyb87].
- Introducing a method to add new rules [Bac87].

We have seen that the wellorder set constructor is not general enough such that all inductive sets that we need are instances of it. The fixed point set constructor, on the other hand, is very complicated. Several different collections of proof rules have been suggested, and we think it is hard, or at least much harder than for most other sets, to realize the correctness of the rules. One reason why the rules are so complicated is because they do not follow the pattern from other set constructors. There is also one premise that requires a substantial modification of the whole theory (the subtype relation between sets) and another of a very "syntactical nature" (the notion of strictly positive set operator). Furthermore, if one wants to define mutually recursive sets, the fixed point construction seems as inadequate as the wellorder constructor and has to be extended to give fixed points to operators over families of sets.

The third approach is not sufficient for our purposes because it is not formalized within the Type Theory formalism and we want to see the set

constructor for inductive sets as a formal entity which we could reason about using the rules of the theory. We want, for example, to be able to make a specification of a set and then formally derive an implementation using the proof rules of Type Theory.

8 Conclusion

We have presented a set constructor in the framework of Per Martin-Löf's type theory. It is a natural generalization of the wellorder set constructor to families of sets. The families of mutually dependent sets that can be introduced using the tree set constructor occur frequently in computer science, for example when representing abstract syntax trees.

An interesting application of the tree type constructor is for specifying parsers. In [Chi87], Chisholm proves the correctness of a parser for a small language. However, since he introduces a new set for the particular parse trees of his language, he has no concept of a grammar or any specification of a parser in general.

Another application is to represent the context sensitive parts of a language using an infinite family of sets. A method like this have already been used in the specification of Algol68 [vWea74] where a two level grammar was used.

Acknowledgement

We would like to thank Peter Dybjer and Per Martin-Löf for their comments on an earlier version of this paper.

References

[Bac87] Roland Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In *Proceedings of the Workshop on General Logic, Edinburgh*. Laboratory for the Foundations of Computer Science, University of Edinburgh, February 1987.

[Chi87] P. Chisholm. Derivation of a Parsing Algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, 8:1–42, 1987.

[CM85] Robert L. Constable and Nax P. Mendler. Recursive Definitions in Type Theory. In *Proceedings of the LICS-Conference, Brooklyn, N. Y., Lecture Notes in Computer Science*. Springer-Verlag, June 1985.

[Dyb87] Peter Dybjer. Inductively defined sets in Martin-Löf's type theory. In *Proceedings of the Workshop on General Logic, Edinburgh*, February 1987. Draft paper.

[Hoa75] C. A. R. Hoare. Recursive Data Structures. *International Journal of Computer and Information Sciences*, 4(2):105–132, 1975.

[Mil84] R. Milner. Standard ML Proposal. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), January 1984.

[ML82] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[Sco82] Dana Scott. Domains for Denotational Semantics. In *Automata, Languages and Programming, 9th Colloquium*, pages 577–613. Springer-Verlag, LNCS 140, July 1982.

[Syn89] Dan Synek. Deriving Rules for Inductive Sets in Martin-Löf's Type Theory. Technical report, Programming Methodology Group, Department of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, 1989. Draft.

[vWea74] A. van Wijngarden et al. Revised report on the Algorithmic Language, ALGOL 68. *Acta Informatica*, 5:1–236, 1974.