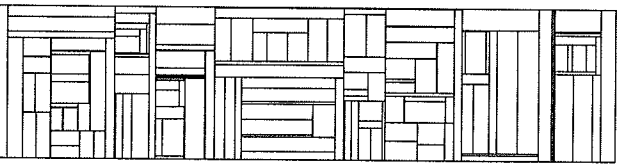
ISSN 0105-8517

Infinite Values in Hierarchical Imperative Types

Michael I. Schwartzbach

DAIMI PB – 293
September 1989

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 86 12 71 88 *Telex: 64767 aausci dk*



Infinite Values in Hierarchical Imperative Types

Michael I. Schwartzbach

Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Århus C, Denmark

Abstract

A system of hierarchical imperative types is extended to allow *infinite* values. The general structure of *value assignments* to types in the context of a hierarchy is considered, and it is shown that a *smallest* and *largest* such exist. A method for obtaining *intermediate* value assignments is investigated, and a general characterization of the infinite values allowable in programming languages is presented. Finally, the set of *rational values* is demonstrated to be appropriate for our imperative hierarchy. Programs can then work on infinite imperative data structures which are allocated lazily during execution.

1 Introduction

In [7] we presented a system of fully recursive types for an imperative programming language. The type system is *hierarchical*, which means that it has a partial ordering of the types, expressing that *any* application written for a small type may be reused for *all* larger types without risk of inconsistencies; static type checking is still possible. Types are specified through a set of recursive equations, and their value sets are taken to be the standard least solutions of the induced equations on sets.

In this paper, we consider the introduction of *infinite* values. The general idea is that a type equation such as

$$\textbf{Type}\ T = \mathrm{Int} \times T$$

no longer denotes the *empty* type, which is clearly the least solution, but rather a type of infinite sequences of integers. This is a well-known idea, which is introduced in the *equational programming* of [5], as *streams* in several functional programming languages such as Miranda [4], and to some extent by the lazy CONS operator in Lisp [3]. The idea is also present in *constructive type theory*, where one distinguishes between *recursive* and *infinite* solutions of more general type equations [1,6]. The aims of our approach are somewhat different from all of these.

Firstly, in our case we have no interest in equational programming (or streams). Recursive types are pragmatically useful in conventional imperative languages, as they abolish the need for explicit pointer manipulations and the writing of a plethora of little procedures for e.g. copying, comparing and storing recursive values. However, when using a recursive tree structure one usually still has to perform a multitude of explicit allocations. For example, when inserting into a search tree one repeatedly replaces an empty leaf by a singleton tree with a node and two empty leaves. This *demolish-and-rebuild* approach is both tedious and expensive. The leaves are really redundant and serve only as a device to limit the recursive unfolding. It is preferable to start out with an infinite search tree (without values in the nodes) in which no explicit further allocation is necessary or indeed possible. With this motivation, our use of infinite values is primarily one of *initializing* infinite data structures.

Secondly, this is to take place in an imperative setting, where variables containing structured values are composed of a similar structure of subvariables. Also, the type hierarchy is to survive, which requires some care.

2 Hierarchical Imperative Types

In this section we shall briefly review the type system of [7]. Types are defined by means of a set of *type equations*

Type $N_1 = T_1$
Type $N_2 = T_2$
...
Type $N_k = T_k$

where the N_i's are *names* and the T_i's are *type expressions*, which are defined as follows

$$\begin{array}{lll} \mathrm{T} ::= & \mathrm{Int} \mid \mathrm{Bool} \mid & \text{simple types} \\ & N_i \mid & \text{type names} \\ & *T & \text{lists} \\ & (n_1 : T_1, \ldots, n_k : T_k) \mid & \text{partial products, } k \geq 0,\ n_i \in \mathcal{N},\ n_i \neq n_j \end{array}$$

Here $\mathcal{N}$ is an infinite set of names. Types are denoted by type expressions. Notice that type definitions may involve arbitrary recursion.

The $*$-operator corresponds to ordinary finite lists. The *partial product* is a generalization of sums and products; its values are *partial* functions from the tag names to values of the corresponding types, in much the same way that products may be regarded as *total* functions. This partiality is essential to the consistency of the hierarchy.

Structural Invariants

In [7] the partial product is combined with *structural invariants* to enable a technique for specifying (recursive) types, which is more compact and convenient than the usual sums-and-products or records-and-pointers. A structural invariant is associated with a partial product and specifies a set of legal domains for the corresponding partial functions. Often a logical notation is used, so that for example the type of binary integer trees may be specified as

Type Tree = (val: Int, left,right: Tree) ! { (left $\vee$ right) $\Rightarrow$ val }

This invariant actually specifies the set of domains

$$\{\{\mathrm{val}\}, \{\mathrm{val}, \mathrm{left}\}, \{\mathrm{val}, \mathrm{right}\}, \{\mathrm{val}, \mathrm{left}, \mathrm{right}\}\}$$

Without the invariant the partiality would allow values that are not *tree-like*, e.g. one with a left- but no val-component. Notice that sums and products may be recovered as partial products with appropriate invariants; in fact, we shall use the usual notation $\times$ for the binary partial product with a Cartesian product-like structural invariant, i.e.

$$T_1 \times T_2 \equiv (\mathrm{fst} : T_1, \mathrm{snd} : T_2)\ !\ \{\mathrm{fst} \wedge \mathrm{snd}\}$$

Partial products with structural invariants are pragmatically superior to the standard sums and products for two reasons. Firstly, the use of sums and products is equivalent to expressing the invariants using only the XOR and AND operators, which is clearly inconvenient[1]. Secondly, the nesting of sums and products force components belonging at the same *conceptual* level to appear at different *syntactical* levels[2]. The partial products alleviate these disadvantages.

Another approach could be to employ *more* type constructors. We can think of the partial product as the Cartesian product of domain-like sets with a $\perp$ element to indicate undefinedness. In [8] a great number of binary domain constructors are considered for the purposes of specifying various domains of infinite values. Some of these correspond to logical operators in the above sense; for example, the separated product $\times_\perp$ seems to resemble the OR operator. Since $\perp$ is always present it is, however, unclear how to *insist* on the presence of a component, which is necessary to define e.g. the AND operator. Also, compositionality seems to break down, since we at the same time want $\neg A = \{\perp\}$ and $\neg\neg A = A$. In any case, with unary or binary constructors the notational disadvantages remain. The structural invariants provide an n-ary type constructor for each n-place propositional statement.

Normal Forms and Type Equivalence

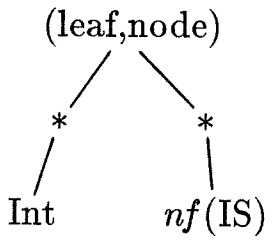
We define an equivalence relation $\approx$ on type expressions, which identifies different type expressions denoting the same *type*. This equivalence is defined as the identity of *normal forms*. To each type expression E we associate a unique normal form $nf(E)$, which is a possibly-infinite labeled tree. Informally, the tree is obtained by *unfolding* the type expression. If we regard the definitions

Type IL = *Int
Type IS = (leaf: IL, node: *IS)

we would expect the normal form nf(IS) to be the infinite tree indicated by

[1] At the theoretical worst, the size of the notation may expand exponentially.

[2] *Variant records* in PASCAL-like languages avoid this problem (while introducing others).



This is merely a short-hand notation for the full tree. Formally, we use the fact that the set of labeled trees form a *complete partial order* under the partial ordering $\sqsubseteq$, where $t_1 \sqsubseteq t_2$, iff t_1 can be obtained from t_2 by replacing any number of subtrees with the singleton tree Ω. In this setting, normal forms can be defined as limits of chains of approximations. The singleton tree Ω is smaller than all other trees and corresponds to the normal form of the type defined by

Type $T = T$

which we shall refer to as the *vacuous* type.

Let us call a type equivalence in this context *consistent* when: No two types with a different outermost type constructor may be identified, and if $F(S_1, \ldots, S_k)$ is equivalent to $F(T_1, \ldots, T_k)$ then each S_i must be equivalent to T_i. The former requirement is self-evident; the latter is necessary to allow consistent selection of subvariables.

Fact 2.1: The equivalence $\approx$ is the largest consistent one.

Thus, $\approx$ is *final* and types are deemed equivalent unless there is some reason for them not to be. As normal forms have a very regular structure, the equivalence of two type expressions is decidable.

We use the notation $|t| = \infty$ and $|t| < \infty$ to denote whether a tree t is infinite or not. The notation *labels*(t) denotes the set of labels in t. Notice that all normal forms of types have finite label sets.

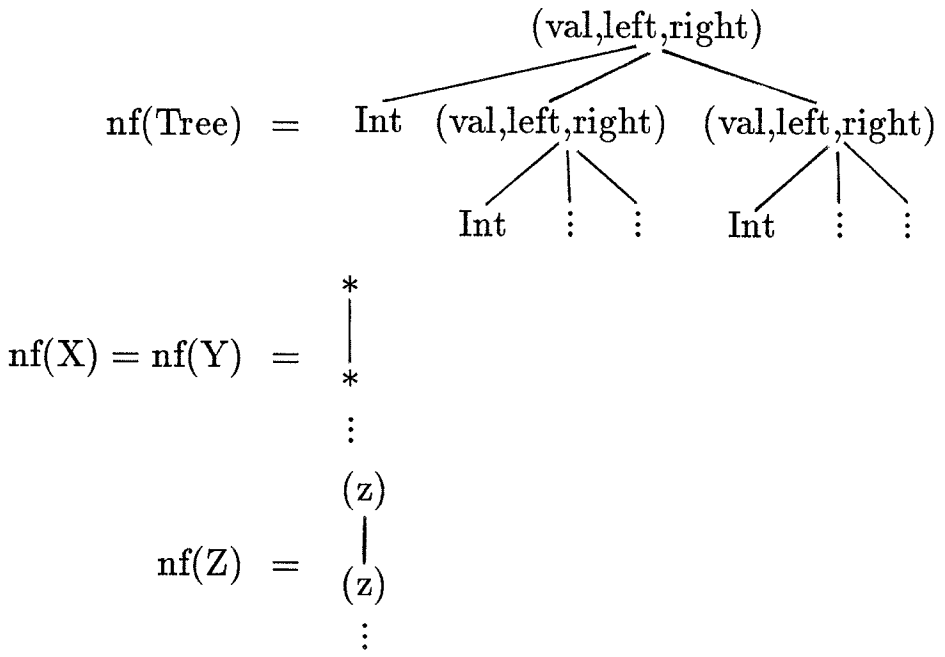
Examples

For the type definitions

Type Tree = (val: Int, left,right: Tree)
Type X = $*$X
Type Y = $**Y$

Type Z = (z:Z)

we have the following normalforms



The Type Ordering

To obtain our hierarchy we need a partial ordering $\preceq$ on types, which ensures that if the relation $T_1 \preceq T_2$ holds, then all applications written for the type T_1 may be reused for the type T_2.

All types allow the definition and use of variables. Int and Bool come with the usual operations. Lists and partial products provide expressions denoting arbitrary constants and the selection of subvariables. Furthermore, the partial products have the usual operations of partial functions, e.g. test for definedness and inclusion and exclusion of components.

In this setting it is hardly surprising that $\sqsubseteq$ may serve as a hierarchical ordering. This ordering may be refined further, by observing that a partial product allows all of the manipulations that are possible for products with *fewer* components, i.e. selection of components and (thanks to the partiality) formation of constants. Hence, we can obtain a better hierarchical ordering, $\preceq$, as the largest consistent refinement of $\sqsubseteq$ which

satisfies the rule

$$\begin{array}{ccc} (n_i) & & (m_j) \\ | & \preceq & | \\ T_i & & S_j \end{array}$$

iff $\{n_i\} \subseteq \{m_j\}$ and $n_i = m_j \Rightarrow T_i \preceq S_j$. To illustrate this ordering, we can observe that the relations

$$\Omega \quad \preceq \quad \begin{array}{c} (a,b) \\ \diagup \quad \diagdown \\ \Omega \quad T_2 \end{array} \quad \preceq \quad \begin{array}{ccc} & (a,b,c) & \\ \diagup & | & \diagdown \\ \Omega & T_2 & T_3 \end{array} \quad \preceq \quad \begin{array}{ccc} & (a,b,c) & \\ \diagup & | & \diagdown \\ T_1 & T_2 & T_3 \end{array}$$

are true for all T_i. In the presence of structural invariants this ordering is somewhat more complicated; details are described in [7], where we also extend the parameter passing mechanism to exploit the hierarchical structure, essentially by allowing the actual parameters to be larger than the formal parameters, subject to certain homogeneity conditions.

The least upper bounds of $\preceq$ correspond to the multiple inheritance aspect of data values: Two types can be joined by the (recursive) unification of their components. In fact, we obtain a generalization of the ordinary multiple inheritance, since we have recursive (infinite) types and the polymorphic type Ω. Least upper bounds may or may not exist, whereas greatest lower bounds always exist.

The type ordering is decidable in much the same way as the equivalence, and least upper bounds as well as greatest lower bounds are computable.

3 Value Assignments

In this setting we define the values to be possibly-infinite labeled trees (not to be confused with the normal forms of types). To each type T we must associate a set of values. These *value assignments* must obey certain obvious conditions. A value assignment ϕ must be *homomorphic* in the following sense

$$\begin{aligned} \phi(\text{Int}) &= \{\ldots, -1, 0, 1, 2, \ldots\} \\ \phi(\text{Bool}) &= \{\text{true}, \text{false}\} \\ \phi(*T) &= *\phi(T) \\ \phi((n_i : T_i)) &= (n_i : \phi(T_i)) \end{aligned}$$

where the type constructors have analogous *value constructors* defined as follows: Let S, S_i be sets of values. Then

$$*S = \left\{ \begin{array}{c} * \\ s_1 \quad \cdots \quad s_k \end{array} \mid k \geq 0, s_i \in S \right\}$$

$$(n_i : S_i) = \left\{ \begin{array}{c} (n_{i_j}) \\ | \\ s_{i_j} \end{array} \mid \{n_{i_j}\} \subseteq \{n_i\}, s_{i_j} \in S_{i_j} \right\}$$

In the presence of structural invariants the allowed subsets $\{n_{i_j}\}$ must belong to the invariant, which yields the modified value constructor

$$(n_i : S_i) \,!\, I = \left\{ \begin{array}{c} (n_{i_j}) \\ | \\ s_{i_j} \end{array} \mid \{n_{i_j}\} \in I, s_{i_j} \in S_{i_j} \right\}$$

Proposition 3.1: The value constructors are all $\subseteq$-monotonic and ω-continuous functions on sets of trees.
Proof: Immediate. □

The homomorphic requirements are easy to motivate, since they simply state the intended meaning of the type constructors: The set $*S$ corresponds to *lists* of S-elements, and the set $(n_i : S_i)$ corresponds to *partial functions* from $\{n_i\}$ to the S_i's.

Finally, ϕ must be monotonic with respect to $\preceq$, that is

$$T_1 \preceq T_2 \Rightarrow \phi(T_1) \subseteq \phi(T_2)$$

This requirement is necessary to maintain the hierarchy, since in hierarchical applications values of type T_1 may be assigned to variables of type T_2.

Values of Finite Types

The homomorphic and monotonic requirements are fairly severe, but as we shall see they allow for more than one value assignment. It is, however, the case that all value assignments must agree on all *finite* types, i.e. types with finite normal forms.

Proposition 3.2: If ϕ_1 and ϕ_2 are value assignments, then

$$\forall T : |T| < \infty \Rightarrow \phi_1(T) = \phi_2(T)$$

Proof: For any value assignment ϕ monotonicity implies that $\phi(\Omega) = \emptyset$, since $\Omega \preceq \text{Int}$, $\Omega \preceq \text{Bool}$ and hence $\phi(\Omega) \subseteq \phi(\text{Int}) \cap \phi(\text{Bool}) = \emptyset$. Since T is a finite type with either Int, Bool, or Ω at the leaves, it follows by straight-forward structural induction that $\phi_1(T) = \phi_2(T)$. □

We shall use the notation VAL_{FIN} for this unique value assignment on finite types. Notice that VAL_{FIN} is monotonic, since both value constructors are monotonic and $\emptyset$ is smaller than all others.

For infinite types this structural induction is no longer well-founded and several choices become available.

Recursive Values

Definition 3.3: The recursive value assignment to the (infinite) type given by the equation

$$\textbf{Type } T = F(T)$$

is defined as

$$\text{VAL}_{\text{REC}}(T) = \bigcup_{i \geq 0} \widetilde{F}^i(\emptyset)$$

where $\widetilde{F}$ is the (composite) value constructor derived from the (composite) type constructor F. This is the standard colimit construction of the least fixed point, which generalizes in the obvious way to mutually recursive type equations.

Proposition 3.4: VAL_{REC} is a value assignment.

Proof: The homomorphic requirements are satisfied since the two value constructors are ω-continuous functions on sets. Regarding monotonicity, we may initially observe that

$$\text{VAL}_{\text{REC}}(T) = \bigcup_{S \preceq T,\, |S| < \infty} \text{VAL}_{\text{FIN}}(S)$$

This follows from the facts that all the $F^n(\Omega)$ are finite types, that VAL_{FIN} is monotonic, and that any finite $S \preceq T$ is smaller than some $F^n(\Omega)$. Now,

if $T_1 \preceq T_2$ then

$$\textsc{Val}_{\textsc{rec}}(T_1) = \bigcup_{S \preceq T_1,\, |S|<\infty} \textsc{Val}_{\textsc{fin}}(S) \subseteq \bigcup_{S \preceq T_2,\, |S|<\infty} \textsc{Val}_{\textsc{fin}}(S) = \textsc{Val}_{\textsc{rec}}(T_2)$$

and monotonicity of $\textsc{Val}_{\textsc{rec}}$ follows. □

Using $\textsc{Val}_{\textsc{rec}}$ we do not get any infinite values. The approximants to the value set of the type

$$\textbf{Type } T = \text{Int} \times T$$

never get any bigger than $\emptyset$, since $\tilde{\times}$ is strict on $\emptyset$. In fact, no value assignment can be smaller than $\textsc{Val}_{\textsc{rec}}$.

Proposition 3.5: If ϕ is any value assignment, then

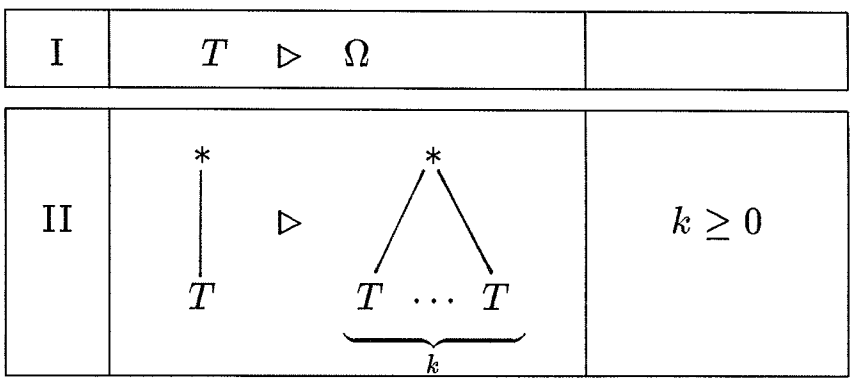
$$\forall T : \textsc{Val}_{\textsc{rec}}(T) \subseteq \phi(T)$$

Proof: Let $v \in \textsc{Val}_{\textsc{rec}}(T)$ be a value. Now, v belongs to some approximant, say $\widetilde{F}^n(\emptyset)$. The homomorphicity implies $v \in \textsc{Val}_{\textsc{rec}}(F^n(\Omega)) = \textsc{Val}_{\textsc{fin}}(F^n(\Omega))$. Then $v \in \phi(F^n(\Omega)) = \textsc{Val}_{\textsc{fin}}(F^n(\Omega))$ and from monotonicity and $F^n(\Omega) \preceq T$ it follows that $v \in \phi(T)$. □

4 Maximal Values

We propose to derive the infinite values through a limit process. Using the fact that both types and values are (infinite) labeled trees, we shall define a transition system which transforms a type into any of its values.

Consider the following non-deterministic transition system on *finite* trees:



III	$\begin{array}{ccc} (n_i) & & (n_{i_j}) \\ \vert & \triangleright & \vert \\ T_i & & T_{i_j} \end{array}$	$\{n_{i_j}\} \subseteq \{n_i\}$
IV	Int $\triangleright$ i	$i \in$ Int
V	Bool $\triangleright$ b	$b \in$ Bool

For products with invariants we have the modified transition:

VI	$\begin{array}{ccc} (n_i)\ !\ I & & (n_{i_j}) \\ \vert & \triangleright & \vert \\ T_i & & T_{i_j} \end{array}$	$\{n_{i_j}\} \in I$

The results of these transitions are not values, since they may contain Ω's; we shall call them *protovalues*[3]. Protovalues are either just values or *approximants* to infinite values.

Proposition 4.1:

$$\forall T,\ |T| < \infty :\ \textsc{Val}_{\textsc{Fin}}(T) = \{v \mid T \rhd^* v \ \wedge\ \text{Int}, \text{Bool}, \Omega \notin \mathit{labels}(v)\}$$

Proof: By induction in the structure of T. It is clearly true for Int, Bool, and Ω. If $T = *S$ we observe that any finite T-value can be obtained by first securing the appropriate fan-out using the II-transition and then inductively expanding the S-subtrees. Similarly, if $T = (n_i : S_i)$. $\square$

We want to generalize this mechanism to infinite types as well; however, this confronts us with the problem of performing a countably infinite number of transition steps. This is, in fact, possible in the present context, since we can perform the transition steps on the directed set (ideal) of finite predecessors and recombine afterwards.

Definition 4.2:

$$t \rhd^\omega v$$

$$\Updownarrow$$

$$(\forall s \preceq t,\ |s| < \infty :\ \exists v_s :\ s \rhd^* v_s) \ \wedge\ (\bigsqcup v_s = v)$$

[3]We do not need the concept of *prototypes(!)*; they are just ordinary types, since Ω is a type.

A similar method for defining *functions* is described in [2].

Definition 4.3: The maximal values are defined to be

$$\text{VAL}_{\text{MAX}}(T) = \{v \mid T \rhd^\omega v \ \wedge \ \text{Int}, \text{Bool}, \Omega \notin \mathit{labels}(v)\}$$

We must verify the required properties of this alleged value assignment, but first we need to establish some natural results.

Lemma 4.4: If t_1, t_2 are finite and $t_1 \preceq t_2$ then $t_2 \rhd^* t_1$.
Proof: By induction in the structure of t_1. If t_1 is Int or Bool, then $t_1 = t_2$ and we are done. If $t_1 = \Omega$ then by transition I we get $t_2 \rhd \Omega = t_1$. If t_1 is a list, then t_2 is a larger list, and we proceed by induction on the subtree. If t_1 is a partial product, then t_2 is a larger partial product, so we can use transition III and continue inductively on the subtrees. □

Lemma 4.5: If $t_1 \preceq t_2$ then $t_2 \rhd^\omega t_1$.
Proof: For all finite $\alpha \preceq t_2$ we define

$$t_\alpha = \begin{cases} \Omega & \text{if } \alpha \not\preceq t_1 \\ \alpha & \text{if } \alpha \preceq t_1 \end{cases}$$

Clearly, $\alpha \rhd^* t_\alpha$ and $\bigsqcup t_\alpha \preceq t_1$. Since every $\alpha \preceq t_1$ is also $\preceq t_2$ we get that $\bigsqcup t_\alpha \succeq t_1$, so $t_1 = \bigsqcup t_\alpha$. Hence, $t_2 \rhd^\omega t_1$. □

Lemma 4.6: $\rhd^\omega$ is reflexive and transitive.
Proof: Reflexivity of $\rhd^\omega$ follows from the reflexivity of $\rhd^*$ since $\alpha \rhd^* \alpha$ and

$$\forall t: \ t = \bigsqcup_{\alpha \preceq t, \, |\alpha| < \infty} \alpha$$

For transitivity we look at $t \rhd^\omega s$ and $s \rhd^\omega r$, that is

$$\forall \alpha \preceq t, |\alpha| < \infty: \ \alpha \rhd^* s_\alpha \wedge \bigsqcup s_\alpha = s$$
$$\forall \beta \preceq s, |\beta| < \infty: \ \beta \rhd^* r_\beta \wedge \bigsqcup r_\beta = r$$

We must show $t \rhd^\omega r$, that is

$$\forall \alpha \preceq t, |\alpha| < \infty: \ \alpha \rhd^* x_\alpha \wedge \bigsqcup x_\alpha = r$$

We define x_α to be a maximal r_β chosen among all $\beta \preceq s_\alpha$, i.e.

$$\forall \beta \preceq s_\alpha : \; x_\alpha \not< r_\beta$$

Using our assumptions and lemma 4.4 we have that

$$\alpha \rhd^* s_\alpha \rhd^* \beta \rhd^* r_\beta = x_\alpha$$

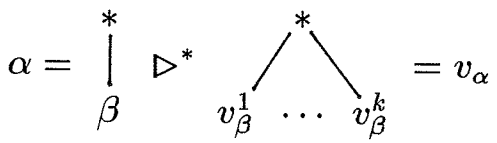
Clearly, $\bigsqcup x_\alpha \preceq \bigsqcup r_\beta$. Since $\bigsqcup s_\alpha = s$ every finite $\beta \preceq s$ is smaller than some s_α. As the x_α are chosen to be maximal, every r_β is smaller than some x_α. Hence, $\bigsqcup r_\beta \preceq \bigsqcup x_\alpha$, so $t \rhd^\omega \bigsqcup x_\alpha = \bigsqcup r_\alpha = r$. $\square$

Now we can show the desired properties of the maximal value assignment.

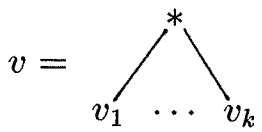
Theorem 4.7: VAL_{MAX} is homomorphic.

Proof: Regard any value $v \in \text{VAL}_{\text{MAX}}(*T)$. For every finite $s \preceq *T$ we have $\alpha \rhd^* v_\alpha$ such that $\bigsqcup v_\alpha = v$. We may ignore all the v_α's which equal Ω, since they cannot contribute to the value. The finite subtrees of $*T$ which can yield protovalues different from Ω are all of the form

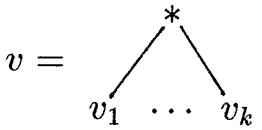
where $\beta \preceq T$. Hence, we must have



where $\beta \rhd^* v_\beta^i$. Since all the v_α's have a least upper bound v, we know that they all have the same fanout, k. Thus,



where $v_i = \bigsqcup v_\beta^i$, so $T \rhd^\omega v_i$; hence, $v \in *\text{VAL}_{\text{MAX}}(T)$. For the other inclusion we select any $v \in *\text{VAL}_{\text{MAX}}(T)$. It must be of the form



where $T \rhd^\omega v_i$, i.e. $\forall \beta \preceq T, |\beta| < \infty : \beta \rhd^* v^i_\beta$ and $\sqcup v^i_\beta = v_i$. Now, for any finite $\alpha \preceq *T$ define

$$v_\alpha = \begin{cases} \Omega & \text{if } \alpha = \Omega \\ \begin{array}{c} * \\ v^1_\beta \quad \cdots \quad v^k_\beta \end{array} & \text{if } \alpha = *\beta \end{cases}$$

Then $\alpha \rhd^* v_\alpha$ and $\sqcup v_\alpha = v$, so $v \in \textsc{ValMax}(*T)$. By a similar argument we can demonstrate that $\textsc{ValMax}((n_i : T_i)) = (n_i : \textsc{ValMax}(T_i))$. For Int, Bool, and Ω the result is obvious, so we conclude that $\textsc{ValMax}$ is homomorphic. □

Theorem 4.8: $\textsc{ValMax}$ is monotonic.
Proof: Assume $T_1 \preceq T_2$; we shall show that if $T_1 \rhd^\omega v$ then $T_2 \rhd^\omega v$. By lemma 4.5 we have that $T_2 \rhd^\omega T_1$, so from lemma 4.6 and $T_2 \rhd^\omega T_1 \rhd^\omega v$ we conclude that $T_2 \rhd^\omega v$. □

The reason for the term "maximal" is the following result.

Theorem 4.9: If ϕ is any value assignment, then

$$\forall T : \phi(T) \subseteq \textsc{ValMax}(T)$$

Proof: Suppose $v \in \phi(T)$. Let $\alpha \preceq T$ be a finite subtree. We shall define $\alpha(v) \preceq v$, the *projection* of v on α, as follows

$$\begin{aligned} \Omega(v) &= \Omega \\ \text{Int}(v) &= v \\ \text{Bool}(v) &= v \\ T\left(\begin{array}{c} * \\ v_1 \quad \cdots \quad v_k \end{array}\right) &= \begin{array}{c} * \\ T(v_1) \quad \cdots \quad T(v_k) \end{array} \\ (n_j : T_j)\left(\begin{array}{c} (n_i) \\ | \\ v_i \end{array}\right) &= \begin{array}{c} (n_k) \\ | \\ T_k(v_k) \end{array} \end{aligned}$$

where $\{n_k\} = \{n_i\} \cap \{n_j\}$. Since ϕ is homomorphic, this is a well-defined and complete case analysis. Now, an easy induction in the structure of

α shows that $\alpha \rhd^* \alpha(v)$. Since each $\alpha(v)$ is a subtree of v they form a directed set. Also, monotonicity of ϕ, in particular $\phi(\Omega) = \emptyset$, gives that any finite subtree of v equals $\alpha(v)$ for some finite $\alpha \preceq T$. We conclude that $v = \bigsqcup \alpha(v) \in \textsc{Val}_{\textsc{MAX}}(T)$. $\square$

With the value assignment $\textsc{Val}_{\textsc{MAX}}$ we can rest assured that we have found all the infinite values that could possible make sense in the context of the hierarchy.

Examples

The maximal values of

Type A = (head: Int, tail: A) ! {tail $\Rightarrow$ head}

are all finite and infinite lists of integers. In contrast, the values of

Type B = (head: Int, tail: B) ! {head $\wedge$ tail}

are only the infinite lists. In general, any infinite type will contain some infinite values, and only the vacuous type is empty. In the recursive value assignment the type B would be empty, too.

Categorical Limits

The standard way of obtaining infinite values is to resolve type equations by constructing the categorically largest fixed point, which is the limit of the chain constructed from the final object in the category [6]. This is almost, but not quite, what we have done. In our case, the value set associated with the (infinite) type defined by

Type $T = F(T)$

would be

$$\textsc{Val}_{\textsc{INF}}(T) = \lim_{n \to \infty} \tilde{F}^n(\{\bullet\})$$

where $\{\bullet\}$ is some singleton set – the final object in the category of sets. This value function is homomorphic, since all the value constructors are

continuous, but it is not monotonic. This is immediately seen, since $\text{VAL}_{\text{INF}}(\Omega) = \{\bullet\} \neq \emptyset$. Hence, we cannot use this as a value assignment. The essence of the problem is that the *protovalues* of the previous approach appear as *values* in this context – an unfortunate technical mishap. It would be possible to recover the maximal values by explicitly stripping away all the values that contain the symbol $\bullet$, but apart from the lack of elegance, it is *much* harder to verify its properties in this guise.

Proposition 4.10: If monotonicity was not an issue, then we would *not* have a largest value assignment.
Proof: The value function

$$\text{VAL}^X_{\text{INF}}(T) = \lim_{n\to\infty} \tilde{F}^n(X)$$

is homomorphic for *any* set X, and $\text{VAL}^X_{\text{INF}}(\Omega) = X$, so in particular the set of Ω-values can be arbitrarily large. □

5 Intermediate Value Assignments

So far, we have seen the two extreme value assignments VAL_{REC} and VAL_{MAX}, between which all other value assignments must be contained. At a glance, it may not be obvious that there are other possibilities, but in fact we have an infinitude of proper value assignments.

All value sets will be subsets of the maximal ones. Such a subset could be characterized in the following manner

$$\text{VAL}_\psi(T) = \{v \in \text{VAL}_{\text{MAX}}(T) \mid \psi(v)\}$$

where ψ is some predicate on trees; for example, we clearly have

$$\text{VAL}_{\text{REC}} = \{v \in \text{VAL}_{\text{MAX}}(T) \mid v \textit{ is finite}\}$$

To find other value assignments, we shall initially explore what the requirements translate to.

Definition 5.1: A predicate ψ on trees is *reducible* iff

$$\psi \text{ holds for } t \Leftrightarrow \psi \text{ holds for all proper subtrees of } t$$

Theorem 5.2: VAL_ψ is a value assignment if and only if ψ is reducible.
Proof: Monotonicity of VAL_ψ is automatically inherited from VAL_{MAX}.

The homomorphic properties

$$\begin{aligned} \textsc{Val}_\psi(\text{Int}) &= \{\ldots,-1,0,1,2,\ldots\} \\ \textsc{Val}_\psi(\text{Bool}) &= \{\text{true},\text{false}\} \end{aligned}$$

tells us that ψ must hold for all simple values. But this is equivalent to the fact that ψ is reducible on singleton trees, since ψ vacuously holds for the empty collection of proper subtrees. For non-singleton trees we have two cases. Firstly, we look at the homomorphic property

$$\textsc{Val}_\psi(*T) = *\textsc{Val}_\psi(T)$$

which translates to

$$\{v \in \textsc{Val}_{\textsc{MAX}}(*T) \mid \psi(v)\} = *\{v \in \textsc{Val}_{\textsc{MAX}}(T) \mid \psi(v)\}$$

Neither containment follows automatically, but they combine to the requirement

$$\psi\left(\begin{array}{ccc} & * & \\ \diagup & & \diagdown \\ v_1 & \cdots & v_k \end{array}\right) \Leftrightarrow \forall i : \psi(v_i)$$

Similarly, for the homomorphic property

$$\textsc{Val}_\psi((n_i : T_i)) = (n_i : \textsc{Val}_\psi(T_i))$$

we get the requirement

$$\psi\left(\begin{array}{c} (n_i) \\ | \\ v_i \end{array}\right) \Leftrightarrow \forall i : \psi(v_i)$$

By induction in the depth of subtrees it follows that $\textsc{Val}_\psi$ being homomorphic corresponds to ψ being reducible. □

Proposition 5.3: $\textsc{Val}_\psi$ is the largest value assignment such that all values satisfy ψ.
Proof: Immediate from maximality of $\textsc{Val}_{\textsc{MAX}}$. □

Examples of reducible predicates are

- If every subtree contains a 0, then every subtree contains a 1.
- Is finite.
- Is computable (in some (additive) time or space bound).

In contrast, the following predicates are not reducible

- Does not contain any 0's.
- Contains a path with infinitely many 1's.
- Is infinite.
- Is uncomputable.

Notice that it is not possible to have just the infinite values.

Characterizing Infinite Values

The reducible predicates describe value assignments in more general situations than the present one. Any programming language would presumably insist on homomorphic requirements of its type constructors, which is enough to allow modified versions of theorem 5.2 to carry through. The absence of the hierarchy would only invalidate proposition 5.3. It seems worthwhile to investigate reducible predicates further to obtain an alternative characterization of infinite values in programming languages.

Proposition 5.4: If ψ is reducible then ψ holds for all finite trees; the opposite implication is false.
Proof: ψ must hold for all singleton trees, since they have no proper subtrees. Hence, by induction ψ holds for all finite trees. To see that the opposite implication is false, just consider the predicate "is finite or has no leaves", which clearly holds for finite trees but is *not* reducible. □

This yields a very simple proof for propositions 3.4 and 3.5:

Corollary 5.5: VAL_{REC} is the smallest value assignment.
Proof: Since "is finite" is reducible VAL_{REC} is a value assignment. As any other value assignment is described by a reducible predicate it follows

from proposition 5.4 that it must contain VAL_{REC}. □

Also, the reducible predicates have some obvious closure properties.

Proposition 5.6: The class of reducible predicates is closed under $\vee$ and $\wedge$, but not $\neg$.
Proof: Clearly true for $\vee$ and $\wedge$. For $\neg$ just look at "is finite" and "is infinite". □

Probably the best way to characterize the reducible predicates is that they are *stable under finite changes.*

Definition 5.7: If $t_1, \ldots, t_n$ are trees, then t is a *finite modification* of these, if it is obtained by combining the t_i's while changing the labels of finitely many nodes, making finite insertions or deletions, or rearranging, copying or deleting finitely many (infinite) subtrees.

Definition 5.8: A predicate ψ on trees is *finitely stable* if whenever $\psi(t_1), \ldots, \psi(t_n)$ holds and t is a finite modification of the t_i's, then also $\psi(t)$ holds.

Proposition 5.9: ψ is reducible iff it is finitely stable.
Proof: Assume ψ reducible and $\psi(t)$. Then ψ holds for all subtrees of t. The finite modification t may be viewed as a new tree which contains finitely many (infinite) subtrees from the t_i's. By starting an induction at the roots of these subtrees it follows, in analogy with proposition 5.4, that $\psi(t)$ holds. Now, assume that ψ is finitely stable. If $\psi(t)$ holds, then ψ holds for any subtree, since it can be obtained as a finite modification of t. If ψ holds for all subtrees of t, then it particularly holds for the finitely many immediate subtrees of t. But t is a finite modification of these, so $\psi(t)$ holds, too. □

Even so, a reducible predicate *can* detect an infinite pattern of labels or tree-structure, as witnessed by the computability predicates.

An intuitive understanding of the situation may be given as follows:

The finite values are always present, since they can be explicitly constructed on run-time. The infinite values cannot be computed in finite time, so they must be given *a priori.* These infinite values are described by the predicate ψ. The program is now free to perform any finite mod-

ifications of the infinite values. This should not create any unexpected infinite values. Hence, reducible means: *closed under finite computation*, which is exactly what we need to have a consistent value assignment in any reasonable setting.

6 The Hierarchy Revisited

The maximal value assignment is too large for comfort. In an actual programming language, we must be able to e.g. copy and compare values in finite time, which is not possible for all the values in VAL_{MAX}. Hence, we would have to abolish these operations for *all* values, which is clearly unacceptable. A possible solution is to introduce a new unary type constructor, ∞, which indicates that infinite values are allowed, i.e.

$$\begin{aligned} \text{VAL}(T) &= \text{VAL}_{\text{REC}}(T) \\ \text{VAL}(\infty T) &= \text{VAL}_{\text{MAX}}(T) \end{aligned}$$

This is somewhat unsatisfactory, since the whole concept of the type hierarchy seems to demand that the relation

$$T \preceq \infty T$$

holds, in which case the problem reappears.

In consequence of our intention to use infinite values to initialize infinite data structures, we can certainly restrict ourselves to fewer values. We want the following properties to hold

- All values have finite representations.
- Equality of values is decidable.
- The set of values is closed under finite computation.

Decidability of equality forces us to direct our attention below the collection of all computable values.

Rational Values

An appealing choice is the collection of *regular* or *rational* values [2], which are characterized by having a finite number of *different* subtrees. Clearly,

this predicate is reducible, so by proposition 5.2 we obtain a proper value assignment by restricting ourselves to these values.

For example, rational infinite lists correspond exactly to the decimal expansion of rational numbers, i.e. a finite arbitrary prefix followed by a finite period.

Specifications

The *only* extension needed to a hierarchical language is a mechanism for specifying the infinite values; their manipulation is no different from that of finite values. The rational values can be specified by a finite set of *value equations* of the form

$$\{v_i = e_i\}$$

where each e_i is a finite constant expression, possibly including the v_j's; the equations must satisfy the *Greibach condition* that no e_i *equals* any v_j. This gives us the finite representation; also, equality of rational values is decidable.

Proposition 6.1: Value equations specify exactly the set of rational, maximal values.
Proof: In [2] it is shown that such equations generate exactly all rational trees. Since, "is rational" is reducible it follows from theorems 5.2 and 4.9 that all rational values are maximal. □

We can achieve this quite elegantly by exploiting the idea that infinite values are used for initializations. Variables are defined as follows

$$\begin{aligned}&\textbf{Var } x_1 : T_1\\ &\textbf{Var } x_2 : T_2\\ &\quad\vdots\\ &\textbf{Var } x_n : T_n\end{aligned}$$

Initially, all variables are undefined; only when they are assigned values is an appropriate structure of subvariables allocated. We extend this to

$$\begin{aligned}&\textbf{Var } x_1 : T_1 = e_1\\ &\textbf{Var } x_2 : T_2 = e_2\end{aligned}$$

$$\vdots$$

Var $x_n : T_n = e_n$

where each e_i is a constant expression of type T_i, possibly involving the x_j, but not equal to any single x_j. In all cases we have initialized the variables with finite or infinite values. Notice that *any* rational value may be expressed as an initial value.

With the type definition

Type L = (head: Int,tail: L) ! {tail ⇒ head}

the variable definitions

Var a: L = (head: 0, tail: a)
Var b: L = (head: 1, tail: c)
Var c: L = (head: 2, tail: b)

will initialize a to an infinite list of 0's and b and c to infinite lists of alternating 1's and 2's with different parity.

The definitions

Type Tree = (val: Int, left,right: Tree) ! { left ∧ right }
Var t: Tree = (left,right: t)

allocates an infinite binary tree without values in the nodes. These may be freely added on later, as the programmer may assume that an infinite number of nodes have been allocated. An infinite tree with all nodes initialized to zero is specified by

Var t: Tree = (val: 0, left,right: t)

Implementation

A possible implementation technique is as follows:

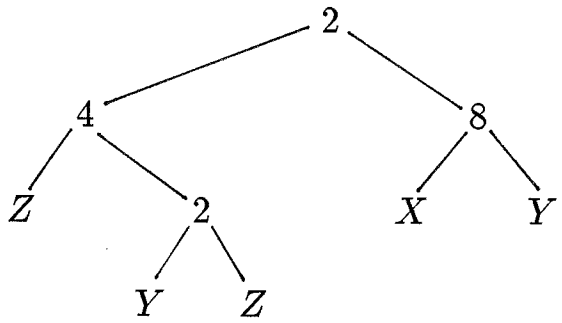
A value is a tree, which is in fact implemented using records and pointers. Products and lists are stored in blocks of words of the appropriate length. Each word contains either a simple value of type Int or Bool, or it contains a pointer to a block containing a structured subvalue.

Infinite values are, of course, implemented lazily, i.e. at any point of the execution only a finite prefix of the value has been allocated. This corresponds to the internal nodes of the pointer tree. The leaves are special nodes containing the periodic subvalues in a "rolled-up" representation, which is just a single variable symbol in a set of equations defining the value.

As an example, we may look at the following tree values

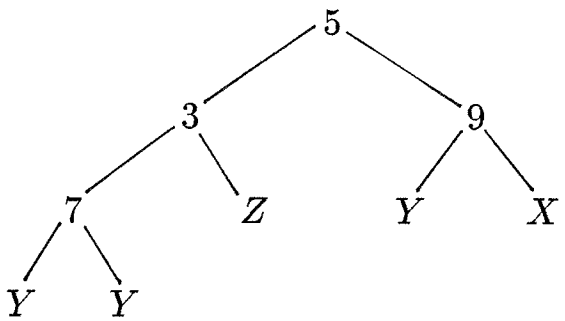
$$\begin{aligned} X &= (\text{val} : 2, \text{left} : Y, \text{right} : Z) \\ Y &= (\text{val} : 4, \text{left} : Z, \text{right} : X) \\ Z &= (\text{val} : 8, \text{left} : X, \text{right} : Y) \end{aligned}$$

A tree variable with the infinite value X could, after some unfolding, look like



Here 6 infinite subtrees remain rolled-up.

Notice that once a node has been allocated and initialized one may modify its contents in the usual imperative fashion (including passing it as a reference parameter), thereby changing the total value. Hence, by assigning to some val-components and re-arranging a few subtrees we can obtain the variable



The defining equations for this value would be rather more complicated, but fortunately we need never concern ourselves with those.

These finite representations of infinite values obviously allow copying and even external I/O-operations.

Equality of infinite values is decidable using a simple algorithm, the execution time of which depends on the sizes of the finite representations as well as on the number and sizes of the defining equations.

References

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

[2] **Courcelle B.** "Fundamental Properties of Infinite Trees" in *Theoretical Computer Science Vol 25 No 1*, North-Holland 1983.

[3] **Henderson, P.** "Functional Programming: Application and Implementation". Prentice-Hall, 1980.

[4] **Turner, D.A.** "Recursion Equations as a Programming Language" in *Functional Programming and Its Application*, CUP 1982.

[5] **O'Donnell M.J.** "Computing in Systems Defined by Equations" in *LNCS Vol 58*, Springer-Verlag 1977.

[6] **Panangaden P., Mendler N. & Schwartzbach, Michael I.** "Recursively Defined Types in Constructive Type Theory" in *Resolution of Equations in Algebraic Structures* eds. Hassan Ait-Kaci & Maurice Nivat, Academic Press 1989.

[7] **Schmidt, Erik M. & Schwartzbach, Michael I.** "An Imperative Type Hierarchy with Partial Products" in *Proceedings of Mathematical Foundations of Computer Science 1989, LNCS Vol 379*, Springer-Verlag, 1989.

[8] **Cartwright, R., Donahue, J.** "The Semantics of Lazy (and Industrius) Evaluation" in *Proceedings of ACM Symposium on* LISP *and Functional Programming 1982.*