Interpretations of Recursively Defined Types

Michael I. Schwartzbach

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

Abstract

We consider a type system where types are labeled, regular trees. Equipped with a type ordering, it forms the basis for a polymorphic, imperative programming language. This paper studies *interpretations*, which are homomorphic, monotonic functions from types to sets of values. We show that they form a partial order with a minimal and a maximal element, and various characterizations of other interpretations are provided. We also briefly consider a unification of types and values.

1 Introduction

In [7,9] we introduced a type system for an imperative programming language. The types are labeled, regular trees containing tree-shaped data values. There is a natural partial order of types, defined as a refinement of the standard approximation order on labeled trees. This type ordering is shown to allow a subtype polymorphism that encompasses 1st order parametric polymorphism and multiple inheritance. In fact, under various assumptions one can show that this system allows optimal code reuse.

The values of a given type have been defined as the least solutions to the recursive equations on sets of trees induced from the type equations. These

values are all finite trees. In this paper we investigate the full spectrum of possible *interpretations* of types, given that the polymorphic mechanism must remain correct. This is equivalent to classifying functions from types to sets of trees that satisfy certain homomorphic and monotonic axioms.

We show that such interpretations form a partial order, with a minimal and a maximal element. A similar distinction between small and large interpretations of (more general) recursive type equations is discussed in [2,6]. Thus, any interpretation can be expressed as a restriction of the maximal interpretation, described by some predicate on values. We characterize the predicates that yield legal interpretations; they must be *decomposable*, or equivalently *finitely stable*.

The variety of interpretations contains many interesting possibilities, e.g. finite, regular, and computable values. Infinite values constitute an important part of many programming languages, when they can be implemented lazily.

Finally, these investigations suggest that we may dissolve the distinction between types and values, and work with a unified concept.

2 Recursively Defined Types

In this section we shall briefly review the type system of [7,9]. Types are defined by means of a set of *type equations*

$$\begin{array}{l}\textbf{Type } T_1 = E_1\\ \textbf{Type } T_2 = E_2\\ \ldots\\ \textbf{Type } T_k = E_k\end{array}$$

where the T_i's are *type variables* and the E_i's are *type expressions*, which are defined as follows

$E ::=$	Int $\mid$ Bool $\mid$	simple types
	$T_i \mid$	type variables
	$*E \mid$	lists
	$(n_1 : E_1, \ldots, n_k : E_k)$	partial products, $k \geq 0$, $n_i \in \mathcal{N}$, $n_i \neq n_j$

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 as part of the type expression 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

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

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.

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

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; at the theoretical worst, the size of the notation may expand exponentially. Secondly, the nesting of sums

and products force components belonging to the same *conceptual* level to appear at different *syntactical* levels. The partial products alleviate these disadvantages.

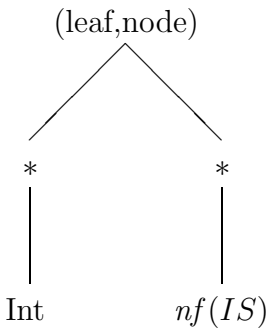
Another approach could be to employ *more* type constructors. We can think of the partial product as the Cartesian product of domain-like sets with a $\perp$ element to indicate undefinedness. In [1] a great number of binary domain constructors are considered for the purposes of specifying various domains 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 at the same time we 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.

Types as Regular Trees

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, this 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



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 unique limits of chains of approximations, as discussed in [3,4]. The singleton tree Ω is smaller than all other trees and corresponds to the type defined by

$$\textbf{Type } T = T$$

which we shall refer to as the *vacuous* type. Note that if two type expressions are equivalent, then their corresponding structural invariants must be equal.

The equivalence $\approx$ is unique in satisfying the following properties: no two type expressions 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.

Type equivalence is decidable; an efficient algorithm is presented in [8].

We let $\mathcal{T}$ denote the set of all types, i.e. normal forms of type expressions. The notation $labels(t)$ denotes the set of labels in t. Notice that all types have finite label sets. We shall write $t' \sqsubset\!\!\sqsubset t$ to denote that t' is finite and $t' \sqsubseteq t$.

The Type Ordering

To obtain our polymorphism 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, including assignments. 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.

The approximation ordering $\sqsubseteq$ may itself serve as a type ordering. However, it can 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 (due to the partiality) formation of constants.

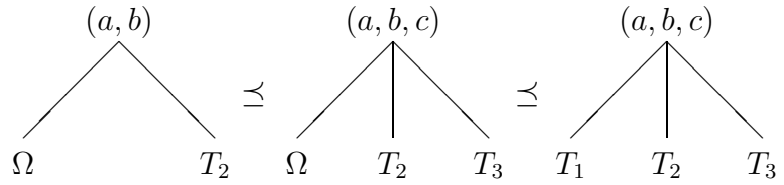
The complete type ordering is obtained in two stages. We first define $\preceq_0$ as the least refinement of $\sqsubseteq$ such that

$$(n_i : T_i) \preceq_0 (m_j : S_j) \;\; \textit{if} \;\; \{n_i\} \subseteq \{m_j\} \;\wedge\; n_i = m_j \Rightarrow T_i \preceq_0 S_j$$

Extending this to infinite types, we define $\preceq$ in terms of finite approximants

$$T \preceq S \;\; \textit{iff} \;\; \forall T' \sqsubset\!\sqsubset T : \; T' \preceq_0 S$$

To illustrate this ordering, we can observe that the relations



hold for all T_i.

In the presence of structural invariants the ordering is somewhat more complicated; for products with invariants we define

$$P_1 \,!\, I_1 \preceq_0 P_2 \,!\, I_2 \;\; \textit{iff} \;\; P_1 \preceq_0 P_2 \;\wedge\; I_1 \preceq I_2$$

where the ordering on invariants is defined by

$$I_1 \preceq I_2 \;\; \textit{iff} \;\; I_1 \subseteq I_2 \;\wedge\; I_1 | I_2$$

The *divisibility* $I_1|I_2$ is defined in [7] as

$$I_1 \times (I_2/I_1) \subseteq I_2$$

where

$$X \times Y = \{x \cup y \,|\, x \in X, y \in Y\}, \;\; X/Y = \{x - \textit{basis}(Y) \,|\, x \in X\}$$

and $\textit{basis}(Y)$ is the set of components names in the product with which Y is associated. This expresses a consistency condition on invariants. For this presentation we need only to know that the ordering implies inclusion of invariants.

In [7] it is described how this ordering can achieve a subtype polymorphism for a PASCAL-like language with assignments. The parameter passing mechanism is extended to exploit the order structure, essentially by allowing the actual parameters to be larger than the formal parameters, subject to certain homogeneity conditions. The semantics of such a *hierarchical* procedure call is to *substitute* the types of the actual parameters for those of the formal parameters, *recompile* the procedure, and execute a normal procedure call[1]. In [9] we introduce a general example language and provide a proof of soundness and optimality of this system.

The least upper bounds of $\preceq$ correspond to *multiple inheritance* [7]: 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. Dually, greatest lower bounds correspond to *multiple specialization* [7]. Least upper bounds may or may not exist, whereas greatest lower bounds always exist.

This type system, together with hierarchical procedure calls, allows the combination of multiple inheritance and full 1st order polymorphism in a language with assignments.

The type ordering is decidable, and least upper bounds as well as greatest lower bounds are computable. Efficient algorithms are presented in [8].

[1] Of course, an actual implementation would employ a uniform data representation that allowed direct code reuse.

3 Interpretations

In this setting we also regard the *values* as being labeled trees. With each type T we shall associate a set of values.

Definition 3.1: An *interpretation* ϕ is a function from types to sets of values that satisfies the following axioms. It must be *homomorphic* in the sense that

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

where the type constructors have analogous *value constructors* defined below. Furthermore, ϕ must be *monotonic* with respect to $\preceq$; that is

$$\forall\, T_1, T_2 \in \mathcal{T} :\ T_1 \preceq T_2 \Rightarrow \phi(T_1) \subseteq \phi(T_2)$$

The value constructors are defined as follows. Let S, S_i be sets of values. Then

$$*S = \left\{ \begin{array}{c} * \\ \diagup\ \cdots\ \diagdown \\ s_1 \qquad\quad s_k \end{array} \;\middle|\; k \geq 0, s_i \in S \right\}$$

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

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

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

The axioms are needed for the correctness proofs in [9] to be valid, but they are quite easy to motivate on their own. Homomorphicity simply states 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. Monotonicity states the intended meaning of the type ordering.

Proposition 3.2: The value constructors are all $\subseteq$-monotonic and ω-continuous functions on sets of values.
Proof: Immediate, since they are all syntactic operators, i.e. they merely combine trees in their entirety. □

Values of Finite Types

The homomorphic and monotonic axioms are fairly severe, but as we shall see they allow for more than one interpretation. It is, however, the case that all interpretations must agree on all *finite* types.

Definition 3.3: When T is finite, then $\textsc{Val}_{\textsc{fin}}(T)$ is defined inductively in T as follows

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

$$\begin{aligned} \textsc{Val}_{\textsc{fin}}(*T) &= *\textsc{Val}_{\textsc{fin}}(T) \\ \textsc{Val}_{\textsc{fin}}((n_i : T_i)\,!\,I) &= (n_i : \textsc{Val}_{\textsc{fin}}(T_i))\,!\,I \end{aligned}$$

Proposition 3.4: $\textsc{Val}_{\textsc{fin}}$ is homomorphic and monotonic.
Proof: It is by definition homomorphic. Monotonicity follows since the value constructors are monotonic, $\emptyset$ is smaller than all other values set, and a partial product clearly gets more values when more components are added or the invariant is increased. □

Proposition 3.5: If ϕ is any interpretation and T is finite, then $\phi(T) = \textsc{Val}_{\textsc{fin}}(T)$.
Proof: For any interpretation ϕ, monotonicity implies $\phi(\Omega) = \emptyset$, since $\Omega \preceq \text{Int}$, $\Omega \preceq \text{Bool}$ and hence $\phi(\Omega) \subseteq \phi(\text{Int}) \cap \phi(\text{Bool}) = \emptyset$. Now the result follows by straightforward structural induction. □

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

Recursive Values

Definition 3.6: The recursive interpretation of the (infinite) type given by the equation

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

is defined as

$$\textsc{Val}_{\textsc{rec}}(T) = \bigcup_{i \geq 0} \widetilde{F}^i(\emptyset)$$

where $\widetilde{F}$ is the (composite) value constructor derived from the (composite) type constructor F. This is the standard construction of the least fixed point,

which generalizes in the obvious way to mutually recursive type equations.

Proposition 3.7: $\textsc{Val}_{\mathrm{REC}}$ is an interpretation.
Proof: The homomorphic axioms are satisfied since the two value constructors are ω-continuous functions on sets. Regarding monotonicity, we may initially observe that

$$\textsc{Val}_{\mathrm{REC}}(T) = \bigcup_{S \preceq T,\, |S|<\infty} \textsc{Val}_{\mathrm{FIN}}(S)$$

This follows from the facts that all the $F^n(\Omega)$ are finite types, that $\textsc{Val}_{\mathrm{FIN}}$ is monotonic, and that any finite $S \preceq T$ is smaller than some $F^n(\Omega)$. Now, if $T_1 \preceq T_2$ then

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

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

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

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

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

Proposition 3.8: If ϕ is any interpretation, then

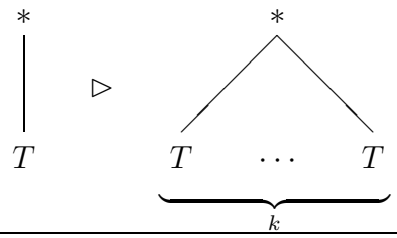
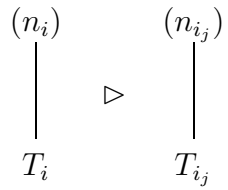
$$\forall\, T \in \mathcal{T} :\ \textsc{Val}_{\mathrm{REC}}(T) \subseteq \phi(T)$$

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

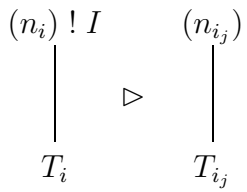
4 The Maximal Interpretation

It is perhaps more surprising that we can find a *maximal* interpretation. Using the fact that both types and values are labeled trees, we shall define a rewriting system which transforms a type into any of its values.

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

I	$T \quad \triangleright \quad \Omega$	
II		$k \geq 0$
III		$\{n_{i_j}\} \subseteq \{n_i\}$
IV	$\text{Int} \quad \triangleright \quad i$	$i \in \text{Int}$
V	$\text{Bool} \quad \triangleright \quad b$	$b \in \text{Bool}$

For products with invariants we have the modified rewrite rule:

VI		$\{n_{i_j}\} \in I$

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

Proposition 4.1: For all finite T we have

$$\textsc{Val}_{\text{FIN}}(T) = \{v \mid T \rhd^* v \ \wedge \ \text{Int}, \text{Bool}, \Omega \notin \mathit{labels}(v)\}$$

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 securing the appropriate fan-out using the II-rule and then inductively expanding the S-subtrees. Similarly, if $T = (n_i : S_i)$. □

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 rewriting steps. This is, in fact, possible in the present context, since we can work with finite approximants.

Definition 4.2: $t \rhd^\omega v$ *iff* $\forall \beta \sqsubset v \, \exists \alpha \sqsubset t : \ \alpha \rhd^* \beta$ A similar method for defining *functions* is described in [3].

Definition 4.3: The maximal interpretation is defined by

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

which mimics proposition 4.1.

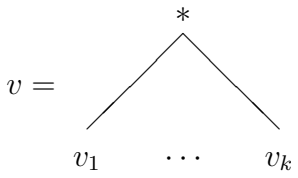
Lemma 4.4: If $t_1 \preceq t_2$ then $t_2 \rhd^\omega t_1$.
Proof: If $\beta \sqsubset t_1$, then $\beta \sqsubset t_2$. Since $\beta \rhd^* \beta$, we are done. □

Lemma 4.5: $\rhd^\omega$ is reflexive and transitive.

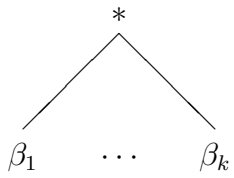
Proof: Reflexivity $t \rhd^\omega t$ holds since $\forall \beta \sqsubset t : \ \beta \rhd^* \beta$. For transitivity we assume $t \rhd^\omega s \rhd^\omega r$, so that $\forall \beta \sqsubset s \, \exists \alpha \sqsubset t : \ \alpha \rhd^* \beta$ and $\forall \gamma \sqsubset r \, \exists \beta \sqsubset s : \ \beta \rhd^* \gamma$. But then $\forall \gamma \sqsubset r \, \exists \beta \sqsubset s \, \exists \alpha \sqsubset t : \ \alpha \rhd^* \beta \rhd^* \gamma$. By transitivity of $\rhd^*$ we have $\forall \gamma \sqsubset r \, \exists \alpha \sqsubset t : \ \alpha \rhd^* \gamma$, so $t \rhd^\omega r$. $\square$

Now we can show that the maximal interpretation satisfies the required axioms.

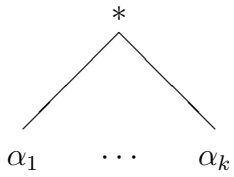
Theorem 4.6: $\textsc{Val}_{\textsc{MAX}}$ is homomorphic.
Proof: Clearly, $\textsc{Val}_{\textsc{MAX}}$ works correctly on Int, Bool, and Ω. Suppose $v \in *\textsc{Val}_{\textsc{MAX}}(T)$. Then



where $v_i \in \textsc{Val}_{\textsc{MAX}}(T)$, i.e. $\forall \beta_i \sqsubset v_i \, \exists \alpha_i \sqsubset T : \ \alpha_i \rhd^* \beta_i$. Any $\beta \sqsubset v$ is either Ω or of the form

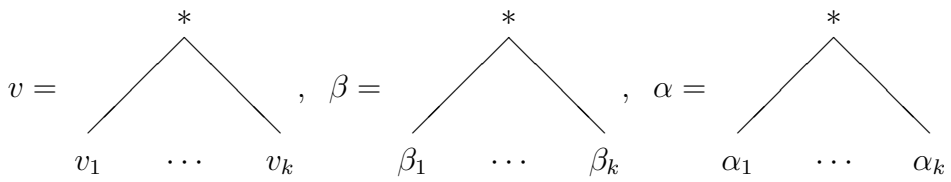


Thus, if we choose α as Ω or



we have $\forall \beta \sqsubset v \, \exists \alpha \sqsubset *T : \ \alpha \rhd^* \beta$. Hence, $\textsc{Val}_{\textsc{MAX}}(*T) \supseteq *\textsc{Val}_{\textsc{MAX}}(T)$. Conversely, if $v \in \textsc{Val}_{\textsc{MAX}}(*T)$, then $\forall \beta \sqsubset v \, \exists \alpha \sqsubset *T : \ \alpha \rhd^* \beta$. Since v

cannot have an Ω-label, we have



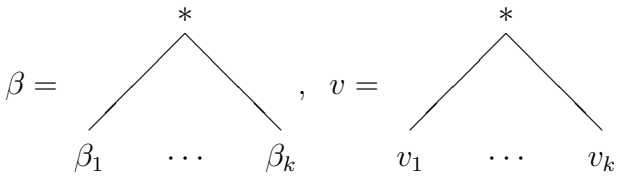
where $\alpha_i \rhd^* \beta_i \sqsubset v_i$. Since β_i is arbitrary, we have $\forall \beta_i \sqsubset v_i \, \exists \alpha_i \sqsubset T : \alpha_i \rhd^* \beta_i$. Hence, $\textsc{Val}_{\textsc{MAX}}(*T) \subseteq *\textsc{Val}_{\textsc{MAX}}(T)$. The result for $(n_i : T_i)$ is proved similarly. $\Box$

Theorem 4.7: $\textsc{Val}_{\textsc{MAX}}$ 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 4.4 we have that $T_2 \rhd^\omega T_1$, so from lemma 4.5 and $T_2 \rhd^\omega T_1 \rhd^\omega v$ we conclude that $T_2 \rhd^\omega v$. $\Box$

Theorem 4.8: If ϕ is any interpretation, then

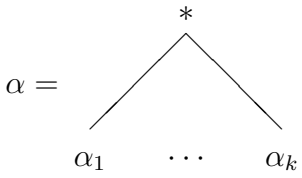
$$\forall T \in \mathcal{T} : \; \phi(T) \subseteq \textsc{Val}_{\textsc{MAX}}(T)$$

Proof: If $T = \Omega$, then $\phi(T) = \emptyset$ and we are done. Otherwise, let $v \in \phi(T)$ and $\beta \sqsubset v$. By induction in β, we shall construct $\alpha \sqsubset T$ such that $\alpha \rhd^* \beta$. If $\beta = \Omega$, then $\alpha = \Omega$ will do. If β is a simple value, then T is the corresponding simple type and $\alpha = T$ will do. If $T = *S$ then



where $\beta_i \sqsubset v_i \in \phi(S)$. By induction hypothesis, we can find $\alpha_i \sqsubset S$ such

that $\alpha_i \rhd^* \beta_i$. But then



will do. We proceed similarly if $\beta = (n_i : \beta_i)$. □

For example, the maximal values of

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

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

Type B = (head: Int, tail: B) ! {head ∧ tail}

are only the infinite lists. In general, any infinite type will contain some infinite values, and only Ω is empty. An infinite type that is empty under $\textsc{Val}_{\text{REC}}$ will exclusively have infinite values under $\textsc{Val}_{\text{MAX}}$, as exemplified by the type B.

The following result shows that if monotonicity was not an axiom, then we would *not* have a maximal interpretation.

Proposition 4.12: If only the homomorphic axioms are required, then interpretations can be arbitrarily large.
Proof: The value function

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

is homomorphic for *any* set X, due to ω-continuity of the value constructors. We have $\textsc{Val}^X_{\text{INF}}(\Omega) = X$, so in particular the sets of Ω-values are unbounded. □

5 Other Interpretations

So far, we have seen the two extreme interpretations, $\textsc{Val}_{\textsc{REC}}$ and $\textsc{Val}_{\textsc{MAX}}$, between which all others 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 interpretations.

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

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

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

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

Obviously, not all predicates will yield legal interpretations; we shall characterize the ones that do.

Definition 5.1: A predicate ψ on trees is *decomposable* when

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

Thus, truth for t can be decomposed into truth for the subtrees.

Theorem 5.2: $\textsc{Val}_{\psi}$ is an interpretation *iff* ψ is decomposable.
Proof: Monotonicity of $\textsc{Val}_{\psi}$ is automatically inherited from $\textsc{Val}_{\textsc{MAX}}$. The homomorphic properties on simple types tells us that ψ must hold for all simple values. But this is equivalent to the fact that ψ is decomposable on singleton trees, since ψ vacuously holds for the empty collection of proper subtrees. For non-singleton trees we have two cases. First, 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

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

Similarly, for the homomorphic property

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

we get the requirement

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

This also works if invariants are employed. By induction in the depth of subtrees it follows that $\textsc{Val}_\psi$ being homomorphic corresponds to ψ being decomposable. □

Proposition 5.3: $\textsc{Val}_\psi$ is the largest interpretation under which all values satisfy ψ.
Proof: Immediate from maximality of $\textsc{Val}_{\textsc{max}}$. □

Examples of decomposable predicates are

a) Is finite.

b) Is regular.

c) Is computable (in some additive time or space bound).

d) If every subtree contains a $*$-label, then every subtree contains an (x)-label.

The case d) will serve as a counterexample in a later result. In contrast, the following predicates are not decomposable

e) Is infinite.

f) Is uncomputable.

g) Contains a path with infinitely many $*$-labels.

h) Does not contain any $*$-labels.

i) Contains a $*$-label.

The examples h) and i) show that a predicate and its negation can both be not decomposable.

The decomposable predicates provide a convenient method for defining interpretations, since it is fairly easy to determine if predicates can be decomposed. It is tempting to believe that that the decomposable predicates form exactly the theory of finite trees, but this is not so.

Proposition 5.4: If ψ is decomposable, 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, since they trivially satisfy the first disjunct, but which is *not* decomposable. □

This provides a very simple proof of propositions 3.7 and 3.8.

Corollary 5.5: VAL_{REC} is the smallest interpretation.
Proof: Since "is finite" is decomposable VAL_{REC} is an interpretation. As any other interpretation is described by a decomposable predicate it follows from proposition 5.4 that it contains VAL_{REC}. □

Proposition 5.6: The class of decomposable predicates is closed under conjunction, but not under disjunction or negation.
Proof: Clearly true for $\wedge$. For $\vee$, we look at the two decomposable predicates b) and d) mentioned above. The unary tree t_1 with labels $* * * * * * \cdots$ clearly satifies b). The unary tree t_2 with labels $*(x) * (x)^2 * (x)^3 * \cdots$ clearly satisfies d). However, a tree with subtrees t_1, t_2 does not satisfy b) $\vee$ d), since t_1 has no (x)-label and t_1 is not regular. For $\neg$, we consider "finite" and "infinite". $\Box$

Probably the best way to characterize the decomposable predicates is to observe that they are *stable under finite computations.*

Definition 5.7: If $t_1, \ldots, t_n, t$ are trees, then we write

$$t_1, t_2, \ldots, t_n \mapsto t$$

if only finitely many subtrees of t are not also subtrees of some t_i. We call t a *finite modification* of the t_i's.

The intuition behind this definition is that t is the result of a finite computation with the t_i's as input. One can combine the t_i's while changing the labels of finitely many nodes, making finite insertions or deletions, or rearranging, copying or deleting finitely many subtrees.

Definition 5.8: A predicate ψ on trees is *finitely stable* if whenever $\psi(t_1), \ldots, \psi(t_n)$ holds and $t_1, t_2, \ldots, t_n \mapsto t$, then also $\psi(t)$ holds.

Proposition 5.9: ψ is decomposable *iff* it is finitely stable.
Proof: Assume that ψ decomposable and $\psi(t)$ holds. Then ψ holds for all subtrees of t. Then ψ holds for all but finitely many subtrees of t, and by induction ψ must hold for the rest, too. 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. $\Box$

Even so, a decomposable 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 finite modifications of the infinite values. This should not create any unexpected infinite values.

6 Unifying Types and Values

In the preceding development, types and values are both labeled trees. This suggests that we may be able to dissolve the distinction between them.

Proposition 6.1: $\rhd^\omega$ is anti-symmetric.
Proof: Suppose that $t \rhd^\omega s$ and $s \rhd^\omega t$. Let $\beta \sqsubset s$. Then we have an $\alpha \sqsubset t$ such that $\alpha \rhd^* \beta$. We also have a $\beta' \sqsubset s$ such that $\beta' \rhd^* \alpha \rhd^* \beta$. By a simple inductive argument it follows that $\alpha \sqsubset s$. By symmetry, it follows that t and s have the same finite approximants and, thus, are equal. $\Box$

This, together with lemma 4.5, shows that $\lhd^\omega$ (as well as $\rhd^\omega$ itself) is a partial order. It provides a unifying characterization of (sub)types and values in the following sense

- if v is a value of type T then $v \lhd^\omega T$
- if S is a subtype of T then $S \lhd^\omega T$

A similar observation has been the basis for a unification of *terms* and *sorts* in algebraic specifications [5].

We can now combine values, protovalues, and types. The "values" of "type" T is

$$\{v \mid T \rhd^\omega v\}$$

and the "type" ordering is $\lesssim$. A hybrid such as

$$(x : \text{Bool}, y : 7)$$

can be viewed simultaneously as a product type with an x-component of type Bool and a y-component that must equal 7, *and* as a product value where the y-component is 7 and the x-component is an undetermined Bool value.

Such a unification radically changes the basis for the polymorphic mechanism, but it is possible to obtain results similar to those in [8,9].

One advantage of such an approach is the ability to deal with undetermined values. It also solves the problem of how to initialize a variable of type T: the natural initial value is just T itself. Since types are regular trees, it would be natural to restrict "values" similarly, which is possible since "is regular" is a decomposable predicate. This provides finite representations for a lazy implementation, and makes equality and $\lesssim$ decidable.

References

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

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

[3] **Courcelle B.** "Fundamental Properties of Infinite Trees" in *Theoretical Computer Science Vol 25 No 1, 95-169*, North-Holland 1983.

[4] **Courcelle B.** "Infinite Trees in Normal Form and Recursive Equations Having a Unique Solution" in *Mathematical Systems Theory 13, 131-180.* Springer-Verlag 1979.

[5] **Mosses P.D.** "Unified Algebras and Institutions" in *Proceedings of Fourth Annual Symposium on Logic in Computer Science, 304-312*, 1989.

[6] **Panangaden P. Mendler N. & Schwartzbach, M.I.** "Recursively Defined Types in Constructive Type Theory" in *Resolution of Equations in Algebraic Structures Vol 1, 369-410* eds. Hassan Ait-Kaci & Maurice Nivat, Academic Press 1989.

[7] **Schmidt, E.M. & Schwartzbach, M.I.** "An Imperative Type Hierarchy with Partial Products" in *Proceedings of MFCS'89*, *LNCS Vol 379, 458-470*, Springer-Verlag, 1989.

[8] **Schwartzbach, M.I. & Schmidt, Erik M.** "Types and Automata". *PB-3165*, Department of Computer Science, Aarhus University, 1990.

[9] **Schwartzbach, M.I.** "Static Correctness of Hierarchical Procedures" in *Proceedings of ICALP'90, LNCS Vol 443, 32-45*, Springer-Verlag, 1990.