To appear in *Theoretical Computer Science*.

Also in Proc. ICALP'90, pages 32-45.

Static Correctness of Hierarchical Procedures

Michael I. Schwartzbach

Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Århus C, Denmark
mis@daimi.aau.dk

Abstract

A system of hierarchical, fully recursive types in a truly imperative language allows program fragments written for small types to be reused for all larger types. To exploit this property to enable type-safe hierarchical procedures, it is necessary to impose a *static requirement* on procedure calls. We introduce an example language and prove the existence of a *sound* requirement which preserves static correctness while allowing hierarchical procedures. This requirement is further shown to be *optimal*, in the sense that it imposes as few restrictions as possible. This establishes the theoretical basis for a general type hierarchy with static type checking, which enables 1st order polymorphism combined with multiple inheritance and specialization in a language with assignments.

We extend the results to include *opaque* types. An opaque version of a type is different from the original but has the same values and the same order relations to other types. The opaque types allow a more flexible polymorphism and provide the usual pragmatic advantages

of distinguishing between intended and unintended type equalities. Opaque types can be viewed as a compromise between *synonym* types and *abstract* types.

1 Introduction

This paper develops a subtype polymorphism for an imperative language with assignments, subvariables, variable parameters, and fully recursive types. It combines multiple inheritance with simple implicit parametric polymorphism.

The idea is to allow hierarchical procedure calls, where the types of the actual parameters are larger than those of the formal parameters. The claim is that if a program is statically correct, then this mechanism will preserve correctness. This is only true if the procedure call avoids some blatant inconsistencies by maintaining a homogeneous choice of larger actual types. We must introduce a *static requirement*, which is a rule that determines the legality of procedure calls.

In section 2 we introduce the types. They are represented as unordered, regular trees. The type ordering is defined as a refinement of the Böhm ordering. Section 3 presents the example language. It is similar to PASCAL, except that it employs the richer types presented in section 2. In section 4 we observe the apparent coincidence that procedures seem to remain type correct when the types of their formal parameters are increased. This is the inspiration to the later introduction of hierarchical procedure calls. Section 5 contains a formal definition of static correctness of programs. The correctness requirements are carefully expressed using a very small set of relations on types. The concept of extended types is introduced; these play the role of type schemes, summarizing the possible types of polymorphic expressions. The exact requirement for the legality of procedure calls is left as an unspecified predicate on the types of formal and actual parameters. Section 6 observes that such a requirement need only satisfy a few simple soundness rules to imply type correctness for programs. This inspires a search for the most liberal such requirement. A candidate is proposed, and a lengthy proof is given for its soundness and optimality. In section 7 various extensions to the language are considered. It is shown that local variables can be handled, whereas global variables must be abandonded. Section 8 describes a technique for making the hierarchical calls more flexible by introducing opaque versions of types. Technically, this is simply a matter of moving form a partial order on types to a preorder. All previous results generalize without much difficulty.

2 The Types

The type system allows dynamic, recursive types, but it is still intended to be employed by a standard imperative language, where variables containing structured values are composed of a similar structure of subvariables.

Types are defined by means of a set of *type equations*

$$\textbf{Type } N_i = \tau_i$$

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

τ ::=	Int | Bool |	simple types
	N_i |	type names
	$*\tau$ |	lists
	$(n_1 : \tau_1, \ldots, n_k : \tau_k)$	partial products, $k \geq 0$, $n_i \neq n_j$

Here the n_i's are names. 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 values of ordinary products may be regarded as *total* functions.

The partiality of the product will prove essential to the correctness of the hierarchy. Furthermore, partial products yield a pragmatically advantageous notation for specifying recursive types, in particular when combined with the notion of *structural invariants*. Details are presented in [7].

The values of types may be taken to be the $\subseteq$-least solutions to the corresponding equations on sets induced by the above interpretation of the type constructors. Other interpretations of types are possible; for example, one may include *infinite (lazy) values*. The variety of different interpretations is investigated in [8].

2.1 Type Equivalence

Several type expressions may be taken to denote the same *type*. These can be identified by an equivalence relation $\approx$, which is defined as the identity of *normal forms*, using the techniques of [4,5]. To each type expression T we associate a unique normal form $nf(T)$, which is a possibly-infinite labeled tree. Informally, the tree is obtained by repeatedly *unfolding* the type expression. Formally, we use the fact that the set of labeled trees form a *complete partial order* under the partial ordering 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

$$\textbf{Type } N = N$$

We shall write Ω to denote any such type expression.

2.2 Type Ordering

The hierarchical ordering is a refinement of $\sqsubseteq$. We want to include orderings between partial product types, such that $(n_i : T_i) \preceq (m_j : S_j)$ *iff*

$$\{n_i\} \subseteq \{m_j\} \text{ and } (\forall i, j : \; n_i = m_j \Rightarrow T_i \preceq S_j)$$

This possibility must extend to infinite types as well. If $\preceq_0$ is this inductive (finite) refinement of $\sqsubseteq$, then the full ordering is defined as

$$S \preceq T \;\Leftrightarrow\; \forall S' \sqsubseteq S,\; |S'| < \infty :\; S' \preceq_0 T$$

Thus, products with fewer components are smaller than products with more components. As noted in [7], trees under this ordering no longer form a cpo. However, all the chains definable by type equations still have limits.

Facts 2.1:

- Ω is the smallest type.
- The type constructors are monotonic and continuous.
- If $T = F(T)$ is a type equation, then $\Omega \preceq F(\Omega) \preceq F^2(\Omega) \preceq \cdots \preceq F^i(\Omega) \preceq \cdots$ is a chain with limit T.
- If $T_1 \preceq T_2$, then all values of type T_1 are also values of type T_2.
- Greatest lower bounds $\sqcap$ always exist.
- Least upper bounds $\sqcup$ may or may not exist.
- All of $\approx$, $\preceq$, $\sqcap$ and $\sqcup$ are computable.

The least upper bounds of $\preceq$ correspond to the constructive aspect of *multiple inheritance*: 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. The greatest lower bounds correspond to *general specialization*: the maximal common subtype of two types can be constructed.

3 An Example Language

The results we present will be valid in any standard imperative language which employs our type system and exploits its ramifications. In order to provide a rigorous framework for stating and proving these results, we shall introduce a modest example language. Hopefully, it will be apparent that all major results will carry over to richer languages without significant modifications. The language is presented by means of its syntax and its informal semantics.

3.1 Syntax

The principal syntactic categories are: statements (S), variables (σ), expressions (ϕ), declarations (D), types (τ), and programs (P). In the following

grammar the symbols N, P, n_i, x range over arbitrary names, and k is any non-negative number.

$$
\begin{array}{lcl}
\mathrm{S} & ::= & \sigma\texttt{:=}\phi \mid \\
 & & \sigma\texttt{:-}n_i \mid \\
 & & \sigma\texttt{:+(}n_i\texttt{:}\phi\texttt{)} \mid \\
 & & P\texttt{(}\phi_1\texttt{,}\ldots\texttt{,}\phi_k\texttt{)} \mid \\
 & & \textbf{if } \phi \textbf{ then } \mathrm{S} \textbf{ end} \mid \\
 & & \textbf{while } \phi \textbf{ do } \mathrm{S} \textbf{ end} \mid \\
 & & \mathrm{S}_1 \texttt{ ; } \mathrm{S}_2 \\
 & & \\
\sigma & ::= & x \mid \sigma\texttt{.}n_i \mid \sigma\texttt{[}\phi\texttt{]} \\
 & & \\
\phi & ::= & 0 \mid \phi\texttt{+1} \mid \phi\texttt{-1} \mid \\
 & & \sigma \mid \\
 & & \phi_1 \texttt{ = } \phi_2 \mid \\
 & & \texttt{[}\phi_1\texttt{,}\ldots\texttt{,}\phi_k\texttt{]} \mid \\
 & & \texttt{|}\phi\texttt{|} \mid \\
 & & \texttt{(}n_1\texttt{:}\phi_1\texttt{,}\ldots\texttt{,}n_k\texttt{:}\phi_k\texttt{)} \mid \\
 & & \textbf{has}\texttt{(}\phi\texttt{,}n_i\texttt{)}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{D} & ::= & \textbf{Type } N \texttt{ = } \tau \mid \\
 & & \textbf{Proc } P\texttt{(}\rho\ x_1\texttt{:}\tau_1\texttt{,}\ldots\texttt{,}\rho\ x_k\texttt{:}\tau_k\texttt{)}\ \mathrm{S}\ \textbf{end}\ P \mid \\
 & & \textbf{Var } x\texttt{:}\tau \\
 & & \\
\rho & ::= & \textbf{var} \mid \textbf{val} \\
 & & \\
\tau & ::= & \mathrm{Int} \mid \mathrm{Bool} \mid \\
 & & N \mid \\
 & & \texttt{*}\tau \mid \\
 & & \texttt{(}n_1\texttt{:}\tau_1\texttt{,}\ldots\texttt{,}n_k\texttt{:}\tau_k\texttt{)} \\
 & & \\
\mathrm{P} & ::= & \mathrm{D}_1\ \mathrm{D}_2\ \ldots\ \mathrm{D}_k\ \mathrm{S}
\end{array}
$$

3.2 Informal Semantics

Most of the language is quite standard: simple expressions, variables, assignments, comparisons, control structures and procedures with variable- or value parameters. There are static scope rules, but global variables may not be accessed from within procedures. As shown in section 7.1 this is a necessary restriction; however, it does not seriously limit the generality of the language, since global variables can be explicitly passed as variable parameters.

The partial product acts as a partial function where $\sigma\texttt{:-}n_i$ removes n_i from the domain of σ, $\sigma\texttt{:+(}n_i\texttt{:}\phi\texttt{)}$ updates σ with the value ϕ on n_i, and $\textbf{has}\texttt{(}\phi\texttt{,}n_i\texttt{)}$ decides whether n_i is in the domain of ϕ. Arbitrary partial product constants can be denoted by $\texttt{(}n_1\texttt{:}\phi_1\texttt{,}\ldots\texttt{,}n_k\texttt{:}\phi_k\texttt{)}$. A subvariable of a partial product

may be selected by $\sigma.n_i$ (provided it is in the domain of σ). A list constant is denoted by $[\phi_1,\ldots,\phi_k]$, and the subvariable with index ϕ is selected by $\sigma[\phi]$ (provided σ has length greater than ϕ). The expression $|\phi|$ denotes the length of the list ϕ.

4 Motivating Hierarchal Procedures

This section will provide an intuitive motivation for the proposed type hierarchy, and it will point out the various difficulties that we must overcome.

The prime motivation is the observation that many procedure calls seem to work fine if the types of actual parameters are larger than those of the formal parameters. In the following examples we compare pairs of procedures, where we increase the sizes of the formal parameters. In all cases we observe that the procedure body can remain unchanged.

```
Proc P(var x,y:Ω,val z:Ω)
    x:=z;
    y:=z
end P

Proc Q(var x:*Ω,val y:Ω)
    if |x| = 0 then
       x:=[y,y,y]
    end
end Q

Proc R(var x:(a:Ω,b:Bool))
    x.b:=has(x,a)
end R
```

```
Proc P(var x,y:Int,val z:Int)
    x:=z;
    y:=z
end P

Proc Q(var x:*Bool,val y:Bool)
    if |x| = 0 then
       x:=[y,y,y]
    end
end Q

Proc R(var x:(a:Int,b:Bool,c:()))
    x.b:=has(x,a)
end R
```

This opens up for a very direct version of code reuse, where the left-hand procedure can emulate the right-hand one by enlarging the types of its formal parameters during a so-called *hierarchical* procedure call. Then Ω works as

a type parameter and inheritance is enabled by the partial product aspect of the type ordering.

There have been many suggestions for languages with a similar subtype polymorphism. Ours is unique in allowing truly imperative features such as assignments, subvariables, and variable parameters. Many systems rely on *coercions* [1,2,4,6] which have distinct disadvantages such as *type loss* and the *update problem* [2]. We avoid these; for example, the procedure

```
Proc Id(var x: Ω)
    skip
end Id
```

will be the identity on both the type and the value of any argument. The presence of variables or *mutable types* [1,2] have so far lead to unsafe type systems, unless the subtype ordering is trivialized in this case. A system which operationally is more similar to ours is that of *type extensions* [10]. However, several important issues are not addressed, leading to various anomalies. For example, just allowing actual parameters to have larger types than formal parameters is too liberal an attitude. We must have a *homogeneous* choice of larger types, as the following example shows. The procedure

```
Proc P(var x: Ω, val y: Ω)
    x:=y
end P
```

will not be correct if the actual type of x is Int and the actual type of y is Bool. We must limit the permitted procedure calls to avoid such blatant inconsistencies. Also, it is not clear that more subtle problems cannot occur with this mechanism; for example, the behavior of recursive types or nested levels of (recursive) hierarchical calls must also be considered. In the following sections we shall provide the necessary framework for supplying a formal proof for the validity of these ideas. This will establish a firm basis for exploiting this useful mechanism without any risk of inconsistencies or anomalies.

5 Static Correctness

In a programming language *static correctness* is a decidable syntactic property of program texts. When all programs are guaranteed to be statically correct, then one can verify certain invariant properties of the execution model. These invariants are crucial for reasoning about program correctness. They are also very useful for developing efficient implementations and performing compile-time optimizations. Typically, static correctness implies such basic properties as: variables of type T can only contain values of type T, operations are only performed on arguments of the proper types, etc.

In this section we shall define static correctness of our programs. To facilitate this we need a number of auxiliary concepts.

5.1 Environments

Correctness will be defined relativly to an *environment*, which is a finite map from (variable) names to types.

Definition 5.1: If σ is a variable and $\mathcal{E}$ is an environment, then $\mathcal{E}\downarrow\sigma$ denotes a type defined as follows

- $\mathcal{E}\downarrow x = \mathcal{E}(x)$ if $x \in \mathrm{dom}(\mathcal{E})$
- $\mathcal{E}\downarrow\sigma\,\texttt{[}\phi\texttt{]} = T$ if $\mathcal{E}\downarrow\sigma = *T$
- $\mathcal{E}\downarrow\sigma\,.n_i = T_i$ if $\mathcal{E}\downarrow\sigma = (n_1 : T_1, \ldots, n_k : T_k)$

We write $\sigma \in \mathcal{E}$ if $\mathcal{E}\downarrow\sigma$ denotes a type according to the above schema. □

Definition 5.2: If $\mathcal{E}$ and $\mathcal{E}'$ are environments, then

$$\mathcal{E} \preceq \mathcal{E}' \text{ iff } \mathrm{dom}(\mathcal{E}) = \mathrm{dom}(\mathcal{E}') \ \wedge\ \forall x :\ \mathcal{E}(x) \preceq \mathcal{E}'(x)$$

□

5.2 Extended Types

We have some polymorphic constants in the language; for example, `[]` denotes the empty list of *any* type, and the constant `(b:87)` can have any type which is a partial product with at least a b-component of type Int.

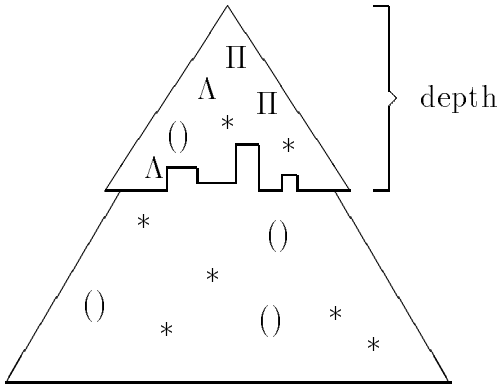
It will prove technically convenient to make this polymorphism explicit by defining an extension of our type system.

Notation 5.3: We introduce the abbreviation $(a_i : A_i)$ instead of the more explicit $(a_1 : A_1, \ldots, a_k : A_k)$. The value of k is implicit and is *not* assumed to be the same in e.g. $(a_i : A_i)$ and $(b_j : B_j)$. □

Definition 5.4: The *x-types* are extensions of the types defined as follows

$$
\begin{array}{lll}
X ::= & \tau \mid & \text{(any type)} \\
& *X \mid & \\
& \Lambda \mid & \\
& \Pi(n_1 : X_1, \ldots, n_k : X_k) &
\end{array}
$$

An x-type is really a type *scheme* that defines a set of types with similar structure. Types can be *elements* of x-types. Any type of the form $*T$ is an element of Λ, and the elements of $\Pi(n_i : X_i)$ are all partial products with at least the components $(n_i : T_i)$, where T_i is an element of X_i. In general, an x-type is a (possibly infinite) tree; however, the x-type *depth*, which is the largest depth of Λ and Π nodes, is always finite, as seen in the following illustration



□

Definition 5.5: (Compatibility) The relation $X_1 \bowtie X_2$ holds *iff* the x-types X_1 and X_2 have at least one element in common. □

Proposition 5.6: $\bowtie$ is the smallest symmetric relation which satisfies

- $T_1 \bowtie T_2$, if $T_1 = T_2$
- $\Lambda \bowtie \Lambda$
- $\Lambda \bowtie *X$
- $*X_1 \bowtie *X_2$ *iff* $X_1 \bowtie X_2$
- $(n_i : T_i) \bowtie \Pi(m_j : Y_j)$ *iff* $\{m_j\} \subseteq \{n_i\} \ \wedge \ (\forall i, j : \ n_i = m_j \Rightarrow T_i \bowtie Y_j)$
- $\Pi(n_i : X_i) \bowtie \Pi(m_j : Y_j)$ *iff* $(\forall i, j : \ n_i = m_j \Rightarrow X_i \bowtie Y_j)$

Proof: Induction in the largest x-type depth. □

Definition 5.7: (Unification) If $X_1 \bowtie X_2$, then $X_1 \otimes X_2$ denotes the unique x-type whose elements are exactly the common elements of X_1 and X_2. Clearly, $\otimes$ is associative and commutative (when defined). □

Proposition 5.8: Whenever its arguments are related by $\bowtie$, then $\otimes$ can be computed as follows

- $T_1 \otimes T_2 = T_1$, if $T_1 = T_2$ are types
- $\Lambda \otimes \Lambda = \Lambda$
- $\Lambda \otimes *X = *X$
- $*X_1 \otimes *X_2 = *(X_1 \otimes X_2)$
- $(n_i : T_i) \otimes \Pi(m_j : Y_j) = (n_i : T_i)$
- $\Pi(n_i : X_i) \otimes \Pi(m_j : Y_j) = \Pi(z_k : Z_k)$ where $\{z_k\} = \{n_i\} \cup \{m_j\}$ and

$$Z_k = \begin{cases} X_i & \text{if } z_k = n_i \notin \{m_j\} \\ X_i \otimes Y_j & \text{if } z_k = n_i = m_j \\ Y_j & \text{if } z_k = m_j \notin \{n_i\} \end{cases}$$

Proof: Induction in the largest x-type depth. □

Proposition 5.9: If $X_1 \bowtie X_2$, then $(X_1 \otimes X_2) \bowtie Y \Leftrightarrow X_1 \bowtie Y \wedge X_2 \bowtie Y$
Proof: $\Rightarrow$ is immediate. For $\Leftarrow$ we use induction in the largest x-type depth in X_i.

- If both X_i are types, then $X_1 = X_2 = X_1 \otimes X_2$ and we are done.
- If e.g. $X_1 = \Lambda$, then $X_1 \otimes X_2 = X_2$ and we are done.
- If $X_i = *Z_i$ then we have two cases: 1) if $Y = \Lambda$, then we are done; 2) if $Y = *Z$, then we use the induction hypothesis.
- If $X_1 = (n_i : T_i)$ and $X_2 = \Pi(m_j : Z_j)$, then $X_1 \otimes X_2 = X_1$ and we are done.
- If $X_1 = \Pi(n_i : Y_i)$ and $X_2 = \Pi(m_j : Z_j)$, then the result follows by induction on the common components. □

Definition 5.10: (Sufficiency) The relation $S \triangleleft X$ states that there is an element of the x-type X which is larger than the type S. □

Proposition 5.11: $\triangleleft$ is the smallest relation which satisfies

- $S \triangleleft T$, if T is a type and $S \preceq T$

- $\Omega \triangleleft X$
- $*S \triangleleft *X$ *iff* $S \triangleleft X$
- $*S \triangleleft \Lambda$
- $(n_i : S_i) \triangleleft \Pi(m_j : X_j)$ *iff* $(\forall i, j : n_i = m_j \Rightarrow S_i \triangleleft X_j)$

Proof: Induction in the structure of X. $\square$

Proposition 5.12: $S \triangleleft X_1 \otimes X_2 \Leftrightarrow S \triangleleft X_1 \wedge S \triangleleft X_2$
Proof: $\Rightarrow$ is immediate. For $\Leftarrow$ we proceed by induction in the length of the largest x-type depth in X_i.

- If both X_i are types, then $S \triangleleft X_1 = X_2 = X_1 \otimes X_2$.
- If e.g. $X_1 = \Lambda$, then $X_1 \otimes X_2 = X_2$.
- If $X_i = *Y_i$, then $S = *T$ and $T \triangleleft Y_i$ so by induction hypothesis $T \triangleleft Y_1 \otimes Y_2$ so $*T \triangleleft *(Y_1 \otimes Y_2) = X_1 \otimes X_2$.
- If $X_1 = (n_i : T_i)$ and $X_2 = \Pi(m_j : Y_j)$, then $X_1 \otimes X_2 = X_1$.
- If $X_1 = \Pi(n_i : Y_i)$ and $X_2 = \Pi(m_j : Z_j)$, then the result follows by induction on the common components. $\square$

Proposition 5.13: All of: $\bowtie$, $\otimes$ and $\triangleleft$ are computable.
Proof: Immediate from decidability of $\approx$ and $\preceq$, the finite depth of x-types, and propositions 5.6, 5.8, and 5.11. $\square$

5.3 Defining Correctness

To specify the correctness conditions we need to talk about the *types* of expressions. These are, as previously stated, not unique, but we can assign a unique x-type $\mathcal{E}[\![\phi]\!]$ to each expression ϕ relative to an environment $\mathcal{E}$.[1] The

[1] In this context it is more convenient to use the "denotational" notation $\mathcal{E}[\![\phi]\!] = X$ rather than the "inferential" notation $\mathcal{E} \vdash \phi : X$.

elements of $\mathcal{E}[\![\phi]\!]$ are exactly the possible types of ϕ.

Definition 5.14: If $\mathcal{E}$ is an environment and ϕ is an expression, then $\mathcal{E}[\![\phi]\!]$ is defined inductively as follows

- $\mathcal{E}[\![0]\!] = \mathcal{E}[\![\phi\texttt{+}1]\!] = \mathcal{E}[\![\phi\texttt{-}1]\!] = \text{Int}$
- $\mathcal{E}[\![\sigma]\!] = \mathcal{E}\downarrow\sigma$
- $\mathcal{E}[\![\phi_1 \texttt{ = } \phi_2]\!] = \text{Bool}$
- $\mathcal{E}[\![\texttt{[}\phi_1\texttt{,}\ldots\texttt{,}\phi_k\texttt{]}]\!] = *(\otimes_i \mathcal{E}[\![\phi_i]\!])$, if $k > 0$
- $\mathcal{E}[\![\texttt{[]}]\!] = \Lambda$
- $\mathcal{E}[\![\texttt{|}\phi\texttt{|}]\!] = \text{Int}$
- $\mathcal{E}[\![\texttt{(}n_i\texttt{:}\phi_i\texttt{)}]\!] = \Pi(n_i : \mathcal{E}[\![\phi_i]\!])$
- $\mathcal{E}[\![\textbf{has}\texttt{(}\phi\texttt{,}n_i\texttt{)}]\!] = \text{Bool}$

A program will only be correct when all such denotations are well-defined. □

Definition 5.15: We present a predicate CORRECT$(\mathcal{E}, \text{S})$ which says that the statement S is statically correct with respect to the environment $\mathcal{E}$. The predicate is described as a list of conditions on phrases: variables, expressions and statements. These conditions must be true for all such phrases in the syntactic derivation of S.

	Phrase:	Condition:
1)	σ	$\sigma \in \mathcal{E}$
2)	σ`[`ϕ`]`	$\mathcal{E}[\![\phi]\!] \bowtie \mathrm{Int}$
3)	σ`:=` ϕ	$\mathcal{E}[\![\sigma]\!] \bowtie \mathcal{E}[\![\phi]\!]$
4)	σ`:-`n_i	$(n_i : \Omega) \triangleleft \mathcal{E}[\![\sigma]\!]$
5)	σ`:+(`n_i`:`ϕ`)`	$(n_i : \Omega) \triangleleft \mathcal{E}[\![\sigma]\!] \ \wedge \ \mathcal{E}[\![\sigma\,.\,n_i]\!] \bowtie \mathcal{E}[\![\phi]\!]$
6)	ϕ`+1`, ϕ`-1`	$\mathcal{E}[\![\phi]\!] \bowtie \mathrm{Int}$
7)	ϕ_1 `=` ϕ_2	$\mathcal{E}[\![\phi_1]\!] \bowtie \mathcal{E}[\![\phi_2]\!]$
8)	`[`ϕ_1`,...,`ϕ_k`]`	$\forall i, j : \ \mathcal{E}[\![\phi_i]\!] \bowtie \mathcal{E}[\![\phi_j]\!]$
9)	`|`ϕ`|`	$*\Omega \triangleleft \mathcal{E}[\![\phi]\!]$
10)	**has**`(`ϕ`,`n_i`)`	$(n_i : \Omega) \triangleleft \mathcal{E}[\![\phi]\!]$
11)	**if** ϕ **then** S **end**	$\mathcal{E}[\![\phi]\!] \bowtie \mathrm{Bool}$
12)	**while** ϕ **do** S **end**	$\mathcal{E}[\![\phi]\!] \bowtie \mathrm{Bool}$
13)	P`(`ϕ_1`,...,`ϕ_k`)`	$\exists \mathcal{A} : \ \mathcal{E}[\![\phi_i]\!] \bowtie \mathcal{A}(x_i) \ \wedge \ \textsc{req}(\mathcal{F}, \mathcal{A})$

For the procedure call we used a few abbreviations. The procedure looks like

Proc P`(`$\rho\ x_1$`:`τ_1`,...,`$\rho\ x_k$`:`τ_k`)`
    S
end P

Now, $\mathcal{F}$ is the *formal* environment mapping x_i to τ_i, whereas $\mathcal{A}$ is the *actual* environment which maps x_i to an appropriate actual type compatible with ϕ_i.

Finally, REQ is the *static requirement*, which is in fact the main topic of this paper. It is a predicate on the formal and actual environments, which determines the permitted degree of hierarchical procedure calls. To get an ordinary language we can use the requirement

$$\textsc{equal}(\mathcal{F}, \mathcal{A}) \equiv (\mathcal{F} = \mathcal{A})$$

which insists that the formal and actual parameter types must be equivalent.

The entire program is statically correct when all statements are correct relative to their *environments*. The environment for the main program consists of

the global variables, and the environment for a procedure body is its formal parameters; thus, global variables are not accessible from within procedures.

Also, we must include various static conditions which are independent of the environment, such as a systematic use of names and bindings, and the fact that actual variable parameters must be variables. □

5.4 Dynamic Aspects

If we use the requirement EQUAL, then the definition of static correctness should be uncontroversial. Examples of invariants are: values of type T can only reside in variables of type T, list operations are only performed on lists, and operations involving a product component n_i are only allowed if the type in fact contains such a component. The polymorphic constants are allowed to remain undetermined as long as it can be assured that they can be assigned a sensible type.

6 Hierarchical Correctness

By relaxing the static requirement we allow some procedure calls where the actual types are larger than the formal types. The semantics of a hierarchical call is to *substitute* the actual types for the formal types, *recompile* the procedure and perform a *normal* procedure call.[2]

This raises some concerns about the static correctness. We may have ensured that the body of the procedure was correct with respect to the *formal* environment, but now it will be executed in a different *actual* environment. Consequently, the requirement must possess a special quality.

Definition 6.1: A static requirement REQ is *sound* if

[2] An implementation would, of course, employ a uniform data representation that allows it to reuse code without further ado.

- $\forall \mathrm{S}, \mathcal{F}, \mathcal{A}:$ CORRECT$(\mathcal{F}, \mathrm{S}) \wedge$ REQ$(\mathcal{F}, \mathcal{A}) \Rightarrow$ CORRECT$(\mathcal{A}, \mathrm{S})$
- Condition 13) in definition 5.15 is decidable

Thus, correctness must be preserved by a sound requirement, and the static correctness conditions must remain decidable. □

Proposition 6.2: The requirement EQUAL is sound.
Proof: If $\mathcal{F} = \mathcal{A}$, then clearly correctness is preserved. Condition 13) is decidable, since we have only one possible $\mathcal{A}$ for which we must check that $\mathcal{E}[\![\phi_i]\!] \bowtie \mathcal{A}(x_i)$, which is the same as $\mathcal{E}[\![\phi_i]\!] \bowtie \tau_i$. Hence, the types of the actual parameters are required to match those of the formal parameters, which is what we would expect in this normal situation. □

Soundness has very important consequences for the dynamic aspects of static correctness. If we verify static correctness for all parts of a program, then we expect certain invariants to hold during execution. With hierarchical calls this property is no longer immediate, but if the static requirement is sound, then the execution invariants will still hold. This can be established essentially by induction in the length of the dynamic chain of procedure calls. If we have length 0, then no hierarchical calls have been performed and we are safe. For longer chains we can perform the induction step by appealing to the facts that the actual parameters satisfy the static requirement and that the soundness condition holds.

6.1 An Optimal Sound Requirement

We shall prove the existence of a *sound* requirement which is *optimal*, in the sense that it is minimally restrictive and, hence, allows as many hierarchical calls as possible.

Definition 6.3: ALL is a static requirement defined by

$$\mathrm{ALL}(\mathcal{F}, \mathcal{A}) \equiv (\mathcal{F} \preceq \mathcal{A}) \wedge (\forall \sigma, \sigma' : \mathcal{F}\downarrow\sigma = \mathcal{F}\downarrow\sigma' \Rightarrow \mathcal{A}\downarrow\sigma = \mathcal{A}\downarrow\sigma')$$

□

Theorem 6.4: ALL is sound.

To prove this main result we must show that all the static correctness conditions are preserved and that condition 13) is decidable.

Lemmas 6.5 through 6.7 show preservation of the basic conditions. We assume $\textsc{all}(\mathcal{F}, \mathcal{A})$.

Lemma 6.5: If $\sigma \in \mathcal{F}$, then $\sigma \in \mathcal{A}$ and $\mathcal{F}\downarrow\sigma \preceq \mathcal{A}\downarrow\sigma$.
Proof: Induction in σ. Assume $\sigma \in \mathcal{F}$. If σ is a name, then we are done since $\mathcal{F} \preceq \mathcal{A}$. Now, assume the result holds for σ. Look at $\sigma.n_i$. Since $\sigma.n_i \in \mathcal{F}$, then $\mathcal{F}\downarrow\sigma = (n_i : T_i)$ and $\mathcal{F}\downarrow\sigma.n_i = T_i$. But as $\mathcal{F}\downarrow\sigma \preceq \mathcal{A}\downarrow\sigma$, then $\mathcal{A}\downarrow\sigma = (m_j : S_j)$ where $\{n_i\} \subseteq \{m_j\}$ and $n_i = m_j \Rightarrow T_i \preceq S_j$. Hence, $\sigma.n_i \in \mathcal{A}$ and $\mathcal{F}\downarrow\sigma.n_i = T_i \preceq S_j = \mathcal{A}\downarrow\sigma.m_j = \mathcal{A}\downarrow\sigma.n_i$. For $\sigma[\phi]$ we have that $\mathcal{F}\downarrow\sigma = *T$ and $\mathcal{F}\downarrow\sigma[\phi] = T$. Since $\mathcal{F}\downarrow\sigma \preceq \mathcal{A}\downarrow\sigma$, then $\mathcal{A}\downarrow\sigma = *S$ where $T \preceq S$. Hence, $\mathcal{F}\downarrow\sigma[\phi] = T \preceq S = \mathcal{A}\downarrow\sigma[\phi]$. □

Lemma 6.6: If $\mathcal{F}[\![\phi_1]\!] \bowtie \mathcal{F}[\![\phi_2]\!]$, then $\mathcal{A}[\![\phi_1]\!] \bowtie \mathcal{A}[\![\phi_2]\!]$.
Proof: Induction in largest depth of an x-type in $\mathcal{F}[\![\phi_i]\!]$. Assume $\mathcal{F}[\![\phi_1]\!] \bowtie \mathcal{F}[\![\phi_2]\!]$:

- If $\mathcal{F}[\![\phi_i]\!]$ both are types, then we have three cases: 1) if both ϕ_i's are variables, then we are done because of soundness; 2) if neither is a variable, then they both have the same simple type in any environment; 3) if only one is a variable, then the other has a simple type, e.g. $\mathcal{F}[\![\phi_1]\!] = \text{Int}$. By lemma 6.5 $\mathcal{F}[\![\phi_1]\!] \preceq \mathcal{A}[\![\phi_1]\!]$, so $\mathcal{A}[\![\phi_1]\!] = \text{Int}$, and we are done.
- If $\mathcal{F}[\![\phi_1]\!] = \mathcal{F}[\![\phi_2]\!] = \Lambda$, then $\phi_1 = \phi_2 = [\,]$, so $\mathcal{A}[\![\phi_1]\!] = \mathcal{A}[\![\phi_2]\!] = \Lambda$.
- If $\mathcal{F}[\![\phi_1]\!] = \Lambda$ and $\mathcal{F}[\![\phi_2]\!] = *X$, then $\phi_2 = [\psi_1, \ldots, \psi_k]$ and $X = \otimes_i(\mathcal{F}[\![\psi_i]\!])$. Hence, $\mathcal{A}[\![\phi_1]\!] = \Lambda$ and $\mathcal{A}[\![\phi_2]\!] = *Y$ where $Y = \otimes_i(\mathcal{A}[\![\psi_i]\!])$.

- If $\mathcal{F}[\![\phi_1]\!] = *X$ and $\mathcal{F}[\![\phi_2]\!] = *Y$, then $\phi_1 = [\psi_1, \ldots, \psi_k]$ and $\phi_2 = [\theta_1, \ldots, \theta_k]$, where $X = \otimes_i(\mathcal{F}[\![\psi_i]\!])$, $Y = \otimes_j(\mathcal{F}[\![\theta_j]\!])$ and $X \bowtie Y$. Using proposition 5.9, $\mathcal{F}[\![\psi_i]\!] \bowtie \mathcal{F}[\![\theta_j]\!]$, so by induction hypothesis $\mathcal{A}[\![\psi_i]\!] \bowtie \mathcal{A}[\![\theta_j]\!]$, so $\otimes_i(\mathcal{A}[\![\psi_i]\!]) \bowtie \otimes_j(\mathcal{A}[\![\theta_j]\!])$ and we are done.
- If $\mathcal{F}[\![\phi_1]\!] = (n_i : T_i)$ and $\mathcal{F}[\![\phi_2]\!] = \Pi(m_j : Y_j)$, then $\{m_j\} \subseteq \{n_i\}$ and $n_i = m_j \Rightarrow T_i \bowtie Y_j$. Here $\phi_1 = \sigma$ and $\phi_2 = (m_j : \psi_j)$, so $\mathcal{F}[\![\sigma.n_i]\!] \bowtie \mathcal{F}[\![\psi_j]\!]$. By induction hypothesis $\mathcal{A}[\![\sigma.n_i]\!] \bowtie \mathcal{A}[\![\psi_j]\!]$, and we can reverse the argument. Notice that by lemma 6.5 $\mathcal{A}[\![\phi_1]\!]$ will have all the necessary $\{n_i\}$-components.
- If $\mathcal{F}[\![\phi_1]\!] = \Pi(n_i : X_i)$ and $\mathcal{F}[\![\phi_2]\!] = \Pi(m_j : Y_j)$, then we proceed by induction on the common components. □

Lemma 6.7: If $S \triangleleft \mathcal{F}[\![\phi]\!]$, then $S \triangleleft \mathcal{A}[\![\phi]\!]$.
Proof: Induction in ϕ. Assume $S \triangleleft \mathcal{F}[\![\phi]\!]$:

- If $\mathcal{F}[\![\phi]\!]$ is Int or Bool, then $\mathcal{F}[\![\phi]\!] = \mathcal{A}[\![\phi]\!]$ and we are done.
- If $\phi = \sigma$, then $\mathcal{F}[\![\phi]\!]$ is a type and by lemma 6.5 $\mathcal{F}[\![\phi]\!] \preceq \mathcal{A}[\![\phi]\!]$. Since $\triangleleft$ is $\preceq$ on types the result follows by transitivity.
- If $\phi = []$, then $\mathcal{F}[\![\phi]\!] = \mathcal{A}[\![\phi]\!] = \Lambda$.
- If $\phi = [\phi_1, \ldots, \phi_k]$, then $\mathcal{F}[\![\phi]\!] = *(\otimes_i \mathcal{F}[\![\phi_i]\!])$. Hence, $S = *T$ where $T \triangleleft \otimes_i(\mathcal{F}[\![\phi_i]\!])$, so (using proposition 5.12) $T \triangleleft \mathcal{F}[\![\phi_i]\!]$. By hypothesis $T \triangleleft \mathcal{A}[\![\phi_i]\!]$, so (using lemma 6.6 and proposition 5.12) $T \triangleleft (\otimes_i \mathcal{A}[\![\phi_i]\!])$ and $S = *T \triangleleft *(\otimes_i \mathcal{A}[\![\phi_i]\!]) = \mathcal{A}[\![\phi]\!]$.
- If $\phi = (m_j : \phi_j)$, then $\mathcal{F}[\![\phi]\!] = \Pi(m_j : \mathcal{F}[\![\phi_j]\!])$. Hence, $S = (n_i : S_i)$ and $n_i = m_j$ implies $S_i \preceq \mathcal{F}[\![\phi_j]\!]$. By hypothesis $S_i \preceq \mathcal{A}[\![\phi_j]\!]$, so $S = (n_i : S_i) \triangleleft (m_j : \mathcal{A}[\![\phi_j]\!]) = \mathcal{A}[\![\phi]\!]$. □

Lemmas 6.9 and 6.11 will establish the decidability of condition 13).

Notation 6.8: If X is an x-type, then a *type address* in X is a sequence $\gamma \in \{n_i, [\,]\}^*$ which may specify a path from the root to a subtree. The

branch from $*X$ to X is selected by $[\,]$, and the branch from $(n_i : X_i)$ or $\Pi(n_i : X_i)$ to X_i is selected by n_i. We use $\gamma \in X$ to indicate that γ leads to a subtree of X, which we will call $X \downarrow \gamma$. $\Box$

Lemma 6.9: Let $F \preceq A$ be types. Then

$$\forall \alpha, \beta : \; F \downarrow \alpha = F \downarrow \beta \; \Rightarrow A \downarrow \alpha = A \downarrow \beta$$

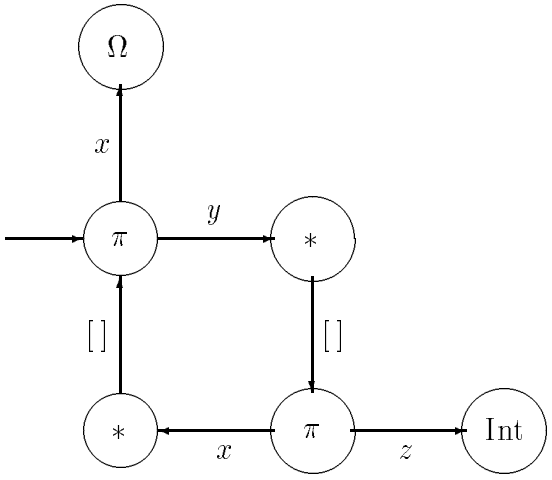
is decidable.

Proof: Any type T is a regular tree with only finitely many *different* subtrees. Hence, we can construct a deterministic, finite automaton M_T on type addresses with one state for each different subtype, such that on input $\gamma \in T$ the automaton M_T will reach the state that corresponds to the subtree $T \downarrow \gamma$. Every state accepts. Each state is labeled with the *coarse* type of the corresponding type: π for products, $*$ for lists, and Int,Bool for the simple types. There is a natural isomorphism between such automata and type equations. The above decision problem translates to a variation of language inclusion for which an efficient algorithm is presented in [9]. $\Box$

Example 6.10: The type T defined by the equations

Type $A = (x : B, y : C)$
Type $B = \Omega$
Type $C = *D$
Type $D = (x : E, z : F)$
Type $E = *A$
Type $F = \text{Int}$

corresponds to the automaton



Lemma 6.11: Condition 13) is decidable.
Proof: We first observe that without loss of generality we need only look at the case with a single parameter, since the full complexity of the problem returns if the parameter type is a product. Hence, we talk about *the* formal type τ and *the* actual x-type $\mathcal{E}[\![\phi]\!]$.

Our first test is whether $\tau \triangleleft \mathcal{E}[\![\phi]\!]$. If not, then no $\mathcal{A}$ exists; otherwise, we proceed.

We call a type address in τ *short* if it indicates a non-type (an x-type) in $\mathcal{E}[\![\phi]\!]$ and *long* if it indicates a type. We begin by computing the *finite* set of short addresses. We shall test the condition in three stages.

Stage 1 (short/long): For each short α we determine if there is a long β such that $\tau \downarrow \alpha = \tau \downarrow \beta$. This can be done by constructing the automaton mentioned in the proof of lemma 6.9 and checking if the equivalence class containing α has a sufficiently long β. If this is the case, then we need to have $\mathcal{E}[\![\phi]\!] \downarrow \alpha \bowtie \mathcal{E}[\![\phi]\!] \downarrow \beta$. If not, then no $\mathcal{A}$ exists; otherwise, we proceed. We can safely replace $\mathcal{E}[\![\phi]\!] \downarrow \alpha$ with $\mathcal{E}[\![\phi]\!] \downarrow \beta$, since this is the only element which can possibly work (this changes the address α from short to long, and in stage 2 will shall test if this element in fact does work). We continue this stage until all short/long combinations have been eliminated.

Stage 2 (long/long): Collect all maximal subtypes in τ that have long addresses and collect the corresponding subtypes in $\mathcal{E}[\![\phi]\!]$. Using lemma 6.9 we can determine if the condition holds for all long/long combinations. If not, then no $\mathcal{A}$ exists; otherwise, we proceed.

Stage 3 (short/short): We are left with the finitely many short/short combinations. We verify for each such α, β that if $\tau \downarrow \alpha = \tau \downarrow \beta$, then $\mathcal{E}[\![\phi]\!] \downarrow \alpha \bowtie \mathcal{E}[\![\phi]\!] \downarrow \beta$. If not, then no $\mathcal{A}$ exists; otherwise, we can find a common element for each set of pairwise $\bowtie$ x-types. Due to proposition 5.11 this common element can be chosen to be larger than the formal type.

After these three stages we know that an $\mathcal{A}$ exists. The only addresses in τ that we have not considered are the ones that are *invalid* in $\mathcal{E}[\![\phi]\!]$. Since $\tau \triangleleft \mathcal{E}[\![\phi]\!]$, all such invalid addresses are blocked in either Λ-nodes, or in Π-nodes with too few explicit components. Thus, the x-type $\mathcal{E}[\![\phi]\!]$ allows any types to complete the actual type in these places. Other type addresses in τ may impose several constraints, but from the above three stages we know that a common choice can be made. $\Box$

Lemmas 6.12 through 6.15 will show preservation of condition 13).

Lemma 6.12: If $\mathcal{E}[\![\phi]\!]$ is a type and $\gamma \in \mathcal{E}[\![\phi]\!]$, then there is an expression $\phi\!\downarrow\!\gamma$ such that if $\mathcal{E} \preceq \mathcal{E}'$, then $\mathcal{E}'[\![\phi\!\downarrow\!\gamma]\!] = \mathcal{E}'[\![\phi]\!] \downarrow \gamma$.
Proof: Induction in ϕ.

- If the type $\mathcal{E}[\![\phi]\!]$ is simple, then γ is empty and $\phi\!\downarrow\!\gamma = \phi$.
- If ϕ is a variable, then we can choose $\phi\!\downarrow\!\gamma = \phi\,.\,\tilde{\gamma}$, where $\tilde{\gamma}$ is a translation of γ to subvariable selections.
- If $\phi = [\phi_1, \ldots, \phi_k]$, then at least one $\mathcal{E}[\![\phi_i]\!]$ is a type; otherwise, $\mathcal{E}[\![\phi]\!]$ would not be a type. We have $\gamma = *\gamma'$, so we can inductively define $\phi\!\downarrow\!\gamma = \phi_i\!\downarrow\!\gamma'$.

Since any other choice for ϕ would result in an x-type, we have exhausted all cases. $\Box$

Lemma 6.13: If $\gamma \in \mathcal{E}[\![\phi]\!]$, then there is an expression $\phi \downarrow \gamma$ such that if $\mathcal{E} \preceq \mathcal{E}'$, then $\mathcal{E}'[\![\phi \downarrow \gamma]\!] = \mathcal{E}'[\![\phi]\!] \downarrow \gamma$.
Proof: We shall prove the more general result that for $\otimes_i \mathcal{E}[\![\phi_i]\!]$ we can find an expression θ such that $(\otimes_i \mathcal{E}'[\![\phi_i]\!]) \downarrow \gamma = \mathcal{E}'[\![\theta]\!]$. We proceed by induction in the largest x-type depth in $\otimes_i \mathcal{E}[\![\phi_i]\!]$.

- If $\otimes_i \mathcal{E}[\![\phi_i]\!]$ is a type, then at least one $\mathcal{E}[\![\phi_j]\!]$ is a type. Hence, we can use lemma 6.13 and define $\theta = \mathcal{E}[\![\phi_j]\!] \downarrow \gamma$.
- If $\otimes_i \mathcal{E}[\![\phi_i]\!] = \Lambda$, then γ is empty and $\theta =$ [] will do.
- If $\otimes_i \mathcal{E}[\![\phi_i]\!] = *X$, then $X = \otimes_j \mathcal{E}[\![\psi_j]\!]$, where the ψ_j's are all the list elements in the ϕ_i's. We have $\gamma = *\gamma'$, so we can use the θ inductively defined for $\otimes_j \mathcal{E}[\![\psi_j]\!]$ and γ'.
- If $\otimes_i \mathcal{E}[\![\phi_i]\!] = (n_k : X_k)$, then $\gamma = m.\gamma'$ where $m \in \{n_k\}$ is some component. Let the ψ_j be the subexpressions for all m-components. Then we can use the θ inductively defined for $\otimes_j \mathcal{E}[\![\psi_j]\!]$ and γ'. $\Box$

Lemma 6.14: If $\mathcal{E}[\![\phi]\!] \downarrow \alpha \bowtie \mathcal{E}[\![\phi]\!] \downarrow \beta$ and $\textsc{all}(\mathcal{E}, \mathcal{E}')$, then $\mathcal{E}'[\![\phi]\!] \downarrow \alpha \bowtie \mathcal{E}'[\![\phi]\!] \downarrow \beta$.
Proof: Using lemma 6.13 we get $\mathcal{E}[\![\phi \downarrow \alpha]\!] \bowtie \mathcal{E}[\![\phi \downarrow \beta]\!]$. Using lemma 6.7 we conclude that $\mathcal{E}'[\![\phi \downarrow \alpha]\!] \bowtie \mathcal{E}'[\![\phi \downarrow \beta]\!]$. Using lemma 6.13 again we get $\mathcal{E}'[\![\phi]\!] \downarrow \alpha \bowtie \mathcal{E}'[\![\phi]\!] \downarrow \beta$. $\Box$

Lemma 6.15: If condition 13) holds for $\mathcal{E}$ and $\textsc{all}(\mathcal{E}, \mathcal{E}')$, then condition 13) also holds for $\mathcal{E}'$.
Proof: Looking at the proof of lemma 6.11 we can see that every time a test succeeds with $\mathcal{E}$, and we are allowed to proceed, then the same test will also succeed with $\mathcal{E}'$. The test $\tau \triangleleft \mathcal{E}'[\![\phi]\!]$ will succeed due to lemma 6.7. The remaining tests will succeed due to lemma 6.14. Hence, if an $\mathcal{A}$ exists for $\mathcal{E}$,

then it can also exists for $\mathcal{E}'$. $\square$

At long last we can summarize the proof of the soundness theorem.

Proof of theorem 6.4: Preservation of correctness can be argued for each individual condition. Condition 1) is covered by lemma 6.5. Conditions 2)–12) are covered by lemmas 6.6 and 6.7. Condition 13) follows from lemma 6.15. Finally, decidability of condition 13) is proved in lemma 6.11. $\square$

Notice, that ALL will be sound for any extension of the language which still allows the static correctness conditions to be expressed in terms of $\bowtie$ and $\triangleleft$. The author believes that this covers most imaginable cases.

We conclude this section by proving the optimality of ALL.

Lemma 6.16: If $S \not\preceq T$, then there is a *constant* expression ϕ such that for all $\mathcal{E}$ we have $S \bowtie \mathcal{E}[\![\phi]\!]$ but not $T \bowtie \mathcal{E}[\![\phi]\!]$.
Proof: If $S \not\preceq T$, then by definition there is a finite $A \preceq S$ such that $A \not\preceq T$. We construct an appropriate ϕ by induction in the structure of A; obviously, we can ignore the case $A = \Omega$.

1) If A is Int, then ϕ is 0.

2) If A is Bool, then ϕ is 0=0.

3) If $A = *A_1$, then we have two cases:

- **3.1)** if T is not a list, then ϕ is `[]`.
- **3.2)** if $T = *T_1$, then $A_1 \not\preceq T_1$; we can inductively find a ϕ_1 and define $\phi = [\phi_1]$.

4) If $A = (n_i : A_i)$, then we have three cases:

- **4.1)** if T is not a product, then $\phi =$ `()`.

4.2) if $T = (m_j : T_j)$ and $\{n_i\} \subseteq \{m_j\}$, then there is some $n_\alpha = m_\beta$ such that $A_\alpha \not\preceq T_\beta$. We find recursively a ϕ_β and define $\phi = (m_\beta : \phi_\beta)$.

4.3) if $T = (m_j : T_j)$ and $n_\alpha \notin \{m_j\}$, then we have four cases:

4.3.1) if A_α is Int, then $\phi = (n_\alpha : 0)$.

4.3.2) if A_α is Bool, then $\phi = (n_\alpha : 0{=}0)$.

4.3.3) if A_α is a list, then $\phi = (n_\alpha : [\,])$.

4.3.4) if A_α is a product, then $\phi = (n_\alpha : ())$.

This completes the construction. □

Theorem 6.17: ALL is optimal, i.e., if REQ is sound, then REQ $\Rightarrow$ ALL.
Proof: Let us assume $\textsc{req}(\mathcal{F}, \mathcal{A})$. If $\textsc{all}(\mathcal{F}, \mathcal{A})$ is false, then there is some σ, σ' for which we have $\mathcal{F} \downarrow \sigma = \mathcal{F} \downarrow \sigma'$ but $\mathcal{A} \downarrow \sigma \neq \mathcal{A} \downarrow \sigma'$ *or* there is some x_i such that $\mathcal{F}(x_i) \not\preceq \mathcal{A}(x_i)$. In the former case $\textsc{correct}(\mathcal{F}, \sigma := \sigma')$ holds but $\textsc{correct}(\mathcal{A}, \sigma := \sigma')$ does not. In the latter case lemma 6.16 gives us a ϕ such that $\mathcal{F}(x_i) \bowtie \mathcal{F}[\![\phi]\!]$ but $\neg \mathcal{A}(x_i) \bowtie \mathcal{A}[\![\phi]\!]$. Now, $\textsc{correct}(\mathcal{F}, x_i := \phi)$ holds but $\textsc{correct}(\mathcal{A}, x_i := \phi)$ does not. In either case REQ is shown to be unsound. □

Soundness and optimality of ALL means that we have found the most flexible polymorphism that can be obtained.

7 Local Variables

The example language is less than typical in one important respect: It lacks *local* variables. In this section we generalize the results to include this possibility. We extend the syntax of our language with the production

$$\text{S} \quad ::= \quad \textbf{local}\ x : \tau\ \text{S}\ \textbf{end}$$

The semantics of the **local**-statement is to execute S in a locally extended environment where the new variable x has type τ. We can nest **local**-statements

in arbitrary levels.

The static correctness of this construct is defined as follows:

$$\textsc{correct}(\mathcal{E}, \textbf{local } x{:}\tau \text{ S } \textbf{end}) \equiv \textsc{correct}(\mathcal{E}[x \leftarrow \tau], \text{S})$$

which is hardly controversial. What happens to hierarchical calls? We do not get any suggestion for the type of the local variable, since it does not correspond to an actual parameter. If the situation is still to work, then we must strengthen the properties of sound requirements.

Definition 7.1: A *sound* requirement REQ must also satisfy

$$\textsc{req}(\mathcal{F}, \mathcal{A}) \Rightarrow \forall x, \tau\, \exists \alpha :\ \textsc{req}(\mathcal{F}[x \leftarrow \tau], \mathcal{A}[x \leftarrow \alpha])$$

In this situation, we can always assign a type to the local variable that will make sense in the hierarchical situation. We can, in fact, pretend that the local variable was a parameter whose actual type was α. Hence, the discussion of the dynamic aspects of static correctness carries through without modifications. □

Theorem 7.2: ALL is still sound and optimal.
Proof: Assume $\textsc{all}(\mathcal{F}, \mathcal{A})$. We shall construct an α that always works; as we shall see, this α will be an appropriate mixture of formal and actual types.

Being regular, the type τ has finitely many different subtypes $\tau_1, \tau_2, \ldots, \tau_k$, where $\tau = \tau_1$. The τ_i's can be uniquely defined [5] through a set of type equations of the form

$$\tau_i = f_i(\tau_1, \tau_2, \ldots, \tau_k)$$

Now, the type $\alpha = \alpha_1$ is defined by the equations

$$\alpha_i = \begin{cases} \mathcal{A} \downarrow \sigma & \text{if } \mathcal{F} \downarrow \sigma = \tau_i \\ f_i(\alpha_1, \alpha_2, \ldots, \alpha_k) & \text{otherwise} \end{cases}$$

This is well-defined since, because ALL holds, $\mathcal{F}\downarrow\sigma = \mathcal{F}\downarrow\sigma' = \tau_i$ implies $\mathcal{A}\downarrow\sigma = \mathcal{A}\downarrow\sigma'$.

From monotonicity of the f_i's and $\mathcal{F}\downarrow\sigma \preceq \mathcal{A}\downarrow\sigma$ we see that $\tau_i \preceq \alpha_i$. From this we conclude $\mathcal{F}[x \leftarrow \tau] \preceq \mathcal{A}[x \leftarrow \alpha]$. Next, we must show

$$\forall\sigma,\sigma' :\ (\mathcal{F}[x \leftarrow \tau]\downarrow\sigma = \mathcal{F}[x \leftarrow \tau]\downarrow\sigma')\ \Rightarrow\ (\mathcal{A}[x \leftarrow \alpha]\downarrow\sigma = \mathcal{A}[x \leftarrow \alpha]\downarrow\sigma')$$

We have two new cases:

1) if two subtypes of τ are equal, then by definition the corresponding subtypes of α are equal.
2) if $\mathcal{F}\downarrow\sigma = \tau_i$, then $\mathcal{A}\downarrow\sigma = \alpha_i$ and we are done.

We conclude that ALL$(\mathcal{F}[x \leftarrow \tau], \mathcal{A}[x \leftarrow \alpha])$ holds, so ALL is still sound. Optimality is immediate, since we have *reduced* the set of sound requirements. □

We can, of course, extend the language further by changing the **local**-statement to

S ::= **local** P **end**

which will in no way influence the validity of the results.

7.1 Global Variables

On a more negative note, we can eliminate the possibility of allowing *global* variables to be accessible from within procedures.

Proposition 7.3: If global variables belong to the formal environments of procedure bodies and REQ is sound, then REQ $\Rightarrow$ EQUAL.
Proof: Assume that $\tau_i = \mathcal{F}(x_i) \neq \mathcal{A}(x_i)$. Then the situation

var $y : \tau_i$

Proc P$(\ldots, x_i : \tau_i, \ldots)$
    $y := x_i$
end P

will not remain statically correct when we substitute $\mathcal{A}$ for $\mathcal{F}$. □

The problem is that, unlike the situation with local variables, the types of global variables are *fixed* in all actual environments.

We do not view this as a major drawback of our system, but rather as an observation of one more deficiency of this variable mechanism.

8 Opacity

A transparent type definition such as

Type Money = Int

provides Money as a *synonym* for the type Int. This allows us to arbitrarily mix values of types Money and Int, which may not be what we wanted. In particular, if we had two definitions such as

Type Apples = Int
Type Oranges = Int

then it is possibly a conceptual mistake to compare such values.

The usual alternative is an *abstract* type definition where the representation type is completely hidden. This certainly provides the desired protection. However, it is now necessary to re-implement all the standard Int operations

for the abstract type. This is clearly unwanted in this situation and a high price to pay for protection.

A third possibility is an *opaque* type definition that offers protection but simultaneously makes all the usual operations available. This is a compromise between the two other kinds of type definitions. The types defined by

Type Apples $\leftarrow$ Int
Type Oranges $\leftarrow$ Int

are different from each other and from Int, but they all allow the usual integer constants, + and $\Leftrightarrow$ operations, and so on.

In this section we incorporate opaque types into the type system. We indicate the minor modifications that are required to carry all major results through. As a very significant special case we obtain a more flexible hierarchical polymorphism by using opaque versions of the type Ω as distinct type "variables".

8.1 Opaque Types

Rather than merely provide opaque *definitions*, we introduce opaque *types* through an *opacity operator*. This is preferable to introducing $\leftarrow$ directly and axiomatizing its properties.

We extend the language of types as follows

τ ::=	Int | Bool |	simple types
	N_i |	type names
	$*\tau$ |	lists
	$(n_1 : \tau_1, \ldots, n_k : \tau_k)$ |	partial products, $k \geq 0$, $n_i \neq n_j$
	$n \Box \tau$	opaque versions

We consider $\Box$ to be a unary type constructor that creates *named*, *opaque* versions of its argument type. The values of an opaque version are the same as those of the original.

Type Equivalence

Type equivalence is defined to be equality of normal forms. The normal form of a type is a (possibly infinite) labeled tree that, informally, is obtained by the unfolding of the type definitions. This technique generalizes without problems, so that

$$n_1 \Box T_1 \approx n_2 \Box T_2 \;\; \textit{iff} \;\; n_1 = n_2 \;\wedge\; T_1 \approx T_2$$

Thus, among the following types

Type $A = \mathrm{Int}$
Type $B = b\Box\mathrm{Int}$
Type $C = c\Box\mathrm{Int}$
Type $D = b\Box B$
Type $E = b\Box A$

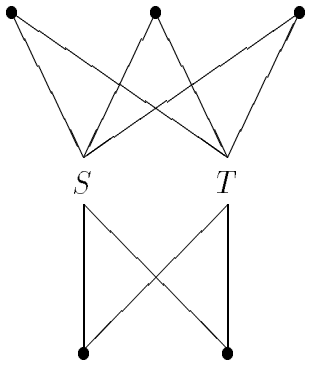
only B and E are equivalent. Type equivalence is still decidable.

Type Ordering

The type ordering concerns itself with possibilities for *code reuse*. The idea is that code written for smaller types can be reused for larger types. For this purpose we want to ignore the protection offered by opacity. Thus, the finite ordering $\preceq_0$ must further satisfy

$$(n\Box T \preceq_0 S \;\Leftrightarrow\; T \preceq_0 S) \;\wedge\; (T \preceq_0 n\Box S \;\Leftrightarrow\; T \preceq_0 S)$$

As before, the type ordering $\preceq$ is the closure of $\preceq_0$. Notice that we now have a *preorder* rather than a partial order; for example, $\mathrm{Int} \preceq m\Box\mathrm{Int}$ and $m\Box\mathrm{Int} \preceq \mathrm{Int}$ but $\mathrm{Int} \not\approx m\Box\mathrm{Int}$. This will in no way influence our results; it is just an observation that two types may be unequal and still be able to reuse each other's code. In general, two types S and T are *opaquely related*, if $S \preceq T$, $T \preceq S$, and $S \not\approx T$. They are different but they have the same order relations to all other types, which may be illustrated as follows



The type (pre)ordering, least upper bounds, and greatest lower bounds remain computable.

The Language

The only required extension to the example language is the opaque types themselves. We add to our grammar the production

$$\tau \quad ::= \quad n\Box\tau$$

For convenience, we also introduce type equations of the form

$$\mathrm{D} \quad ::= \quad \textbf{Type}\ N \leftarrow \tau$$

They abbreviate the more involved equations

$$\textbf{Type}\ N = N\Box\tau$$

While the N on the left-hand side is simply a type variable that can be α-reduced, the N on the right-hand side is an integral part of the type. This allows us to write opaque *definitions* such as

$$\textbf{Type}\ \mathrm{Money} \leftarrow \mathrm{Int}$$

Here, Money is no longer merely a synonym for Int; it is a new and different type.

Since opaque definitions merely abbreviates opaque types, we also have a natural interpretation of recursive opaque definitions such as

Type $F \leftarrow F$
Type $G \leftarrow *G$

While the usefulness of such types may be questioned, their properties are at least simply understood. For example, F enjoys the unique property of being equal to an opaque version of itself.

8.2 Extended Types

We now have a new class of polymorphic constants besides `[]` and `(b:87)`; for example, the constant 7 denotes a value not only of type Int, but also of all opaquely related types.

To handle this situation we extend the x-types to

$$\begin{array}{rl} X ::= & \tau \mid \\ & *X \mid \\ & \Lambda \mid \\ & \Pi(n_1 : X_1, \ldots, n_k : X_k) \mid \\ & \Box X \end{array}$$

The elements of $\Box X$ are the elements of X and their opaque versions. The computations on x-types must be modified as follows.

Proposition 5.6+: $\bowtie$ is the smallest symmetric relation which satisfies

- $T_1 \bowtie T_2$, if $T_1 = T_2$ are types
- $\Lambda \bowtie \Lambda$
- $\Lambda \bowtie *X$
- $*X_1 \bowtie *X_2$ *iff* $X_1 \bowtie X_2$
- $(n_i : T_i) \bowtie \Pi(m_j : Y_j)$ *iff* $\{m_j\} \subseteq \{n_i\} \ \wedge \ (\forall i, j : \ n_i = m_j \Rightarrow T_i \bowtie Y_j)$
- $\Pi(n_i : X_i) \bowtie \Pi(m_j : Y_j)$ *iff* $(\forall i, j : \ n_i = m_j \Rightarrow X_i \bowtie Y_j)$
- $X \bowtie \Box X$
- $\Box X_1 \bowtie \Box X_2$ *iff* $X_1 \bowtie X_2$
- $n\Box T \bowtie \Box X$ *iff* $T \bowtie \Box X$

□

Proposition 5.8+: Whenever its arguments are related by $\bowtie$, then $\otimes$ can be computed as follows

- $T_1 \otimes T_2 = T_1$, if $T_1 = T_2$ are types
- $\Lambda \otimes \Lambda = \Lambda$
- $\Lambda \otimes *X = *X$
- $*X_1 \otimes *X_2 = *(X_1 \otimes X_2)$
- $(n_i : T_i) \otimes \Pi(m_j : Y_j) = (n_i : T_i)$
- $\Pi(n_i : X_i) \otimes \Pi(m_j : Y_j) = \Pi(z_k : Z_k)$ where $\{z_k\} = \{n_i\} \cup \{m_j\}$ and
$$Z_k = \begin{cases} X_i & \text{if } z_k = n_i \notin \{m_j\} \\ X_i \otimes Y_j & \text{if } z_k = n_i = m_j \\ Y_j & \text{if } z_k = m_j \notin \{n_i\} \end{cases}$$
- $X \otimes \Box X = X$
- $\Box X_1 \otimes \Box X_2 = \Box(X_1 \otimes X_2)$
- $n\Box T \otimes \Box X = n\Box T$

□

Proposition 5.11+: The relation $S \triangleleft X$ determines if there is an element of the x-type X which is larger than the type S. It is the smallest relation which satisfies

- $S \triangleleft T$, if T is a type and $S \preceq T$

- $\Omega \triangleleft X$
- $*S \triangleleft *X$ *iff* $S \triangleleft X$
- $*S \triangleleft \Lambda$
- $(n_i : S_i) \triangleleft \Pi(m_j : X_j)$ *iff* $(\forall i, j : n_i = m_j \Rightarrow S_i \triangleleft X_j)$
- $n\Box S \triangleleft X$ *iff* $S \triangleleft X$
- $S \triangleleft \Box X$ *iff* $S \triangleleft X$

□

All proofs of propositions in section 5 generalize without difficulties.

8.3 Correctness

These extensions allow us once again to assign unique x-types to expressions

Definition 5.14+: If $\mathcal{E}$ is an environment and ϕ is an expression, then $\mathcal{E}[\![\phi]\!]$ is defined inductively as follows

- $\mathcal{E}[\![0]\!] = \Box\text{Int}$
- $\mathcal{E}[\![\phi\texttt{+}1]\!] = \mathcal{E}[\![\phi\texttt{-}1]\!] = \mathcal{E}[\![\phi]\!]$
- $\mathcal{E}[\![\sigma]\!] = \mathcal{E} \downarrow \sigma$
- $\mathcal{E}[\![\phi_1 \texttt{ = } \phi_2]\!] = \Box\text{Bool}$
- $\mathcal{E}[\![\texttt{[}\phi_1\texttt{,}\ldots\texttt{,}\phi_k\texttt{]}]\!] = \Box*(\otimes_i \mathcal{E}[\![\phi_i]\!])$, if $k > 0$
- $\mathcal{E}[\![\texttt{[]}]\!] = \Box\Lambda$
- $\mathcal{E}[\![\texttt{|}\,\phi\,\texttt{|}]\!] = \Box\text{Int}$
- $\mathcal{E}[\![\texttt{(}n_i\texttt{:}\,\phi_i\texttt{)}]\!] = \Box\Pi(n_i : \mathcal{E}[\![\phi_i]\!])$
- $\mathcal{E}[\![\textbf{has}\texttt{(}\phi\texttt{,}n_i\texttt{)}]\!] = \Box\text{Bool}$

□

Until the type of an expression has been fixed, it will match all opaquely related types alike.

No other definitions need to be changed; in particular, the definitions of correctness and soundness remain the same.

The proofs of lemmas 6.5–6.7, 6.9, and 6.12–6.15 only require minor modifications to handle the extra cases in the structural induction. The proofs of the main results, lemma 6.11 and theorem 6.4, can remain unchanged. The proof of optimality in theorem 6.17 only requires a trivial modification of lemma 6.16. All of section 7 goes through unchanged.

This shows how opaque types with remarkably little effort can be integrated into this hierarchical type system. In the following section we demonstrate how they even provide an added flexibility.

8.4 Hierarchical Procedures

As demonstrated earlier, it is pragmatically useful to distinguish between intended and unintended type equalities. In connection with the hierarchical polymorphism, opaque types can serve another important function. A hierarchical call of a procedure such as

```
Proc P(var x,y:Ω)
    x:=x; y:=y
end P
```

requires that the actual types of x and y are equal, since their formal types are equal. However, since the procedure keeps the two variables separate this is actually too strict. By specifying the formal types as two opaque versions of Ω we guarantee that they will never be mixed and, hence, we can allow more hierarchical calls of the procedure.

As a more telling example, consider the following "generic" type of finite maps. Without opaque types we could not avail ourselves of *two* type "variables".

```
Type Arg ← Ω
Type Res ← Ω
Type Map = (a:Arg, r:Res, next:Map)

Proc Update(var m:Map, val a:Arg, val r:Res)
    m:=(a,r,m)
end Update

  ⋮
```

All these Map-procedures can now be reused for maps with arbitrary types in place of Arg and Res.

9 Conclusions and Future Work

The results in this paper establish the theoretical basis for a powerful and general type hierarchy with static type checking. Naturally, we hope that this can develop into a complete programming language.

It is worth noting that ALL in fact defines a partial order $\leq$ on types. This seems to suggest that the hierarchical mechanism may be viewed as a version of (implicit) bounded parametric polymorphism [3]. The ordering $\leq$ is, however, radically different from the usual subtyping relation, as it satisfies

$$\forall S_1 \leq T_1 \; \exists S_2 \leq T_2 : \; (S_1, S_2) \not\leq (T_1, T_2)$$

which we might call *anti-compositionality*, to coin a phrase.

An obvious direction of research concerns higher-order types. A naive inclusion of function (or procedure) types will be quite consistent with the present system. However, as is usually the case, the valid relations between higher-order types are not the ones that we would hope for. A more promising approach is to directly develop a module concept.

The introduction of opaque types seems to fill a gap between synonym types and abstract types. Another view is that they provide a unification of struc-

tural and name equivalence of types; the programmer can decide on the combination which is most suited for the application. Opaque types have been smoothly integrated with the hierarchical system; they can even be seen to increase the available polymorphic flexibility.

10 References

[1] **Cardelli, L.** "Typeful Programming" *DEC Research Report 45*, 1989.

[2] **Cardelli, L. & Mitchell, J.** "Operations on Records" in *Proceedings of MFPS'90, LNCS Vol 442*, Springer-Verlag, 1990.

[3] **Cardelli, L. & Wegner, P.** "On Understanding Types, Data Abstraction, and Polymorphism" in *Computing Surveys, Vol 17 No 4*, ACM 1985.

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

[6] **Reynolds, J.C.** "Three approaches to type structure.", In *Mathematical Foundations of Software Development, LNCS Vol 185*, Springer-Verlag 1985.

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

[8] **Schwartzbach, M.I.** "Infinite Values in Hierarchical Imperative Types" in *Proceedings of CAAP'90, LNCS*, Springer-Verlag 1990.

[9] **Schwartzbach, M.I. & Schmidt, E.M.** "Types and Automata". *PB-316*, Department of Computer Science, Aarhus University, 1990.

[10] **Wirth, N.** "Type Extensions.", In *Transactions on Programming Languages and Systems Vol 10 No 2*, ACM 1988.