ISSN 0105-8517

Static Correctness of Hierarchical Procedures

Michael I. Schwartzbach

DAIMI PB – 295
November 1989

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark

Static Correctness of Hierarchical Procedures

Michael I. Schwartzbach[1]

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

Draft of November 3, 1989

Abstract

A system of hierarchical, imperative, fully recursive types 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 very powerful and general type hierarchy with static type checking, which enables 1st order polymorphism and (multiple) inheritance in a language with assignments.

1 Introduction

In [4] we presented a system of fully recursive, hierarchical types for an imperative language. The *hierarchical* aspect of the type system means that we have a partial order on types, such that program fragments written for small types may be applied to all larger types. This can form the basis for a method of reusing software that combines simple parametric polymorphism with elements of class inheritance. Static type checking is still possible.

[1]E-mail address: mis@daimi.dk

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 this paper we describe an example language, give a precise definition of static correctness, and prove for a particular choice of static requirement, ALL, that correctness is preserved. We further prove that ALL is optimal, in the sense that it allows as many hierarchical calls as can possibly make sense. The results generalize from the modest example language to include all conventional imperative language features.

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 $\mid$ Bool $\mid$	simple types
	$N_i \mid$	type names
	$*\tau \mid$	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[2].

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 [4].

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 assignments of values to types are possible; for example, one may include *infinite (lazy) values.* The variety of different value assignments is investigated in [5].

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.* 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.

Proposition 2.1: The equivalence is decidable.
Proof: Normal forms are clearly *regular*, i.e. the have only a finite number of *different* subtrees. Hence, their equality is decidable, as stated in [2]. □

[2]Thus, a partial product is essentially the *union* of all the ordinary products whose names are among those specified.

2.2 Type Ordering

The hierarchical ordering is a refinement of $\sqsubseteq$. We want to include orderings between partial product types of the following kind

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

iff $\{n_i\} \subseteq \{m_j\}$ and $(foralli, j : n_i = m_j \Rightarrow T_i \preceq S_j)$. 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.

__**Facts 2.2**__:

- 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.
- The ordering, $\sqcap$'s and $\sqcup$'s are all computable.

The least upper bounds of $\preceq$ correspond to the *multiple inheritance* aspect of data values: Two types can be joined by the (recursive) unification of their components. In fact, we obtain a generalization of the ordinary multiple inheritance, since we have recursive (infinite) types and the polymorphic type Ω.

The greatest lower bounds correspond to the *general specialization* of data values: The maximal common subtype of two types can be determined.

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.

$$
\begin{array}{lcl}
\mathrm{S} & ::= & \sigma := \phi \mid \\
 & & \sigma :- n_i \mid \\
 & & \sigma :+ (n_i : \phi) \mid \\
 & & P(\phi_1, \ldots, \phi_k) \mid \\
 & & \textbf{if } \phi \textbf{ then } \mathrm{S} \textbf{ end} \mid \\
 & & \textbf{while } \phi \textbf{ do } \mathrm{S} \textbf{ end} \mid \\
 & & \mathrm{S}_1 \ ; \ \mathrm{S}_2
\end{array}
$$

$$
\sigma \ ::= \ x \mid \sigma . n_i \mid \sigma [\phi]
$$

$$
\begin{array}{lcl}
\phi & ::= & 0 \mid \phi\text{+}1 \mid \phi\text{-}1 \mid \\
 & & \sigma \mid \\
 & & \phi_1 = \phi_2 \mid \\
 & & [\phi_1, \ldots, \phi_k] \mid \\
 & & |\phi| \mid \\
 & & (n_1 : \phi_1, \ldots, n_k : \phi_k) \mid \\
 & & \textbf{has}(\phi, n_i)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{D} & ::= & \textbf{Type } N = \tau \mid \\
 & & \textbf{Proc } P(\rho\, x_1 : \tau_1, \ldots, \rho\, x_k : \tau_k) \ \mathrm{S} \textbf{ end } P \mid
\end{array}
$$

Var $x : \tau$

$$
\begin{aligned}
\rho \; &::= \; \textbf{var} \mid \textbf{val} \\
\tau \; &::= \; \text{Int} \mid \text{Bool} \mid \\
& \quad\; N \mid \\
& \quad\; *\tau \mid \\
& \quad\; (n_1 : \tau_1, \ldots, n_k : \tau_k) \\
\text{P} \; &::= \; \text{D}_1 \; \text{D}_2 \ldots \text{D}_k \; \text{S}
\end{aligned}
$$

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 $\sigma :- n_i$ removes n_i from the domain of σ, $\sigma :+ (n_i : \phi)$ updates σ with the value ϕ on n_i, and **has**(ϕ, n_i) decides whether n_i is in the domain of ϕ. Arbitrary partial product constants can be denoted by $(n_1 : \phi_1, \ldots, n_k : \phi_k)$. 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 The Hierarchy

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 does not have to be changed

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

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

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

We claim that our type system is the right one for allowing hierarchical procedure calls, as other approaches have various shortcomings. One method could be to rely on *coercions* and *inclusions* of values [3], such that $T_1 \preceq T_2$ if $\mathrm{Val}(T_1) \subseteq \mathrm{Val}(T_2)$. In our system this monotonicity is a necessary but not sufficient condition. Certainly, we have Int $\subseteq$ Real, but if Int procedures can be reused for Reals then we face the awkward task of explaining e.g. what the 3.1415'th element of a list is. The technique of *retracts*, which yield best approximations, is not always the answer. In e.g. a generic search tree, it would be most unfortunate if one could only recover approximations to the inserted elements. The system of [7] is based on records and projections between these and fails to have the above mentioned monotonicity property. This causes various inconsistencies, since the hierarchical mechanism allows values of small types to be assigned to variables of larger types. The system of [1] avoids such problem by excluding records with "updatable fields" from the hierarchy.

Common to all such systems, including ours, is that just allowing actual parameters to have larger types than formal parameters is too liberal an attitude – a fact that is not always realized. 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 relative 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 $x \in \mathrm{dom}(\mathcal{E})$
- $\mathcal{E}\downarrow\sigma\,[\phi] = T$ 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

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

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.

Definition 5.5: The relation $X_1 \bowtie X_2$ holds iff X_1 and X_2 has 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: 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. □

Proposition 5.10: The relation $S \lhd X$ 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

- $S \lhd T$, if T is a type and $S \preceq T$
- $\Omega \lhd X$
- $*S \lhd *X$ iff $S \lhd X$
- $*S \lhd \Lambda$
- $(n_i : S_i) \lhd \Pi(m_j : X_j)$ iff $(\forall i, j :\ n_i = m_j \Rightarrow S_i \lhd X_j)$

Proof: Induction in the structure of X. □

Proposition 5.11: $S \lhd X_1 \otimes X_2 \ \Leftrightarrow\ S \lhd X_1 \ \wedge\ S \lhd X_2$
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 $S \lhd X_1 = X_2 = X_1 \otimes X_2$.
- If e.g. $X_1 = \Lambda$ then $X_1 \otimes X_2 = X_2$.
- If $X_i = *Y_i$ then $S = *T$ and $T \lhd Y_i$ so by induction hypothesis $T \lhd Y_1 \otimes Y_2$ so $*T \lhd *(Y_1 \otimes Y_2) = X_1 \otimes X_2$.
- 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. □

Proposition 5.12: All of: $\bowtie$, $\otimes$ and $\lhd$ are computable.
Proof: Immediate from decidability of $\approx$ and $\preceq$. □

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

Definition 5.13: 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 $k > 0$
- $\mathcal{E}[\![\texttt{[]}]\!] = \Lambda$
- $\mathcal{E}[\![\texttt{|}\phi\texttt{|}]\!] = \text{Int}$
- $\mathcal{E}[\![\texttt{(}n_i\texttt{:}\phi_i\texttt{)}]\!] = \Pi(n_i : \mathcal{E}[\![\phi_i]\!])$
- $\mathcal{E}[\![\textbf{has}\texttt{(}\phi\texttt{,}n_i\texttt{)}]\!] = \text{Bool}$

We shall only use $\mathcal{E}[\![\phi]\!]$ in situations where it will in fact be defined.

Definition 5.14: We present a predicate $\text{CORRECT}(\mathcal{E}, \text{S})$ 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 derivation of S.

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

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

Proc $P(\rho\, x_1 : \tau_1, \ldots, \rho\, x_k : \tau_k)$
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

$$\text{EQUAL}(\mathcal{F}, \mathcal{A}) \equiv \mathcal{F} = \mathcal{A}$$

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.[3]

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. Thus, the requirement must possess a special quality.

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

- $\forall \mathrm{S}, \mathcal{F}, \mathcal{A} : \textsc{correct}(\mathcal{F}, \mathrm{S}) \wedge \textsc{req}(\mathcal{F}, \mathcal{A}) \Rightarrow \textsc{correct}(\mathcal{A}, \mathrm{S})$
- Condition 13) in definition 5.14 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.

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

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

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

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 $\text{ALL}(\mathcal{F},\mathcal{A})$.

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 $X = \otimes(\mathcal{F}[\![\psi_i]\!])$. Hence, $\mathcal{A}[\![\phi_1]\!] = \Lambda$ and $\mathcal{A}[\![\phi_2]\!] = *Y$ where $Y = \otimes(\mathcal{A}[\![\psi_i]\!])$.

- 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 $n_i = m_j \Rightarrow T_i \bowtie X_j$. 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 $S \lhd \mathcal{F}[\![\phi]\!]$ then $S \lhd \mathcal{A}[\![\phi]\!]$.
Proof: Induction in ϕ. Assume $S \lhd \mathcal{F}[\![\phi]\!]$:

- 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 $\lhd$ 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 $T \lhd \otimes_i(\mathcal{F}[\![\phi_i]\!])$, so (using proposition 5.11) $T \lhd \mathcal{F}[\![\phi_i]\!]$. By induction hypothesis $T \lhd \mathcal{A}[\![\phi_i]\!]$, so (using lemma 6.6 and proposition 5.11) $T \lhd (\otimes_i \mathcal{A}[\![\phi_i]\!])$ and $S = *T \lhd *(\otimes_i \mathcal{A}[\![\phi_i]\!]) = \mathcal{A}[\![\phi]\!]$.
- 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 induction hypothesis $S_i \preceq \mathcal{A}[\![\phi_j]\!]$ so $S = (n_i : S_i) \lhd (m_j : \mathcal{A}[\![\phi_j]\!]) = \mathcal{A}[\![\phi]\!]$. □

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 specifies a subtree of X, which we will call $X \downarrow \gamma$.

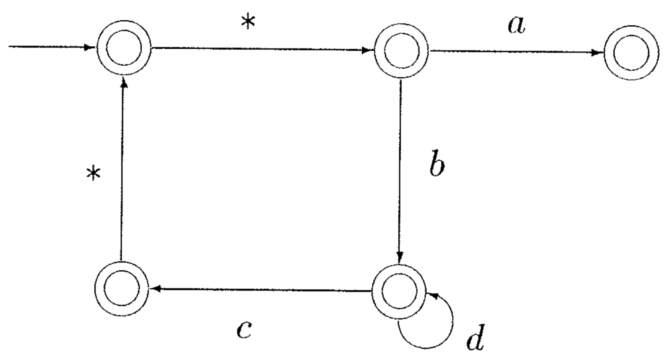
Lemma 6.9: Let $F \preceq A$ be types. Then $\forall \alpha, \beta : F \downarrow \alpha = F \downarrow \beta \Rightarrow A \downarrow \alpha = A \downarrow \beta$ is a decidable condition.

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. Since $F \preceq A$ we have that $\mathcal{L}(M_F) \subseteq \mathcal{L}(M_A)$. Each language splits into equivalence classes, one for each state, such that equivalent addresses reach the same (acceptance) state. Let $\mathcal{L}(M_F) = \bigcup F_i$ and $\mathcal{L}(M_A) = \bigcup F_j$. The condition translates to $\forall i\ \exists j : F_i \subseteq A_j$, which is clearly decidable since each class is a regular language. □

Example 6.10: The type T defined by the equations

$$T = *R$$
$$R = (a : \text{Int},\ b : S)$$
$$S = (c : *T,\ d : S)$$

has 5 different subtypes, viz. $\{T, R, \text{Int}, S, *T\}$. The corresponding automaton has 5 (accept) states and the following transitions



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 $\mathcal{E}[\![\phi]\!]$.

To demonstrate decidability we present an algorithm that computes an appropriate $\mathcal{A}$ or determines that none exists.

Our first test is whether $\tau \lhd \mathcal{E}[\![\phi]\!]$. 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, an $\mathcal{A}$ does exist, since we for each set of pairwise $\bowtie$ x-types can find a common element. Due to proposition 5.11 this common element can be chosen to be larger than the formal type. $\square$

Example 6.12: Consider the following procedure heading

Proc P(**val** $x_1:\tau_1$, **val** $x_2:\tau_2$, **val** $x_3:\tau_3$)

where

$$\tau_1=\tau_2=*(a:\Omega),\quad \tau_3=\Omega$$

We call with the actual parameters

$$\phi_1 = \texttt{[(c:[])]}$$
$$\phi_2 = \texttt{[(a:[]),(a:[[]]),(b:7)]}$$
$$\phi_2 = \texttt{[[5=7]]}$$

The x-types of the parameters, and our first proposal for actual types, are

$$\mathcal{A}(x_1) = \mathcal{E}[\![\phi_1]\!] = *\Pi(c : \Lambda)$$
$$\mathcal{A}(x_2) = \mathcal{E}[\![\phi_2]\!] = *(\Pi(a : \Lambda) \otimes \Pi(a : *\Lambda) \otimes \Pi(b : \text{Int})) = *\Pi(a : *\Lambda, b : \text{Int})$$
$$\mathcal{A}(x_3) = \mathcal{E}[\![\phi_3]\!] = **\text{Bool}$$

We can verify that $\tau_i \lhd \mathcal{E}[\![\phi_i]\!]$. In stage 1 we find the single short/long pair $x_2 * a, x_3$ and obtain

$$\mathcal{A}(x_1) = *\Pi(c : \Lambda)$$
$$\mathcal{A}(x_2) = *\Pi(a : **\text{Bool}, b : \text{Int})$$
$$\mathcal{A}(x_3) = **\text{Bool}$$

In stage 2 we find no long/long conflicts and, thus, confirms the substitution performed in stage 1. In stage 3 we detect the short/short pairs x_1, x_2 and x_1*, x_2* for which we must select common elements. We end up with the actual types

$$\mathcal{A}(x_1) = *(a : **\text{Bool}, b : \text{Int}, c : *\Omega)$$
$$\mathcal{A}(x_2) = *(a : **\text{Bool}, b : \text{Int}, c : *\Omega)$$
$$\mathcal{A}(x_3) = **\text{Bool}$$

that are clearly satisfactory. In place of $*\Omega$ we could have chosen *any* list type; however, it seems appropriate to select the smallest actual types possible.

Lemmas 6.13 through 6.16 will show preservation of condition 13).

Lemma 6.13: 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 must 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. □

Lemma 6.14: 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 $(\otimes_i \mathcal{E}'[\![\phi_i]\!])\!\downarrow\!\gamma = \mathcal{E}'[\![\theta]\!]$. Induction in 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 must 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. The we can use the θ inductively defined for $\otimes_j \mathcal{E}[\![\psi_j]\!]$ and γ'. □

Lemma 6.15: If $\mathcal{E}[\![\phi]\!]\downarrow\alpha \bowtie \mathcal{E}[\![\phi]\!]\downarrow\beta$ and $\text{ALL}(\mathcal{E},\mathcal{E}')$ then $\mathcal{E}'[\![\phi]\!]\downarrow\alpha \bowtie \mathcal{E}'[\![\phi]\!]\downarrow\beta$.
Proof: Using lemma 6.14 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 6.14 again we get $\mathcal{E}'[\![\phi]\!]\downarrow\alpha \bowtie \mathcal{E}'[\![\phi]\!]\downarrow\beta$. □

Lemma 6.16: If condition 13) holds for $\mathcal{E}$ and $\text{ALL}(\mathcal{E},\mathcal{E}')$ 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 in the algorithm, then the same test will also succeed with $\mathcal{E}'$. The test $\tau \lhd \mathcal{E}'[\![\phi]\!]$ will succeed due to lemma 6.7. The remaining tests will succeed due to lemma 6.15. Hence, if the algorithm can produce an $\mathcal{A}$ that works for $\mathcal{E}$, then it can also produce one that works for $\mathcal{E}'$. □

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 6.16. 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 6.17: 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 $\phi =$ [ϕ_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_β:ϕ_β).
- **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_α:0).
  - **4.3.2)** if A_α is Bool then $\phi =$ (n_α:0=0).
  - **4.3.3)** if A_α is a list then $\phi =$ (n_α:[]).
  - **4.3.4)** if A_α is a product then $\phi =$ (n_α:()).

This completes the construction. □

Theorem 6.18: If REQ is sound then REQ $\Rightarrow$ ALL.
Proof: Let us assume REQ$(\mathcal{F},\mathcal{A})$. If ALL$(\mathcal{F},\mathcal{A})$ is false then there is some σ,σ' such that $\mathcal{F}\downarrow\sigma=\mathcal{F}\downarrow\sigma'$ but $\mathcal{A}\downarrow\sigma\neq\mathcal{A}\downarrow\sigma'$ *or* there is some x_i such that $\mathcal{F}(x_i)\not\leq\mathcal{A}(x_i)$. In the former case CORRECT$(\mathcal{F},\sigma:=\sigma')$ holds but CORRECT$(\mathcal{A},\sigma:=\sigma')$ does not. In the latter case lemma 6.17 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, CORRECT$(\mathcal{F},x_i:=\phi)$ holds but 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 hierarchy that can be permitted.

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

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

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

The static correctness of this construct is defined as follows:

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

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

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

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 $\text{ALL}(\mathcal{F}, \mathcal{A})$. We shall construct an α that always works; as we shall see, this α will be an appropriate mixture of formal and actual types.

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 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 $\text{ALL}(\mathcal{F}[x\leftarrow\tau], \mathcal{A}[x\leftarrow\alpha])$ 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 the hierarchy, but rather as an observation of one more deficiency of this variable mechanism.

8 Weaker Requirements

An idea to allow even more hierarchical procedure calls might suggest itself. If we permit the static requirement to consider the body of the procedure, then more actual parameters can be shown to preserve correctness; for example, consider the following procedures

Proc P(**var** x,y:(a:Int))
  x:=x;
  y:=y
end P

```
Proc Q(var z:*Ω)
    z:=z
end Q
```

For procedure P we require that the actual types of x and y are equal, but really they can never conflict. For procedure Q we require that the actual type of z is a list, but really any type will do.

We shall argue against yielding to such demands by weakening the static requirement. In doing so, we would have to abandon an important abstraction principle, as the interface to a procedure would no longer be reflected solely by its formal parameters. Imagine, now, the development of a large software system. Some day, the body of an obscure procedure is modified slightly, in a manner that disallows a few hierarchical calls. This change is propagated throughout, and in a nightmarish scenario no part of the system would remain correct. So much for abstraction and modularity!

The logical next step on this perilous road would lead us to redefine static correctness, such that only code that is actually executed need to be correct. Though undecidable, we could apply various conservative estimates, but the result would be even more precarious than the above mentioned.

In many cases, the problem can be remedied by writing better programs. Clearly, procedure P above should be split in two, and the formal parameter in procedure Q should have type Ω. In [4] we suggest the introduction of an infinite family of named Ω-like types that will allow more than one "type variable" in procedures. This, too, will resolve many situations.

9 Conclusions and Future Work

The results in this paper establish the theoretical basis for a very powerful and general type hierarchy with static type checking. Naturally, we hope that this can develop into a complete programming language. Much work is needed to develop efficient data representations and to tune the type checking algorithms.

Independently of such advances, we expect this work to shed some light on static type checking in object oriented languages – a fairly elusive concept, so far. Some applications of class hierarchies are inconsistent with the idea that classes are types, but to a large extent this hierarchical mechanism can be viewed as a restriction of the present one, in which case a greatly simplified version of the proposed type checking discipline can be employed. Work is currently being undertaken to investigate these ideas [6].

10 References

[1] **Cardelli, Luca** "Amber." Proceedings of the *Treizieme Ecole de Printemps d'Informatique Theorique*, May 1985.

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

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

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

[5] **Schwartzbach, Michael I.** "Infinite Values in Hierarchical Imperative Types" DAIMI PB-293, Department of Computer Science, Aarhus University 1989.

[6] **Schwartzbach, Michael I. & Palsberg, Jens** "A New Understanding of Classes as Types", In preparation.

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