ISSN 0105-8517

An Imperative Type Hierarchy with Partial Products

Erik M. Schmidt
Michael I. Schwartzbach

DAIMI PB – 271
December 1988

An Imperative Type Hierarchy with Partial Products

Erik M. Schmidt
Michael I. Schwartzbach

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

Abstract

A *type hierarchy* for an imperative language defines an ordering on the types such that any application for small types may be reused for all larger types. The imperative facet makes this non-trivial; the straight-forward definitions will yield an inconsistent system. We introduce a new type constructor, the *partial product*, and show how to define a *consistent* hierarchy in the context of *fully recursive* types. A simple *polymorphism* is derived. By extending the types to include *structural invariants* we obtain a particularly appropriate notation for defining recursive types, that is superior to traditional type sums and products. We show how the ordering on types *extends* to an ordering on types with invariants. We allow the use of *least upper bounds* in type definitions and show how to resolve type equations involving these, and how to compute upper bounds of invariants.

1 Introduction

A *type hierarchy* consists of an *ordering* of the types of a programming language, along with a method for allowing applications written for "small" types to be reused for "larger" types. The ordering typically centers around inclusions or projections of value sets. Our aim is to introduce a type hierarchy as an *orthogonal* extension to an imperative language. We want the ordering to be an *independently* defined relation on types, so that whenever this relation holds then all applications for the smaller type will be reusable for the larger type. The imperative facet, in particular the use of reference parameters, makes this a non-trivial problem.

Type hierarchies are well-known in the context of functional languages. The language Amber [Cardelli85], which exemplifies many aspects of high-level type systems, has such an independently defined type ordering[1]. However, the hierarchy does not extend to the imperative facet of the language, since any relation between *updatable fields* is explicitly disallowed.

[1]Alas, with the opposite notion of large vs. small.

Other type orderings are based on *coercions* of values [Reynolds85, Tennent88]. Here, one allows expressions of the smaller type to be used as expressions of the larger type, and variables of the larger type to be used as variables of the smaller type. This is, however, not sufficient to define the type hierarchy we seek. In section 3 it is demonstrated that the required ordering can not be defined on values alone; in particular, we show that the archetypical coercion from integers to reals can not be exploited in this context.

Object oriented languages such as Simula [Dahl70] are imperative languages with a type hierarchy, but the type ordering is not independently defined. It arises as an incidental of the construction of *classes*, which serve as wrappers for the types and limits the available reusable applications.

In an imperative setting, a type hierarchy, such as we desire, is introduced in [Wirth88] with the following simple assumptions

- the assignment a:=b is legal if the type of b is larger than that of a
- in a procedure call the types of the actual parameters may be larger than those of the formal parameters

Here types are records whose values are products of their component values, and the ordering expresses the possibility of *projecting* large products onto small products. This situation, however, immediately implies the possibility of assigning small values to large variables. Consider the program

```
Proc P(var a: A)
    a := α
end P

Var b: B

P(b)
```

where B is larger than A and α is a constant of type A. It must be legal according to the definitions, but the procedure call has the effect of the assignment b:= α; since the projection philosophy is no longer applicable, it is not clear how α should be interpreted as a value of type B; other anomalies may also occur – the system is inconsistent. A resolution of this problem is to consider α to be a *partial* value of type B.

In this paper, we introduce a *partial product* type constructor, which is incorporated into a system of fully recursive types. We show how a few simple conditions yield a *consistent* imperative type hierarchy. We also derive a mechanism for simple polymorphism. Apart from allowing the hierarchy to be consistently defined, the partial product is particularly appropriate for expressing *recursive types* and yields considerable notational benefits; it is strictly more powerful than type sums and products. A seeming shortcoming of the extreme generality of the partial product leads to the development of a technique for combining *structural invariants* with type constructors. We show how the ordering on types *extends* to an ordering on types with invariants. We allow the use of *least upper bounds* in type definitions and show how to resolve recursive type equations involving these, and how to compute upper bounds of invariants. The resulting system allows *static* type checking.

2 The Types

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

$$\begin{aligned} &\textbf{Type } N_1 = T_1 \\ &\textbf{Type } N_2 = T_2 \\ &\dots \\ &\textbf{Type } N_k = T_k \end{aligned}$$

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

T ::= Int | Bool | Char |	simple types
N_i	type names
$(n_1 : T_1, \dots, n_k : T_k)$	partial products, $k \geq 0$, $n_i \in \mathcal{N}$, $n_i \neq n_j$
$*T$	lists

Here $\mathcal{N}$ is an infinite set of *names.* Types are denoted by type expressions. Notice, that type definitions may involve arbitrary recursion. If N is a type name then $rhs(N)$ denotes the right-hand side of its definition.

Value Sets

Each type expression has an associated set of values, $val(T)$, defined as follows

- $val(\text{Int}) = \{\dots, -1, 0, 1, 2, \dots\}$
- $val(\text{Bool}) = \{\text{true, false}\}$
- $val(\text{Char}) = \{\text{a, b, c, } \dots\}$
- $val((n_1 : T_1, \dots, n_k : T_k)) = \{\phi : \{n_1, \dots, n_k\} \circ\!\!\rightarrow \bigcup_i val(T_i) \mid \phi(n_i) \in val(T_i)\}$
- $val(*T) = val(T)^*$

Here * denotes *finite sequences* and $\circ\!\!\rightarrow$ denotes *partial functions.* If the type definitions involve recursion we obtain a set of (simultaneous) equations on sets involving these operations. The value sets are taken to be the unique least solutions to the equations; these always exist, since * and $\circ\!\!\rightarrow$ are both ω-continuous functions on sets (when the left-hand argument of $\circ\!\!\rightarrow$ is fixed).

Type Specific Manipulations

We introduce a number of type specific manipulations that allow us to write programs.

• For any type T it is possible to define named variables: **Var** $x : T$. Any variable may be used as an expression denoting its contents. Assignments $x := e$ and comparisons $e_1 = e_2$ are also possible for all types.

• For the simple types we have the usual constants and operations.

• For the partial product $P = (n_1 : T_1, \ldots, n_k : T_k)$ we allow the following manipulations. The expression $(m_1 : e_1, \ldots, m_q : e_q)$ denotes a value of type P if $\{m_j\} \subseteq \{n_i\}$ and whenever $m_j = n_i$ then e_j denotes a value of type T_i. If x is a variable of type P, then $\mathbf{has}(x, n_i)$ is a Bool-expression denoting whether n_i is in the domain of the value denoted by x; if so, then $x.n_i$ denotes the subvariable of type T_i containing this component. The statement $x\,{:}{+}(n_i : e_i)$ updates the n_i-component of x to contain the T_i-value denoted by e_i, and the statement $x\,{:}{-}n_i$ removes n_i from the domain of x.

• For the list type $L = *T$ we allow the expression $[e_0, \ldots, e_k]$ if each e_i denotes a value of type T. If x is a variable of type L and i is an Int-expression, then $x.(i)$ denotes the subvariable of type T containing the corresponding component, if it exists. The expression $|x|$ of type Int denotes the number of components of x.

Other manipulations could be introduced, but these are sufficient for the purposes of this paper.

Type Equivalence

Some type expressions allow exactly the same set of manipulations, and we do not wish to distinguish between these, but merely regard them as different syntactic versions of the same *type*; e.g., with the definitions

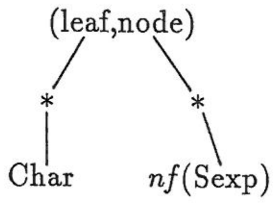
Type A = Int
Type B = A
Type C = (x: A, y: Int)
Type D = (x: B, y: A)

we want C and D to be equivalent. We shall define an equivalence relation $\approx$ to factor out these syntactic differences. Clearly $\approx$ must be a congruence with respect to the type constructors. The appropriate choice is the largest consistent congruence generated by the type equations. By *consistent* we mean that it does not identify any pair of types with different *outermost* type constructors. .

This is a very implicit definition; we can give a much more explicit one by associating with each type expression T a unique *normal form* $nf(T)$, which is a (possibly infinite) finite-branching labeled tree. The general idea is to repeatedly substitute right-hand sides of definitions for type names. If we regard the definitions

Type Atom = *Char
Type Sexp = (leaf: Atom, node: *Sexp)

we would expect the normal form nf(Sexp) to be the infinite tree



This is merely a short-hand notation for the full tree.

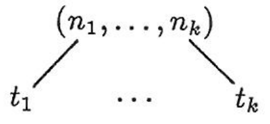
Formally, we need to get a handle on infinite trees. Regard the set $\mathcal{T}$ of all finite and infinite Σ-labeled trees (the set Σ will be evident from the definitions). We can define a *partial order* on $\mathcal{T}$. The relation $t_1 \sqsubseteq t_2$ holds iff we can obtain t_1 by replacing some subtrees of t_2 with the symbol Ω.[2] The structure $(\mathcal{T}, \sqsubseteq)$ is a *complete partial order* with the singleton tree Ω as the least element. In $\mathcal{T}$ we can obtain the normal forms of type *names* as limits. Apart from allowing a formal definition of normal forms, this ordering will later be refined to yield the desired type hierarchy.

Assume that our type definitions are

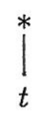
$$\begin{aligned}
&\textbf{Type}\ \ N_1 = F_1(N_1, N_2, \ldots, N_k)\\
&\textbf{Type}\ \ N_2 = F_2(N_1, N_2, \ldots, N_k)\\
&\quad\vdots\\
&\textbf{Type}\ \ N_k = F_k(N_1, N_2, \ldots, N_k)
\end{aligned}$$

Each F_i extends in the obvious manner to a k-ary function $F_i^{\mathcal{T}}$ on $\mathcal{T}$-trees; for single type constructors it goes as follows

- Int$^{\mathcal{T}}$, Bool$^{\mathcal{T}}$ and Char$^{\mathcal{T}}$ are constants yielding the singleton trees Int, Bool and Char.
- $N_i^{\mathcal{T}}$ is the same function as the right-hand side of its definition. If this chain of right-hand sides never reach a type constructor, then $N_i^{\mathcal{T}}$ is the constant function yielding Ω.
- $(n_1 : T_1, \ldots, n_k : T_k)$ correspond to the k-ary function mapping $t_1, \ldots, t_k \in \mathcal{T}$ to



- $*T$ correspond to the unary function mapping $t \in \mathcal{T}$ to



[2]This method (and the notation) is inspired by the infinite normal forms of the untyped λ-calculus [Barendregt84].

We now define a family of k-tuples of approximations to normal forms. The first approximation is the trivial one

$$(A_1^0, A_2^0, \ldots, A_k^0) = (\Omega, \Omega, \ldots, \Omega)$$

Suppose the i'th approximation is

$$(A_1^i, A_2^i, \ldots, A_k^i)$$

then we define the $i+1$'th approximation as

$$(F_1^{\mathcal{T}}(A_1^i, A_2^i, \ldots, A_k^i), F_2^{\mathcal{T}}(A_1^i, A_2^i, \ldots, A_k^i), \ldots, F_k^{\mathcal{T}}(A_1^i, A_2^i, \ldots, A_k^i))$$

Clearly each $\{A_j^i\}_{i \geq 0}$ forms an ascending chain in $\mathcal{T}$, so we can define

$$nf(N_j) = \lim_i A_j^i = \bigsqcup_i \{A_j^i\}$$

In the non-recursive case the chain of approximations will be finite. This extends in a natural way to normal forms of general type *expressions*, since the type constructors, regarded as operations on trees, are *continuous* in $(\mathcal{T}, \sqsubseteq)$.

We can now define

$$T_1 \approx T_2 \Leftrightarrow nf(T_1) = nf(T_2)$$

Obviously, this gives a congruence relation. Why is it the largest consistent one? Suppose that the congruence Ψ is larger. Then there must exists two types, A and B, such that $nf(A) \neq nf(B)$ and the relation $A \Psi B$ holds. The trees $nf(A)$ and $nf(B)$ must contain two subtrees with the same tree addresses but with different roots; otherwise, the trees would be identical by definition. These subtrees are normal forms of some types A' and B'. Since Ψ is a congruence, we have that $A' \Psi B'$, so that Ψ identifies two types with different outermost type constructors and, hence, violates consistency.

Thus, our equivalence construction is a *final* one, i.e. things are deemed equivalent unless there is some reason to conclude otherwise. Mutually recursive types are interesting to observe in connection with this. With the definitions

Type A = A
Type B = C
Type C = B

the types A, B, and C are all equivalent (and empty); their common normal form is the singleton tree Ω. This high-lights the finality; with an *initial* construction A would not be equivalent to B or C. For notational convenience we introduce the *type constant* Ω for which $nf(\Omega) = \Omega$. We could choose any of the above types as a representative for Ω.

The equivalence is decidable, since the infinite normal forms all have a very regular structure. One way to see this is to think of the equality of normal forms as being established through fixed point induction; another view is to observe, that equivalence of type expressions correspond to equivalence of certain tree-grammars with precisely one production for each non-terminal. A cubic algorithm is presented in an appendix.

3 The Type Ordering

We want to define a partial order on types (i.e. equivalence classes of type expressions), such that $T_1 \preceq T_2$ states that T_2 is larger than T_1, meaning that T_2 allows at least the same manipulations as T_1.

We can define $\preceq$ by refining the ordering $\sqsubseteq$ on normal forms. Since Ω manipulations are possible for all types, it is certainly the case that $\sqsubseteq$ satisfies the desired property. There is, however, a suitable ordering of partial products, that will also work. A product with more components allows at least the same manipulations as one with fewer components. Hence, we define $\preceq$ to be the smallest refinement of $\sqsubseteq$ that satisfies the rule

$$\begin{tikzcd}
& (m_1, m_2, \ldots, m_q) \arrow[dl, dash] \arrow[d, dash] \arrow[drr, dash] & & & & & (n_1, n_2, \ldots, n_k) \arrow[dl, dash] \arrow[d, dash] \arrow[drr, dash] & & \\
A_1 & A_2 & \ldots & A_q & \preceq & B_1 & B_2 & \ldots & B_k
\end{tikzcd}$$

$$\text{iff } \{m_1, m_2, \ldots, m_q\} \subseteq \{n_1, n_2, \ldots, n_k\} \text{ and } m_j = n_i \Rightarrow A_j \preceq B_i$$

We can now define

$$T_1 \preceq T_2 \Leftrightarrow nf(T_1) \preceq nf(T_2)$$

This yields a partial order on types since $\preceq$ is anti-symmetric, so

$$T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 \approx T_2$$

To illustrate this ordering, we can observe, that the relation

$$\begin{tikzcd}
(a, b) \arrow[d, dash] \arrow[dr, dash] & & & & (a, b, c) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\Omega & T_2 & \preceq & T_1 & T_2 & T_3
\end{tikzcd}$$

holds for all T_i. We observe the following facts

- Ω is the smallest type.
- If $T_1 \preceq T_2$ then $val(T_1) \subseteq val(T_2)$. This is easy to see, since $val(\Omega)=\emptyset$ and if the partial product P_1 is like P_2, except that it has fewer components, then $val(P_1) \subseteq val(P_2)$. The converse is *not* the case: the types $(x : \Omega)$ and $(y : \Omega)$ both have the value set[3] $\{()\}$, but they are clearly *incomparable*. Thus, the ordering is not definable on values alone. An analogy can be made with real and integer numbers. If we introduced a type Real, it would reasonably be the case that $val(\text{Int}) \subseteq val(\text{Real})$ whereas the relation Int $\preceq$ Real would *not* hold, since not all Int manipulations make sense for Reals. Certainly, the arithmetic operations could be extended to Reals, but things such as $x.(i)$ can not be interpreted if i is a real number. Hence, containment of values is a much weaker notion than $\preceq$; the former may be used to define *coercions*, but the latter is required for type hierarchies.

[3]() denotes the everywhere undefined function.

- The type constructors are monotonic and continuous with respect to the ordering.
- Many expressions, such as () have several different types, but there is always a unique smallest type, since greatest lower bounds exist[4].
- If **Type** $T = F(T)$ 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.
- The ordering is decidable. In fact, we can use the algorithm for deciding equivalence of type expressions, with only trivial modifications.

4 Consistency

The use of types guarantee that if a program is correctly typed, then certain type specific errors will never occur during its execution; for example, there is no attempt to add Int- and Bool-values, and lists are never confused with products. This guarantee may be construed as a notion of *consistency* of the programming language.

If we inspect the various manipulations that we allow (or could imagine allowing), it turns out that all the legal manipulations of a type are also possible for all *larger* types. This basically amounts to a few simple observations

- Ω manipulations are possible with any type.
- nothing is larger than a simple type.
- anything larger than a list is still a list.
- anything larger than a product is still a product, but with more components.
- values of small types are also values of large types.

The closure of these rules with respect to type construction and recursion yields the desired result.

This inspires us to extend the programming language to allow reuse of program fragments while maintaining consistency.

5 Procedures

Program fragments are typically expressed as *procedures*. The definition

```
Proc $P$(var $a : A$, $b : B$)
  $S$
end $P$
```

[4]Computed using *linearity* and the fact that Ω is smallest.

denotes a procedure P with a variable (reference) parameter a of type A and a value parameter b of type B; the body of the procedure is S.

We want to exploit that program fragments for small types may be reused for larger types. If we consider a procedure with a single parameter and no global variables, then this is just the situation we want. For example, the procedure

```
Type A = (a: Int)
Proc P(var x: A)
   if has(x,a) → x.a:=7 fi
end P
```

clearly works for all actual parameters of type $\succeq$ A. Hence, we define

- In a procedure call, the type of an actual parameter may be larger than that of the corresponding formal parameter.

To express true polymorphism, we must require a *uniform* execution of such procedure calls, i.e. it is not permissible to coerce or restrict the actual parameters. To motivate this, consider that the procedure

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

should act as the identity on all arguments of legal types. If we allow several parameters, then we get a possibility for confusion that we must rule out. The scenario

```
Type AB = (a: Int, b: Bool)
Type A = (a: Int)
Var z: AB
Proc P(var x: A, y: A)
   x:=y
end P

P(z,(a:7,b:'@'))
```

is troublesome, since the Bool-variable x.b is assigned a Char-value. Thus, we must insist upon the following *homogeneity* requirement

- If any two formal parameters have equivalent (sub-)types, then the corresponding (sub-)types of the actual parameters must also be equivalent.

A similar requirement is needed in the system of [Wirth88]. All this could be avoided if we introduced an *explicit* parameterization of procedures with types, but that would inflict an unnecessary notational burden.

With these flexible procedure calls, we can exploit the type ordering in programs, without risk of inconsistency. Since all the necessary requirements are decidable, we can still obtain static type checking.

Inconsistent extensions

At a glance it seems plausible to allow the assignment of large values to small variables, but this leads to inconsistency. Consider the scenario

```
Type AB = (a: Int, b: Bool)
Type A = (a: Int)
Var x: AB
Proc P(var y: A)
   y:=(a:7,b:'@')
end P

P(x)
```

We end up with a Char-value in a Bool-variable, so assignment is only possible between equivalent types. We could of course allow the *projective* assignment, but this seems in violation of the idea of a uniform execution; anyway, it can be done by explicit selection of components.

Inconsistency is also possible if procedures can access *global* variables. Consider the situation

```
Type AB = (a: Int, b: Bool)
Type A = (a: Int)
Var x: AB
Proc P(var y: A)
   Proc Q(z: A)
      y:=z
   end Q
   Q((a:7,b:'@'))
end P

P(x)
```

This is essentially the same situation as before.

6 Polymorphism

The type Ω allows us to write simple *polymorphic* procedures, such as

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

This procedure will work for *any* type, since Ω is smallest.

The (sensible) requirements for procedure calls impose the limitation, that we can only have a single type "variable". To rectify this situation, we introduce an infinite family of empty types

$$\{\square n\}_{n\in\mathcal{N}}$$

These types will work like *placeholders* or *type variables*, but they are just ordinary types. We want them to form a flat layer just above Ω, so we define them by

- $val(\square n) = \emptyset$
- $\Omega \prec \square n \prec T$, if T is not a $\square$-type or Ω

This mechanism is more sophisticated than it may seem. We can demand as much structure as we wish of the argument types. The following procedure works on all *lists*, but not on other types

```
Proc P(var x: *□a, y: □a)
   x.(0):=y
end P
```

Placeholder types can without problems mix with all the other (recursive) types.

7 Least Upper Bounds

Two types T_1 and T_2 may or may not have a *least upper bound* $T_1 \sqcup T_2$. For example, if

$$T_1 = (a : A, b : B) \text{ and } T_2 = (c : C)$$

then

$$T_1 \sqcup T_2 = (a : A, b : B, c : C)$$

The least upper bound of the two recursive types

$$T_1 = (x : T_1, y : \text{Int}) \text{ and } T_2 = (x : T_2, z : \text{Bool})$$

is the recursive type

$$T = (x : T, y : \text{Int}, z : \text{Bool})$$

The following pairs do not have any *any* upper bounds, let alone a least one

$$T_1 = (i : \text{Int}) \text{ and } T_2 = *\text{Bool}$$

$$T_1 = (i : \text{Int}, y : \text{Bool}) \text{ and } T_2 = (i : \text{Char})$$

Least upper bounds are interesting, as they include the *multiple inheritance* [Cardelli84] aspect of e.g. object oriented data values, a generalization of *prefixing* [Dahl70]. Notice, that the existence of the polymorphic types Ω and $\{\square n\}$ makes this a further generalization of multiple inheritance.

We can not elevate $\sqcup$ to a proper type constructor [5], but we can allow type definitions of the form

[5] Section 8 dismisses the obvious way of doing this.

Type $T = T_1 \sqcup T_2$

If the least upper bound does not exist, then this is an illegal definition, which would presumably result in a compiler error message; otherwise, the type T denotes the computed least upper bound.

The type constructors behave *linearly* with respect to the least upper bounds. The following properties hold, whenever the respective least upper bounds exist

- $T \sqcup \Omega = T$
- $T \sqcup T = T$
- $(x : T_1) \sqcup (y : T_2) = (x : T_1, y : T_2)$
- $(x : T_1) \sqcup (x : T_2) = (x : T_1 \sqcup T_2)$
- $*A \sqcup *B = *(A \sqcup B)$

Also, $\sqcup$ is monotonic and continuous, so that

- $A_1 \preceq A_2 \wedge B_1 \preceq B_2 \Rightarrow A_1 \sqcup A_2 \preceq B_1 \sqcup B_2$
- $(\bigsqcup_i A_i) \sqcup (\bigsqcup_i B_i) = \bigsqcup_i (A_i \sqcup B_i)$

Using these properties it is easy to modify the equivalence/ordering algorithm to compute the least upper bound of two types, or to decide that none exist.

Least Upper Bounds and Recursive Types

It seems restrictive to allow only a single $\sqcup$ on the right-hand side of a type definition. Clearly, we can make sense of a definition like

Type $T = E$

where E is an arbitrary type expression, possibly containing $\sqcup$'s. We just compute the type denoted by E if it exists. If we (quite naturally) allow recursion to enter this game, then the situation becomes considerably more complex. What sense should we make of the definition

Type A = A $\sqcup$ Int

The proper answer is that A equals Int, since this is the smallest type satisfying the equation. Similarly, the definition

Type B = (x:B) $\sqcup$ (y:Int)

denotes the type B = (x:B, y:Int). In contrast, the equation

Type C = (x: C $\sqcup$ Int)

has no solution, since simple types and products are incompatible. More exotic specimens such as

Type D = (x: D $\sqcup$ (x:D))
Type E = $*$(x: E $\sqcup$ $*$E) $\sqcup$ E
Type F = $*$F $\sqcup$ F

can not be analyzed at a glance.

We need a general method to resolve such situations. Since we do not add any new types, we want an algorithm that given a type equation containing $\sqcup$'s will tell us whether it has a solution and, if so, will disclose a type equation without $\sqcup$'s that denotes this solution.

Consider a type equation

Type $T = F(T)$

where F may contain several $\sqcup$'s. If any solution is to exist, then it must be the limit of the chain

$$\Omega \preceq F(\Omega) \preceq F^2(\Omega) \preceq \cdots \preceq F^i(\Omega) \preceq \cdots$$

provided that all the approximants exist (this is a chain because $\sqcup$ is monotonic in both arguments).

We need to define a transformation on type functions; if F is a type expression with a free variable T, then $\nabla_T F$ (or just ∇F) is a reduced $\sqcup$-free version, computed under the assumption that T is smaller than all other types, except Ω. Another way of putting this is that T is substituted with a *colored* [6] Ω (which is larger than regular Ω's), then F is reduced, and finally T is substituted back for the colored Ω's. The reduction ∇F may or may not exist.

We now claim that a necessary condition for the equation $T = F(T)$ to have a solution is that ∇F exists. This is easy, since otherwise the first approximant $F(\Omega) = \nabla F(\Omega)$ can not exist.

In fact, the chain $\{F^i(\Omega)\}$ is *equal* to the chain

$$\Omega \preceq \nabla F(\Omega) \preceq \nabla(F^2)(\Omega) \preceq \cdots \preceq \nabla(F^i)(\Omega) \preceq \cdots$$

since it does not matter if we apply ∇ or not, as we shall later substitute with Ω.

The crucial observation is that

$$\forall i \geq 0 : \ \nabla(F^i)(\Omega) = (\nabla F)^i(\Omega)$$

To realize this, just think of the colored Ω's as being present all along.

But this means, that if the equation $T = F(T)$ has a solution, then ∇F must exist and T is the limit of the chain

$$\Omega \preceq \nabla F(\Omega) \preceq (\nabla F)^2(\Omega) \preceq \cdots \preceq (\nabla F)^i(\Omega) \preceq \cdots$$

i.e. the recursive type defined by

[6]This is needed to avoid confusion with any Ω *constants* in F.

Type $T = \nabla F(T)$

which does not involve $\sqcup$'s.

In summary, to resolve a recursive equation with $\sqcup$'s, we first check that ∇F exists and then verify that the type $T = \nabla F(T)$ satisfies the original equation.

Now we can analyze the types D, E, and F with ease by computing the reductions

Type D = ∇(x: D $\sqcup$ (x:D)) = (x:(x:D))
Type E = ∇*(x: E $\sqcup$ *E) $\sqcup$ E = *(x: *E)
Type F = ∇(*F $\sqcup$ F) = *F

By inspection we see that D and F are solutions, whereas E is not.

The proposed method generalizes without problems to mutually recursive type equations.

8 Type Completeness

The ordered collection of types has a fairly rich structure by now, but the structure is far from *complete*. Some pairs of types have least upper bounds, whereas others do not. Recursive types give rise to chains with limits, whereas the chain

$$\Omega \preceq (n_1 : T_1) \preceq (n_1 : T_1, n_2 : T_2) \preceq (n_1 : T_1, n_2 : T_2, n_3 : T_3) \preceq \cdots$$

does not have a limit. From a purely algebraic viewpoint it is tempting to close this structure by adding a *top* element Θ, i.e. a type which is greater than all others. This would allow us to make $\sqcup$ into a proper type constructor. However, such a step leads to inconsistency. Consider the scenario

```
Proc P(var i: Int)          Proc Q(var b: Bool)
  i:=7                        ...
end P                       end Q

Var t: Θ
P(t)
Q(t)
```

This is legal since Θ is larger than both Int and Bool. But inside Q the Bool-variable b contains an Int-value. In a sense, the introduction of the type Θ corresponds to the abolition of any type discipline. Values of type Θ can be thought of as *bitstrings* on which no typing is performed.

A more careful process of adding individual upper bounds and limit points could possibly make sense, but we can think of no applications for this.

9 Structural Invariants

The partial product became a fact of life in order to achieve consistency of the type hierarchy. Fortunately, there are important uses for it, too.

The partiality of the product allows for very compact definitions of recursive types. The usual definition of a binary tree of integers use type *sums* and *products*

> **Type** Tree = **Sum**(Empty: Unit, NonEmpty: Node)
> **Type** Node = **Prod**(val: Int, left,right: Tree)

where Unit is a singleton type with value set {•}. Here the empty tree is denoted (Empty: •) and a one-element tree is denoted

> (NonEmpty: (val: 7, left: (Empty: •), right: (Empty: •)))

With the partial product we could write the definition as

> **Type** Tree = (val: Int, left,right: Tree)

The empty tree is denoted () and the one-element tree is denoted

> (val:7, left:(), right:())

This alleviates some of the notational burden that seem to disadvantage recursive type definitions.

We must, however, confront the problem that the above definition strictly speaking does not define a type of binary trees. It contains a value like

> (left:(val:7))

which is undesirable, since an application might rely on all three components being present in non-empty trees. The natural solution would be to decide upon an *invariant* to rule out such "odd" values, but it would be much more satisfying to be able to obtain this precision using type constructors. For this purpose we introduce the concept of structural invariants that may be associated with product definitions.

A *structural invariant* S over a partial product type

$$(n_1 : T_1, \dots, n_k : T_k)$$

is a set of subsets of $\{n_1, \dots, n_k\}$ such that if ϕ is a value of the product then it is guaranteed that $dom(\phi) \in S$. Formally, we must associate the component names with the invariant; we call this the *basis* of the invariant and write *basis*$(S) = \{n_1, \dots, n_k\}$.

Now we can enhance type definitions with invariants in the following way

> **Type** Tree = (val: Int, left,right: Tree) ! {∅, {val,left,right}}

This invariant indicates that either *all* or *none* of the components must be present. One could develop a number of standard invariants, so that the definition might look like

$$\textbf{Type}\ \text{Tree} = (\text{val: Int, left,right: Tree})\ !\ \textbf{ext}$$

We formally extend the type expressions so that structural invariants are explicitly included

$$\begin{array}{ll} \text{T} ::= \text{Int} \mid \text{Bool} \mid \text{Char} \mid & \\ \quad N_i & \\ \quad (n_1 : T_1, \ldots, n_k : T_k)\ !\ S & k \geq 0,\ n_i \in \mathcal{N},\ \textit{basis}(S) = \{n_1, \ldots, n_k\} \\ \quad *T & \end{array}$$

Some interesting standard invariants, with basis $\{n_1, \ldots, n_k\}$ could be

- **ext** $= \{\emptyset, \{n_1, \ldots, n_k\}\}$
- **prod** $= \{\{n_1, \ldots, n_k\}\}$
- **sum** $= \{\{n_1\}, \ldots, \{n_k\}\}$
- **true** $= \mathcal{P}\{n_1, \ldots, n_k\}$
- **false** $= \emptyset$

These are far from sufficient, though. The *variant record* concept of many imperative languages is quite cumbersome to emulate with sums and products. The trouble is that components belonging to the same conceptual level appear at different syntactic levels. A typical example is a symbol table, where we want a fixed information for each symbol (such as name and static level) and a variable information for each kind of symbol (e.g. type and procedure); this we can express as follows

$$\textbf{Type}\ \text{Symbol} = (\text{n: Text, s: Int, p: P, t: T})\ !\ \{\{\text{n,s,p}\}, \{\text{n,s,t}\}\}$$

More elegance could be achieved by defining a logical notation, such as

$$\textbf{Type}\ \text{Symbol} = (\text{n: Text, s: Int, p: P, t: T})\ !\ \{\text{n} \wedge \text{s} \wedge (\text{p} \mid \text{t})\}$$

where | is *exclusive-or*. If we had the type

$$\textbf{Type}\ T = (a : A, b : B, c : C, d : D)$$

and wanted to ensure that if we had the a-component then we also had the d-component, then we could write the invariant as

$$\{a \Rightarrow d\}$$

It would be awkward to express this using type sums and products. The authors' best attempt is the definition

$$\begin{array}{l} \textbf{Type}\ \text{T} = \textbf{Prod}(b : B, c : C, \textbf{Sum}(\text{None : Unit}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{One} : \textbf{Prod}(d : D), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{Two} : \textbf{Prod}(a : A, d : D))) \end{array}$$

In this way, a more clever binary tree could be defined by

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

with which the singleton tree could be denoted (with ultimate simplicity) by

(val:7)

This leaves us with the task of extending the equivalence and the ordering to fit these new definitions. The equivalence is easy to deal with. When should $P_1\ !\ S_1$ be equivalent with $P_2\ !\ S_2$? Clearly, the product parts must still be equivalent, and since equivalence will allow assignments, the invariants must be identical. Hence, we define

$$P_1\ !\ S_1 \approx P_2\ !\ S_2 \Leftrightarrow P_1 \approx P_2 \wedge S_1 = S_2$$

Structural Invariants and the Hierarchy

The ordering is more complicated to mend. When should $P_1\ !\ S_1$ be smaller than $P_2\ !\ S_2$? Obviously, we must have $P_1 \preceq P_2$, but what are the requirements on S_1 and S_2? We must seek inspiration in the definition of consistency and extend it by demanding that all invariants must be respected at all times.

Initially we need a few operations on invariants. Let X and Y be invariants; then we define

- $X \times Y = \{x \cup y \mid x \in X \wedge y \in Y\}$ $\qquad$ $basis(X \times Y) = basis(X) \cup basis(Y)$
- $X/Y \ = \{x - basis(Y) \mid x \in X\}$ $\qquad$ $basis(X/Y) \ = basis(X) - basis(Y)$
- $X \downarrow Y = \{x \cap basis(Y) \mid x \in X\}$ $\qquad$ $basis(X \downarrow Y) = basis(X) \cap basis(Y)$

Invariants with $\emptyset$, $\{\emptyset\}$, $\times$ and $/$ behave somewhat like the integers with 0, 1, multiplication and division; among others, the following equations hold

- $X \times \emptyset = \emptyset$ $\qquad$ • $X \times Y = Y \times X$
- $X \times \{\emptyset\} = X$ $\qquad$ • $X \times (Y \times Z) = (X \times Y) \times Z$
- $X/X = \{\emptyset\}$ $\qquad$ • $(X/Y)/Z = X/(Y * Z)$

It is not the case, however, that $(X \times X)/X = X$.

Returning to the question of $P_1\ !\ S_1 \preceq P_2\ !\ S_2$, we can firstly observe that if a formal parameter with an S_1-invariant must accept an actual parameter with an S_2-invariant, then it must be the case that

$$S_2 \downarrow S_1 \subseteq S_1$$

This states that the P_1-part of any P_2-parameter must still satisfy the S_1-invariant; if not, the invariant would be violated inside the procedure.

Thinking about variable parameters we realize that the P_1-part of any actual parameter can be substituted[7] with any P_1-value satisfying S_1. Hence, the remaining $(P_2–P_1)$-part of

[7]Using the :+ and :- manipulations.

the actual parameter must combine with any S_1-value and still satisfy S_2. This is possible exactly when

$$S_1 \times (S_2/S_1) \subseteq S_2$$

These two conditions are equivalent to the single condition

$$S_1 \times (S_2/S_1) = S_2$$

which we can call S_1 *divides* S_2 and write as $S_1|S_2$. This fact is easily seen. If the two conditions hold, we must prove that $S_1 \times (S_2/S_1) \supseteq S_2$. Consider any $x \in S_2$. We can write it as $x = x' \cup x''$ where $x' \in basis(S_2) - basis(S_1)$ and $x'' \in basis(S_1)$ Hence, we conclude that $x' \cup x'' \in (S_2 \downarrow S_1) \times (S_2/S_1)$. By assumption $S_2 \downarrow S_1 \subseteq S_2$, so we find that $x = x' \cup x'' \in S_2 \times (S_2/S_1)$, as desired. Conversely, if $S_1|S_2$ then $S_2 \downarrow S_1 = S_1$, so in particular $S_2 \downarrow S_1 \subseteq S_1$ holds.

Finally, a variable parameter can be assigned a constant of the formal parameter type. This necessitates that $S_1 \subseteq S_2$.

Hence, we define

$$P_1 \,!\, S_1 \preceq P_2 \,!\, S_2 \Leftrightarrow P_1 \preceq P_2 \wedge S_1 \preceq S_2$$

where the ordering on invariants is defined by

$$S_1 \preceq S_2 \Leftrightarrow S_1|S_2 \wedge S_1 \subseteq S_2$$

The two requirements on the invariants are independent. If we look at the two invariants $X = \{\emptyset, \{a\}\}$ and $Y = \{\emptyset, \{a\}, \{b\}\}$ then $X \subseteq Y$ but $X \times (Y/X) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ which does not equal Y. Conversely, if $X = \{\{a\}\}$ and $Y = \{\{a, b\}\}$ then $X|Y$ but $X \not\subseteq Y$.

An example of the relationship $X \preceq Y$ is

$$X = \{\emptyset, \{a\}, \{a, b\}\}$$

$$Y = \{\emptyset, \{a\}, \{c\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$$

We find that

$$X \times (Y/X) = X \times \{\emptyset, \{c\}\} = Y$$

so $X|Y$, and clearly $X \subseteq Y$. This illustrates a particularly useful case. If we extend a partial product type with a new component n, then we can extend the invariant S to $S \times \{\emptyset, \{n\}\}$, which is *larger* than S.

Clearly, $S_1 \preceq S_2$ is decidable, so static type checking is maintained.

We can think of an ordinary partial product with component names B as implicitly equipped with the invariant $\mathbf{True} = 2^B$ (the basis B is implicit in $\mathbf{True}$). We can observe, that if $P \,!\, S \preceq Q \,!\, \mathbf{True}$ holds, then $S|\mathbf{True}$, so $\mathbf{True} = S \times (\mathbf{True}/S) = S \times \mathbf{True}$; this implies that $S = \mathbf{True}$. Hence, $\mathbf{True}$ does not have any non-trivial divisors.

The type $\mathbf{List}(T)$ may be thought of as an infinite homogeneous partial product $(i : T)_{i \geq 0}$ with the invariant $\{\emptyset, \{0\}, \{0, 1\}, \{0, 1, 2\}, \ldots\}$.

Least Upper Bounds and Structural Invariants

What is an upper bound for partial products with structural invariants? We must expect to define

$$P_1 \mathbin{!} S_1 \sqcup P_2 \mathbin{!} S_2 = P_1 \sqcup P_2 \mathbin{!} S_1 \sqcup S_2$$

where $S_1 \sqcup S_2$ is the least invariant such that $S_i \preceq S_1 \sqcup S_2$. Such an invariant may or may not exist. We can provide a method for computing the least upper bound or deciding than none exists. We need a new operation on invariants

- $X \star Y = \{z \in \mathit{basis}(X) \cup \mathit{basis}(Y) \mid z \cap \mathit{basis}(X) \in X \wedge z \cap \mathit{basis}(Y) \in Y\}$

where $\mathit{basis}(X \star Y) = \mathit{basis}(X) \cup \mathit{basis}(Y)$. We now claim that

$$S_1 \preceq S \wedge S_2 \preceq S \Rightarrow S = S_1 \star S_2$$

To see this we choose any $z \in S_1 \star S_2$. Now, define $p = z \cap \mathit{basis}(S_1) \in S_1 \subseteq S$ and $q = z \cap \mathit{basis}(S_2) \in S_2 \subseteq S$. Let $r = p \cap q$. Then $r \in S \downarrow S_2 \subseteq S_2$, and $p - r \in S/S_2$. Hence, $z = q \cup (p - r) \in S_2 \times (S/S_2) = S$, so $S_1 \star S_2 \subseteq S$. If we have any $z \in S - S_1 \star S_2$, then $z \cap \mathit{basis}(S_1) \notin S_1$ or $z \cap \mathit{basis}(S_2) \notin S_2$, so $S \downarrow S_1 \not\subseteq S_1$ or $S \downarrow S_2 \not\subseteq S_2$, which leads to the contradiction $S_1 \not\preceq S$ or $S_2 \not\preceq S$, so $S = S_1 \star S_2$.

Thus, we only have one candidate for upper bound, so if it exists it must be the least one. In particular, notice that invariants with disjoint bases always have a least upper bound.

10 Conclusion

We have modified the intuitive imperative type hierarchy to achieve a notion of consistency. This involved a new type constructor: the partial product. This, together with the concept of structural invariants, yield a notation for defining recursive types that is superior to both the usual recursive type definitions and the traditional pointers and variant records. Least upper bounds, which generalize multiple inheritance, have been smoothly integrated with strutural invariants and recursion.

References

[Barendregt84] H. Barendregt. "The Lambda Calculus, Its Syntax and Semantics.", North Holland, 1984.

[Cardelli84] L. Cardelli. "A semantics of multiple inheritance." In *Semantics of Data Types*, LNCS 173, Springer-Verlag 1984.

[Cardelli85] L. Cardelli. "Amber." Proceedings of the *Treizieme Ecole de Printemps d'Informatique Theorique*, May 1985.

[Dahl70] O.J. Dahl, B. Myrhaug, K. Nygaard, "Simula Information, Common Base Language.", Norwegian Computing Center, October 1970.

[Reynolds85] J.C. Reynolds, "Three approaches to type structure.", In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.

[Tennent88] R.D. Tennent, "Denotational Semantics of Algol-Like Languages.", To appear in *Handbook of Logic in Computer Science*, Vol.II, Oxford University Press.

[Wirth88] N. Wirth, "Type Extensions.", In *Transactions on Programming Languages and Systems*, Vol.10, No.2, 1988.

Appendix: Decidability

Normal forms are limit points, and properties about them can be verified using *fixed point induction.* In this context we are interested in equality of limit points, which is clearly an admissible predicate for fixed point induction. Given the tuple of normal forms, we wish to prove that two of them are equal. Our base case is

$$(\Omega, \Omega, \ldots, \Omega)$$

where the equality is trivially true. For the induction we must prove that the iteration preserves the equality. Given two type names it is easy to check this property of the corresponding tree iterator. Obviously, this extends to a method for deciding the equivalence of general type expressions, since the type constructors preserve equality of normal forms.

The following algorithm is the general one, that decides the equivalence of two arbitrary type expressions. Firstly, we need a procedure **Strip** that reduces a type name to an equivalent type expression with an outermost type constructor. This may not always be possible. A *vacuous* type name such as

$$\textbf{Type}\ \ \mathrm{Nil} = \mathrm{Nil}$$

cannot reduce to a type constructor. Such identifiers are all equivalent type expressions; their normal form is the singleton tree Ω; we let their reduction be the symbol Ω. We obtain **Strip** as follows:

```
Strip($T$): return S($T, \emptyset$)

S($T, N$):  if $T \in N \rightarrow$ return $\Omega$ fi
            if $T$ is an identifier $\rightarrow$ return S($rhs(T), N \cup \{T\}$) fi
            return $T$
```

This procedure runs in time $O(k)$, where k is the number of type definitions under consideration.

Using **Strip** we can define a procedure **Equiv** that decides the equivalence of two type expressions

```
Equiv($S,T$): return E($S,T,\emptyset$)

E($S,T,A$):   if $S$ and $T$ are id's $\rightarrow$ if $(S,T) \in A \rightarrow$ return true fi
                                      $A := A \cup \{(S,T)\}$
              fi
              $S,T :=$ Strip($S$), Strip($T$)
              if $S$ and $T$ are basic types $\rightarrow$ return $A \equiv B$ fi
              if $S$ and $T$ have different outermost constructors $\rightarrow$ return false fi
              let $S_1, S_2, \ldots, S_n$ be the sub-type expressions of $S$
              let $T_1, T_2, \ldots, T_n$ be the sub-type expressions of $T$
              return $\bigwedge_i$ E($S_i, T_i, A$)
```

This procedure runs in time $O(k^3 + k(|S| + |T|))$, where again k is the number of type definitions under consideration.