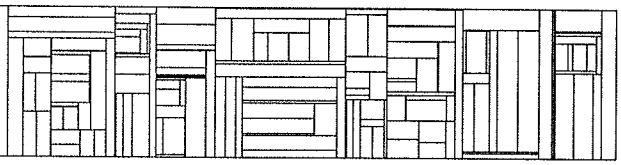
ISSN 0105-8517

A Note on Opaque Types

Michael I. Schwartzbach

DAIMI PB – 313
April 1990

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



A Note on Opaque Types

Michael I. Schwartzbach[1]

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

Abstract

We extend the results in [1] to include *opaque* types. An opaque version of a type is different from the original but has the same values and the same order relations to other types. The opaque types allow a more flexible polymorphism and provide the usual pragmatic advantages of distinguishing between intended and unintended type equalities. Opaque types can be viewed as a compromise between *synonym* types and *abstract* types.

1 Introduction

This note extends the results in [1] and is based on the definitions and results presented there.

A transparent type definition such as

Type Money = Int

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

Type Apples = Int
Type Oranges = Int

then it is possibly a mistake to compare such values.

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

The usual alternative is an *abstract* type definition where the representation type is completely hidden. This certainly provides the desired protection. However, it is now necessary to re-implement all the standard Int operations for the abstract type. This is clearly unwanted in this situation and a high price to pay for protection.

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

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

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

In the following sections we incorporate opaque types into the hierarchical type system presented in [1]. We indicate the minor modifications that are required to carry all major results through. As a very significant special case we obtain a more flexible hierarchical polymorphism by using opaque versions of the type Ω as type "variables".

2 Opaque Types

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

We extend the language of types as follows

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

Here n and the n_i's are names. Notice that type definitions may involve arbitrary recursion.

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

Type Equivalence

In [1] type equivalence is defined to be equality of normal forms. The normal form of a type is a (possibly infinite) labeled tree that, informally, is obtained by the unfolding of the type definitions. This technique generalizes without problems, so that

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

Thus, among the following types

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

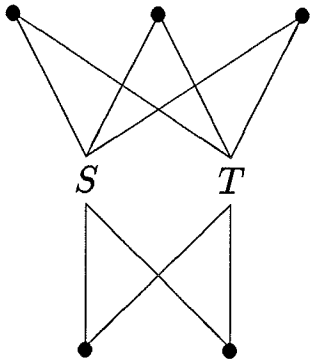
only B and E are equivalent. Type equivalence is still computable.

Type Ordering

The type ordering in [1] concerns itself with possibilities for *code reuse.* The idea is that code written for smaller types can be reused for larger types. For this purpose we want to ignore the protection offered by opacity. Thus, the finite ordering $\preceq_0$ must further satisfy

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

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



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

The Language

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

$$\tau \ ::= \ n\square\tau$$

For convenience, we also introduce type equations of the form

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

They abbreviate the more involved equations

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

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

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

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

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

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

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

3 Extended Types

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

To handle this situation we extend the x-types to

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

The elements of $\Box X$ are the elements of X and their opaque versions.

The computations on x-types must be modified as follows

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

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

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

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

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

- $X \otimes \Box X = X$
- $\Box X_1 \otimes \Box X_2 = \Box(X_1 \otimes X_2)$
- $n\Box T \otimes \Box X = n\Box T$

Proposition 5.10: 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* $(\forall i, j : \; n_i = m_j \Rightarrow S_i \lhd X_j)$
- $n\Box S \lhd X$ *iff* $S \lhd X$
- $S \lhd \Box X$ *iff* $S \lhd X$

All proofs of propositions in section 5 in [1] generalize without difficulties.

4 Correctness

The extensions in the preceding section allow us once again to assign unique x-types to expressions

Definition 5.13: If $\mathcal{E}$ is an environment and ϕ is an expression, then $\mathcal{E}[\![\phi]\!]$ is defined inductively as follows

- $\mathcal{E}[\![0]\!] = \Box\text{Int}$
- $\mathcal{E}[\![\phi{+}1]\!] = \mathcal{E}[\![\phi{-}1]\!] = \mathcal{E}[\![\phi]\!]$
- $\mathcal{E}[\![\sigma]\!] = \mathcal{E} \downarrow \sigma$

- $\mathcal{E}[\![\phi_1 = \phi_2]\!] = \Box\mathrm{Bool}$
- $\mathcal{E}[\![[\phi_1,\ldots,\phi_k]]\!] = \Box*(\otimes_i \mathcal{E}[\![\phi_i]\!])$, if $k > 0$
- $\mathcal{E}[\![[]]\!] = \Box\Lambda$
- $\mathcal{E}[\![|\phi|]\!] = \Box\mathrm{Int}$
- $\mathcal{E}[\![(n_i{:}\phi_i)]\!] = \Box\Pi(n_i : \mathcal{E}[\![\phi_i]\!])$
- $\mathcal{E}[\![\mathbf{has}(\phi, n_i)]\!] = \Box\mathrm{Bool}$

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

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

The proofs of lemmas 6.5, 6.6, 6.7, 6.9, 6.11 and 6.12 only require minor modifications to handle the extra cases in the structural induction. The proofs of the main results, lemma 6.10 and theorem 6.4, can remain unchanged. The proof of optimality in theorem 6.16, only requires a trivial modification of lemma 6.15. All of section 7 in [1] go through unchanged.

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

5 Hierarchical Procedures

As demonstrated in the introduction, it is pragmatically useful to distinguish between intended and unintended type equalities. In connection with the polymorphic mechanism of [1] opaque types can serve another important function.

A hierarchical call of a procedure such as

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

requires that the actual types of x and y are equal, since their formal types are equal. However, since the procedure keeps the two variables separate this is actually too strict. By specifying the formal types as two

opaque versions of Ω we guarantee that they will never be mixed and, hence, we can allow more hierarchical calls of the procedure.

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

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

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

⋮
```

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

6 Conclusion

The introduction of opaque types seems to fill a gap between synonym types and abstract types. Another view is that they provide a unification of structural and name equivalence of types; the programmer can decide on the combination which is most suited for the application.

Opaque types have been smoothly integrated with the hierarchical system [1]; they can even be seen to increase the available polymorhic flexibility.

Acknowledgement: The idea to obtain extra type variables as opaque versions of Ω originated through discussions with Jens Palsberg.

7 References

[1] **Schwartzbach, Michael I.** "Static Correctness of Hierarchical Procedures" in *Proceedings of ICALP'90, LNCS*, Springer-Verlag, 1990.