Transparent and Opaque Interpretations of Datatypes

Karl Crary Robert Harper Perry Cheng
Leaf Petersen Chris Stone

November 1998

CMU-CS-98-177

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-98-04

This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: ML, recursive types, abstract data types, translucent types, typed compilation.

Standard ML employs an opaque (or *generative*) interpretation of datatype specifications, in which every datatype specification provides a new, abstract type that is different from any other type, including other identically specified datatypes. An alternative interpretation is the transparent one, in which a datatype specification exposes the underlying recursive type implementation of the datatype.

It is commonly believed that the transparent interpretation is strictly more permissive than the opaque interpretation; that all programs typable under the opaque discipline are also typable under the transparent discipline. The purpose of this note is to illustrate that this common belief is incorrect (in the usual equational theory for types), and to discuss some of the implications of that fact.

1 An Example

To see the issue involved, consider the signatures `SIG1` and `SIG2`:

```
signature SIG1 =
   sig
      datatype u = C of u * u | D of int
      type t = u * u
   end
signature SIG2 =
   sig
      type t
      datatype u = C of t | D of int
   end
```

Is `SIG1` a subsignature of `SIG2`? In an opaque interpretation (and in Standard ML [3]) the answer is yes. But in a transparent interpretation the answer is no. To show why this is so, we give the opaque and transparent interpretations of `SIG1` and `SIG2` in a type theory without datatypes but with sums and iso-recursive types (recursive types in which fold and unfold must be mediated by an explicit isomorphism).

In an opaque interpretation, a datatype specification provides an abstract type along with introduction and elimination functions for that type [2]:

```
signature SIG1_opaque =
   sig
      type u
      type t = u * u
      val u_in  : (u * u + int) -> u
      val u_out : u -> (u * u + int)
   end
signature SIG2_opaque =
   sig
      type t
      type u
      val u_in  : (t + int) -> u
      val u_out : u -> (t + int)
   end
```

In this interpretation, SIG1 matches SIG2 because $(\mathtt{u} * \mathtt{u} + \mathtt{int}) \to \mathtt{u}$ is equal to $(\mathtt{t} + \mathtt{int}) \to \mathtt{u}$ under the assumption that $\mathtt{t} = \mathtt{u} * \mathtt{u}$, and similarly $\mathtt{u} \to (\mathtt{u} * \mathtt{u} + \mathtt{int})$ is equal to $\mathtt{u} \to (\mathtt{t} + \mathtt{int})$.

However, in a transparent interpretation, a datatype specification exposes the underlying recursive type:

```
signature SIG1_transparent =
   sig
      type u = μα. α * α + int
      type t = u * u
   end
signature SIG2_transparent =
   sig
      type t
      type u = μα. t + int
   end
```

In this interpretation, SIG1 does not match SIG2 because u's abbreviation in SIG1 is not equal to its abbreviation in SIG2. Invoking $\mathtt{t} = \mathtt{u} * \mathtt{u}$, the latter may be shown equal to

$$\mu\alpha.\,(\mu\alpha.\,\alpha * \alpha + \mathtt{int}) * (\mu\alpha.\,\alpha * \alpha + \mathtt{int}) + \mathtt{int}$$

which, in the usual equational theory for types, is not the same as SIG1's abbreviation:

$$\mu\alpha.\,\alpha * \alpha + \mathtt{int}$$

What is happening here is, in order for SIG1 to match SIG2, the datatype specification for u in SIG2 must be able to "capture" a definition given to t, even when t is defined in terms of u. This is possible in the opaque setting because t and u are independent abstract types, and any interplay between them is deferred to value fields. In a transparent setting, the necessary capture is impossible; u and α are different variables and the recursive binding of α cannot capture any occurrences of u.

2 Implications

This example illustrates that under the usual equality rules for iso-recursive types, Standard ML is incompatible with a transparent interpretation. However, in an implementation it is unacceptable to incur the cost of a function call for every datatype construction and pattern match, so the transparent interpretation is required. In a type-preserving compiler, one may adopt internally a new interpretation of the language, but only when that internal interpretation is at least as permissive as the external one, which we have shown is not the case here. This poses no problem to those compilers that erase types before compiling, but how can Standard ML be implemented in a type-preserving manner?

Shao, in the FLINT compiler [5], addresses this problem with what we call "Shao's equation" (where we write $E[E'/X]$ to mean the capture-avoiding substitution of E' for X in E):

$$\mu\alpha.\tau = \mu\alpha.(\tau[\mu\alpha.\tau/\alpha])$$

Shao's equation addresses the problem with the example above by rendering the two abbreviations equal. More generally, for any equational theory one may prove that if the transparent interpretation accepts every program typable under the opaque interpretation, then that theory must include all instances of Shao's equation. Thus, we argue that Shao's equation is *essential* to efficient, type-preserving compilation of languages with opaque datatypes, such as Standard ML.

Note that this equation falls short of the equation for equi-recursive types (recursive types in which fold and unfold need not be performed explicitly):

$$\mu_{\text{equi}}\,\alpha.\tau = \tau[\mu_{\text{equi}}\,\alpha.\tau/\alpha]$$

Since the right-hand side of Shao's equation is still a recursive type (in contrast to the right-hand side of the equi-recursive type equation) it is possible that the type equality problem with Shao's equation may be solved more efficiently than the problem for equi-recursive types [1]. Indeed, Shao claims to have an efficient algorithm for the problem [4].

Nevertheless, there is some question as to the validity of Shao's equation. In many semantic contexts, though certainly not all, the equation may be justifiable. Note that terms having the left-hand side type,

$$\mu\alpha.\tau \stackrel{\text{def}}{=} \tau_{\text{left}}$$

and terms having the right-hand side type,

$$\mu\alpha.(\tau[\mu\alpha.\tau/\alpha]) \stackrel{\text{def}}{=} \tau_{\text{right}}$$

both unfold to members of the same type:

$$\tau[\mu\alpha.\tau/\alpha]$$

Thus, terms may be coerced from one type to the other by unfolding them at one type and refolding them at the other type. For instance, if e has type τ_{left}, then $\mathtt{unfold}[\tau_{\mathrm{left}}]e$ has type $\tau[\mu\alpha.\tau/\alpha]$, and so $\mathtt{fold}[\tau_{\mathrm{right}}](\mathtt{unfold}[\tau_{\mathrm{left}}]e)$ has type τ_{right}.

Thus, Shao's equation is justifiable in a semantic framework in which such an fold-unfold operation (at different types) is the identity. (Fold-unfold at the same type would be the identity in nearly any semantic framework.) A particularly important case where this is true is when fold and unfold themselves are no-ops, as is the case in most implementations.

3 Conclusions

Opaque datatypes are purported to carry software engineering benefits, but datatypes must be transparent, at least internally, to achieve efficient compilation. Were the transparent discipline more permissive than the opaque one, this would not pose a problem, but we show that this is not so.

The opaque and transparent disciplines can be reconciled only by adopting Shao's equation. Therefore, we argue that Shao's equation is essential to efficient, type-preserving compilation of any language with opaque datatypes. This equation is not valid in every semantic context, and although it may be permissible in many important ones, at the very least it complicates typechecking. Thus, there are good reasons why one could prefer to reject Shao's equation. However, in a type-preserving compiler, if we wish not to embrace Shao's equation, we are left with no choice but to abandon opaque datatypes as well.

References

[1] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[2] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998. To appear.

[3] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[4] Zhong Shao. Personal communication.

[5] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

A Proof

Suppose an equational theory is given and suppose that the transparent interpretation accepts every program typable under the opaque interpretation. Let $\tau[\alpha]$ be an arbitrary type with free variable α and let `SIG1'` and `SIG2'` be defined as follows:

```
signature SIG1' =
   sig
      datatype u = C of τ[u]
      type t = τ[u]
   end
signature SIG2' =
   sig
      type t
      datatype u = C of t
   end
```

In the opaque interpretation `SIG1'` matches `SIG2'`, so `SIG1'` must match `SIG2'` under the transparent interpretation as well. In `SIG1'` the abbreviation for `u` is

$$\mu\alpha.\tau[\alpha]$$

and in `SIG2'` it is

$$\mu\alpha.\mathtt{t}$$

which, invoking $\mathtt{t} = \tau[\mathtt{u}]$, is equal to

$$\mu\alpha.\tau[\mu\alpha.\tau[\alpha]]$$

Since these abbreviations must be equal, we conclude

$$\mu\alpha.\tau[\alpha] = \mu\alpha.\tau[\mu\alpha.\tau[\alpha]]$$

■