# Computer Science 15-312
# Data Abstraction

Robert Harper

Spring, 2000

## 1 Introduction

Data abstraction is perhaps the most fundamental technique for structuring programs to ensure their robustness over time and to facilitate team development. The fundamental idea of data abstraction is the separation of the *client* from the *implementor* of the abstraction by an *interface*. The interface is a form of "contract" between the client and implementor. It specifies the operations that may be performed on values of the abstract type by the client and, at the same time, imposes the obligation on the implementor to provide these operations with the specified functionality. By limiting the client's view of the abstract type to a specified set of operations, the interface protects the client from depending on the details of the implementation of the abstraction, most especially its representation in terms of well-known constructs of the programming language. Doing so ensures that the implementor is free to change the representation (and, correspondingly, the implementation of the operations) of the abstract type without affecting the behavior of a client of the abstraction.

The purpose of this note is to develop a rigorous account of data abstraction in an extension of PolyMinML with *existential types*. Existential types provide the fundamental linguistic mechanisms for defining interfaces, implementing them, and using the implementation in client code. Using this extension of PolyMinML we will then develop a formal treatment of representation independence based on Reynolds's Parametricity Theorem for PolyMinML. The representation independence theorem will then serve as the basis for proving the correctness of abstract type implementations using bisimulation relations.

## 2 Existential Types

### 2.1 Abstract Syntax

The syntax of PolyMinML is extended with the following constructs:

$$
\begin{array}{llll}
\textit{Polytypes} & \sigma & ::= & \ldots \\
 & & \mid & \exists t(\sigma) \\
\textit{Expressions} & e & ::= & \ldots \\
 & & \mid & \texttt{pack}\,\tau\,\texttt{with}\,e\,\texttt{as}\,\sigma\,\texttt{end} \\
 & & \mid & \texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\,\texttt{:}\,\sigma\,\texttt{in}\,e_2\,\texttt{end} \\
\textit{Values} & v & ::= & \ldots \\
 & & \mid & \texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\sigma\,\texttt{end}
\end{array}
$$

The polytype $\exists t(\sigma)$ is called an *existential* type. An existential type is the *interface* of an abstract type. An *implementation* of the existential type $\exists t(\sigma)$ is a *package value* of the form $\texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}$ consisting of a monotype $\tau$ together with a value $v$ of type $[\tau/t]\sigma$. The monotype $\tau$ is the *representation type* of the implementation; the value $v$ is the implementation of the operations of the abstract type. A client makes use of an implementation by *opening* it within a scope, written $\texttt{open}\,e_i\,\texttt{as}\,t\,\texttt{with}\,x\,\texttt{:}\,\sigma\,\texttt{in}\,e_c\,\texttt{end}$, where $e_i$ is an implementation of the interface $\exists t(\sigma)$, and $e_c$ is the client code defined in terms of an unknown type $t$ (standing for the representation type) and an unknown value $x$ of type $\sigma$ (standing for the unknown operations).

In an existential type $\exists t(\sigma)$ the type variable $t$ is bound in $\sigma$, and may be renamed at will to satisfy uniqueness requirements. In an expression of the form $\texttt{open}\,e_i\,\texttt{as}\,t\,\texttt{with}\,x\,\texttt{:}\,\sigma\,\texttt{in}\,e_c\,\texttt{end}$ the type variable $t$ and the ordinary variable $x$ are bound in $e_c$, and may also be renamed at will to satisfy non-occurrence requirements. As we will see below, renaming of bound variables is crucial for ensuring that an abstract type is "new" in the sense of being distinct from any other type whenever it is opened for use in a scope. This is sometimes called *generativity* of abstract types, since each occurrence of `open` "generates" a "new" type for use within the body of the client. In reality this informal notion of generativity comes down to renaming of bound variables to ensure their uniqueness in a context.

### 2.2 Correspondence With ML

To fix ideas, it is worthwhile to draw analogies between the present formalism and (some aspects of) the Standard ML module system. We have the following correspondences:

| *PolyMinML + Existentials* | *Standard ML* |
|---|---|
| Existential type | Signature |
| Package | Structure, with opaque ascription |
| Opening a package | `open` declaration |

Here is an example of these correspondences in action. In the sequel we will use ML-like notation with the understanding that it is to be interpreted in PolyMinML in the following fashion.

Here is an ML signature for a persistent representation of queues:

```
signature QUEUE =
  sig
    type queue
    val empty : queue
    val insert : int * queue -> queue
    val remove : queue -> int * queue
  end
```

This signature is deliberately stripped down to simplify the development. In particular we leave undefined the meaning of `remove` on an empty queue.

The corresponding existential type is $\sigma_q := \exists q(\tau_q)$, where

$$\tau_q := q*((\texttt{int}*q)\to q)*(q\to(\texttt{int}*q))$$

That is, the operations of the abstraction consist of a three-tuple of values, one for the empty queue, one for the insert function, and one for the remove function.

Here is a straightforward implementation of the `QUEUE` interface in ML:

```
structure QL :> QUEUE =
  struct
    type queue = int list
    val empty = nil
    fun insert (x, xs) = x::xs
    fun remove xs =
        let val (x,xs') = rev xs in (x, rev xs') end
  end
```

A queue is a list in reverse enqueue order — the last element to be enqueued is at the head of the list. Notice that we use *opaque* signature ascription to ensure that the type `queue` is hidden from the client!

The corresponding package is $e_q := \texttt{pack int list with } v_q \texttt{ as } \sigma_q \texttt{ end}$, where

$$v_q := (\texttt{nil}, (v_i, v_r))$$

where $v_i$ and $v_r$ are the obvious function abstractions corresponding to the ML code given above.

Finally, a client of an abstraction in ML might typically open it within a scope:

```
local
  open QL
in
  ...
end
```

This corresponds to writing

$$\texttt{open QL as q with <n,i,r> : } \tau_q \texttt{ in ... end}$$

in the existential type formalism, renaming variables for convenience.

### 2.3 Static Semantics

The static semantics is an extension of that of PolyMinML with rules governing the new constructs. The rule of formation for existential types is as follows:

$$\frac{\Delta \cup \{\, t \,\} \vdash \sigma \ \mathsf{ok} \quad t \notin \Delta}{\Delta \vdash \exists t(\sigma) \ \mathsf{ok}}$$

The requirement $t \notin \Delta$ may always be met by renaming the bound variable.

The typing rule for packages is as follows:

$$\frac{\Delta \vdash \tau \ \mathsf{ok} \quad \Delta \vdash \exists t(\sigma) \ \mathsf{ok} \quad \Gamma \vdash_\Delta e : [\tau/t]\sigma}{\Gamma \vdash_\Delta \texttt{pack}\, \tau\, \texttt{with}\, e\, \texttt{as}\, \exists t(\sigma)\, \texttt{end}}$$

The implementation, $e$, of the operations "knows" the representation type, $\tau$, of the ADT.

The typing rule for opening a package is as follows:

$$\frac{\Delta \vdash \tau' \ \mathsf{ok} \quad \Gamma, x{:}\sigma \vdash_{\Delta \cup \{\, t \,\}} e_c : \tau_c \quad \Gamma \vdash_\Delta e_i : \exists t(\sigma) \quad t \notin \Delta}{\Gamma \vdash_\Delta \texttt{open}\, e_i\, \texttt{as}\, t\, \texttt{with}\, x{:}\sigma\, \texttt{in}\, e_c\, \texttt{end} : \tau_c}$$

This is a complex rule, so study it carefully! Two things to note:

1. The type of the client, $\tau_c$, must not involve the abstract type $t$. This prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.

2. The body of the client, $e_c$, is type checked without knowledge of the representation type, $t$. The client is, in effect, polymorphic in $t$.

As usual, the condition $t \notin \Delta$ can always be met by renaming the bound variable $t$ of the `open` expression to ensure that it is distinct from all other active types $\Delta$. It is in this sense that abstract types are "new"! Whenever a client opens a package, it introduces a local name for the representation type, which is bound within the body of the client. By our general conventions on bound variables, this local name may be chosen to ensure that it is distinct from any other such local name that may be in scope, which ensures that the "new" type is different from any other type currently in scope. At an informal level this ensures that the representation type is "held abstract"; we will make this intuition more precise in Section 3 below.

### 2.4 Dynamic Semantics

We will use structured operational semantics (SOS) to specify the dynamic semantics of existential types. Here is the rule for evaluating package expressions:

$$\frac{e \mapsto e'}{\texttt{pack}\,\tau\,\texttt{with}\,e\,\texttt{as}\,\sigma\,\texttt{end} \mapsto \texttt{pack}\,\tau\,\texttt{with}\,e'\,\texttt{as}\,\sigma\,\texttt{end}}$$

Opening a package begins by evaluating the package expressions:

$$\frac{e_i \mapsto e_i'}{\texttt{open}\,e_i\,\texttt{as}\,t\,\texttt{with}\,x\!:\!\sigma\,\texttt{in}\,e_c\,\texttt{end} \mapsto \texttt{open}\,e_i'\,\texttt{as}\,t\,\texttt{with}\,x\!:\!\sigma\,\texttt{in}\,e_c\,\texttt{end}}$$

Once the package is fully evaluated, we bind $t$ to the representation type and $x$ to the implementation of the operations within the client code:

$$\frac{}{\texttt{open}\,\texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\sigma\,\texttt{end}\,\texttt{as}\,t\,\texttt{with}\,x\!:\!\sigma\,\texttt{in}\,e_c\,\texttt{end} \mapsto [\tau, v/t, x]e_c}$$

Observe that *there are no abstract types at run time*! During execution of the client, the representation type is *fully exposed*. It is held abstract *only* during type checking to ensure that the client does not (accidentally or maliciously) depend on the implementation details of the abstraction. Once the program type checks there is no longer any need to enforce abstraction. The dynamic semantics reflects this intuition directly.

### 2.5 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for PolyMinML to the new constructs.

**Theorem 1 (Preservation)**
*If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

**Lemma 2 (Canonical Forms)**
*If $v : \exists t(\sigma)$ is a value, then $v = \texttt{pack}\,\tau\,\texttt{with}\,v'\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}$ for some monotype $\tau$ and some value $v' : [\tau/t]\sigma$.*

**Theorem 3 (Progress)**
*If $e : \tau$ then either $e$ value or there exists $e'$ such that $e \mapsto e'$.*

## 3 Representation Independence

Parametricity is the essence of representation independence. The typing rules for `open` given above ensure that the client of an abstract type is polymorphic in the representation type. According to our informal understanding of parametricity this means that the client's behavior is in some sense "independent" of the representation type.

More formally, we say that an (admissible) relation $R : \tau_1 \leftrightarrow \tau_2$ is a *bisimulation* between the packages

$$\texttt{pack}\,\tau_1\,\texttt{with}\,v_1\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}$$

and

$$\texttt{pack}\,\tau_2\,\texttt{with}\,v_2\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}$$

of type $\exists t(\sigma)$ iff $v_1 \cong v_2 : \sigma$, taking equality at type $t$ to be the relation $R$. The reason for calling such a relation $R$ a bisimulation will become apparent shortly. Two packages are said to be *bisimilar* whenever there is a bisimulation between them.

Since the client $e_c$ of a data abstraction of type $\exists t(\sigma)$ is essentially a polymorphic function of type $\forall t(\sigma \to \tau_c)$, where $t \notin \mathrm{FTV}(\tau_c)$, it follows from the Parametricity Theorem that

$$[\tau_1, v_1/t, x]e_c \cong [\tau_2, v_2/t, x]e_c : \tau_c$$

whenever $R$ is such a bisimulation. Consequently,

$$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\sigma\,\texttt{in}\,e_c\,\texttt{end} \cong \texttt{open}\,e_2\,\texttt{as}\,t\,\texttt{with}\,x{:}\sigma\,\texttt{in}\,e_c\,\texttt{end} : \tau_c.$$

That is, the two implementations are indistinguishable by any client of the abstraction, and hence may be regarded as equivalent. This is called *Representation Independence*; it is merely a restatement of the Parametricity Theorem in the context of existential types.

This observation licenses the following technique for proving the correctness of an ADT implementation. Suppose that we have an implementation of an abstract type $\exists t(\sigma)$ that is "clever" in some way. We wish to show that it is a correct implementation of the abstraction. Let us therefore call it a *candidate* implementation. The Representation Theorem suggests a technique for proving the candidate correct. First, we define a *reference* implementation of the same abstract type that is "obviously correct". Then we establish that the reference implementation and the candidate implementation are bisimilar. Consequently, they are equivalent, which is to say that the candidate is "equally correct as" the reference implementation.

Returning to the queues example, let us take as a reference implementation the package determined by representing queues as lists. As a candidate implementation we take the package corresponding to the following ML code:

```
structure QFB :> QUEUE =
  struct
    type queue = int list * int list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
  end
```

We will show that `QL` and `QFB` are bisimilar, and therefore indistinguishable by any client.

Define the relation $R : \texttt{int list} \leftrightarrow \texttt{int list*int list}$ as follows:

$$R = \{ (l, (b,f))) \mid l \cong b\texttt{@rev(}f\texttt{)} \}$$

We will show that $R$ is a bisimulation by showing that implementations of `empty`, `insert`, and `remove` determined by the structures `QL` and `QFB` are equivalent relative to $R$.

To do so, we will establish the following facts:

1. $\texttt{QL.empty}\, R\, \texttt{QFB.empty}$.

2. Assuming that $m \cong n : \texttt{int}$ and $l\, R\, (b,f)$, show that

$$\texttt{QL.insert((}m\texttt{,}l\texttt{))}\, R\, \texttt{QFB.insert((}n\texttt{,(}b\texttt{,}f\texttt{)))}.$$

3. Assuming that $l\,R\,$(b,f), show that

$$\texttt{QL.remove(}l\texttt{)} \cong \texttt{QFB.remove((}b\texttt{,}f\texttt{))} : \texttt{int*}t,$$

taking $t$ equality to be the relation $R$.

Observe that the latter two statements amount to the assertion that the operations *preserve* the relation $R$ — they map related input queues to related output queues. It is in this sense that we say that $R$ is a bisimulation, for we are showing that the operations from QL simulate, and are simulated by, the operations from QFB, up to the relationship $R$ between their representations.

The proofs of these facts are relatively straightforward, given some relatively obvious lemmas about expression equivalence.

1. To show that QL.empty $R$ QFB.empty, it suffices to show that

$$\texttt{nil@rev(nil)} \cong \texttt{nil} : \texttt{int list},$$

which is obvious from the definitions of append and reverse.

2. For insert, we assume that $m \cong n$ : int and $l\,R\,$(b,f), and prove that

$$\texttt{QL.insert(}m,l\texttt{)}\ R\ \texttt{QFB.insert(}n,\texttt{(}b\texttt{,}f\texttt{))}.$$

By the definition of QL.insert, the left-hand side is equivalent to $m$::$l$, and by the definition of QR.insert, the right-hand side is equivalent to ($n$::$b$,$f$). It suffices to show that

$$m\texttt{::}l \cong (n\texttt{::}b)\texttt{@rev(}f\texttt{)} : \texttt{int list}.$$

Calculating, we obtain

$$\begin{array}{rcl}(n\texttt{::}b)\texttt{@rev(}f\texttt{)} & \cong & n\texttt{::}(b\texttt{@rev(}f\texttt{)}) \\ & \cong & n\texttt{::}l\end{array}$$

since $l \cong b$@rev($f$). Since $m \cong n$ : int, it follows that $m = n$, which completes the proof.

3. For remove, we assume that $l$ is related by $R$ to ($b$,$f$), which is to say that $l \cong b$@rev($f$). We are to show

$$\texttt{QL.remove(}l\texttt{)} \cong \texttt{QFB.remove((}b\texttt{,}f\texttt{))} : \texttt{int*}t,$$

taking $t$ equality to be the relation $R$. Assuming that the queue is non-empty, so that the remove is defined, we have $l \cong l'$@[$m$] for some $l'$ and $m$. We proceed by cases according to whether or not $f$ is empty. If $f$ is non-empty, then $f \cong n\texttt{::}f'$ for some $n$ and $f'$. Then by the definition of `QFB.remove`,

$$\texttt{QFB.remove((}b\texttt{,}f\texttt{))} \cong \texttt{(}n\texttt{,(}b\texttt{,}f'\texttt{))} : \texttt{int*}t,$$

relative to $R$. We must show that

$$\texttt{(}m\texttt{,}l'\texttt{)} \cong \texttt{(}n\texttt{,(}b\texttt{,}f'\texttt{))} : \texttt{int*}t,$$

relative to $R$. This means that we must show that $m = n$ and $l' \cong b\texttt{@rev(}f'\texttt{)} : \texttt{int list}$.

Calculating from our assumptions,

$$\begin{array}{rcl} l & = & l'\texttt{@[}m\texttt{]} \\ & = & b\texttt{@rev(}f\texttt{)} \\ & = & b\texttt{@rev(}n\texttt{::}f'\texttt{)} \\ & = & b\texttt{@(rev(}f'\texttt{)@[}n\texttt{])} \\ & = & (b\texttt{@rev(}f'\texttt{))@[}n\texttt{]} \end{array}$$

From this the result follows. Finally, if $f$ is empty, then $b \cong b'\texttt{@[}n\texttt{]}$ for some $b'$ and $n$. But then $\texttt{rev(}b\texttt{)} \cong n\texttt{::rev(}b'\texttt{)}$, which reduces to the case for $f$ non-empty.

This completes the proof — by Representation Independence the reference and candidate implementations are equivalent.