CMPSCI 530/630: Programming Languages
Data Abstraction and Existential Types
Spring 2006
(adapted from Robert Harper’s Spring 2002 version)

Data Abstraction
Recall the central ideas of data abstraction:
• Define a representation type together with operations that manipulate values of that type.
• Hold the representation type abstract from clients of the ADT to ensure representation independence.

Data Abstraction and Polymorphism
The client is polymorphic in the representation typed
Therefore the behavior of the client is independent of the choice of representation.
This is called representation independence for abstract types.
Representation independence is ensured by polymorphic abstraction.

Existential Types
We’ll extend the syntax of PolyMinML with the following constructs:

Types \( \tau : = \exists (\tau) \)
Expressions \( e : = \text{pack } \tau \text{ with } x: \sigma \text{ end} \)
\( | \text{open } e_1 \text{ as } x: \sigma \text{ in } e_2 \text{ end} \)
Values \( v : = \text{pack } \tau \text{ with } x: \sigma \text{ end} \)

Note: open binds \( t \) within \( e_2 \! \).

Existential Types
The type \( \exists (\sigma) \) is an existential type. Its elements consist of
1. some type \( \tau \), together with
2. an implementation \( e \) of type \( \{ \tau/t \} \sigma \).
In practice \( \sigma \) is another existential, or a tuple or record of function types.

Existential Types
Binding and scope:
• \( t \) is bound in \( \sigma \) in \( \exists (\sigma) \).
• \( t \) and \( x \) are bound in \( e_2 \) in \( \text{open } e_1 \text{ as } x: \sigma \text{ in } e_2 \text{ end} \).

As usual, we implicitly rename bound variables to avoid clashes.
Existential Types

For example, consider the signature

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

This signature corresponds to the existential type

```
σq := 3y(τq)
```

where

```
τq := q*(int*τq→τq)
```

Similarly, the implementation

```
structure Queue :> QUEUE = struct
  type queue = int list
  val empty = ... list with ⟨ve, vi, vr⟩as σq end.
```

where ve, vi, and vr are the PolyMinML analogues of the ML functions given above.

Finally, the client code

```
local open Queue in e end
```

corresponds to the expression

```
open Queue as queue with ⟨empty, insert, remove⟩:τq in e end
```

This signature corresponds to the existential type

```
σq := 3y(τq)
```

where

```
τq := q*(int*τq→τq)
```

The bound type variable t can always be chosen not to occur in ∆ (by renaming the bound variable).
**Static Semantics**

The typing rule for opening a package is crucial:

\[ \Delta \vdash \tau_c \text{ type } \Gamma, x: \sigma \vdash e_i : \tau_e \quad \Gamma \vdash \Delta \vdash \exists(\sigma) \quad t \notin \Delta \]

\[ \Gamma \vdash \Delta \vdash \text{open } e_i \text{ as } \{ x : \sigma \} \text{ in } e_c \vdash \tau_e \]

That is,

- \( e_i \) must be a package of type \( \exists(\sigma) \).
- \( e_i \) must be of type \( \tau_e \) while holding \( t \) abstract.
- \( \tau_c \) must not involve \( t \).

**Static Semantics**

Observe that the client, \( e_c \), is

- **polymorphic** in the representation type \( t \) of the ADT, and
- **abstracted** on the implementation of its operations.

Linking consists of simplifying the open expression after type checking to ensure that the client is polymorphic.

**Dynamic Semantics**

The SOS rules for open expressions are as follows.

First, we evaluate the package:

\[ e_i \mapsto e_i' \]

\[ \text{open } e_i \text{ as } \{ x : \sigma \} \text{ in } e_c \vdash \text{open } e_i' \text{ as } \{ x : \sigma \} \text{ in } e_c \]

Then we run the body with the “opened” package:

\[ \text{open pack } \tau \text{ with } v \text{ as end as } \{ x : \sigma \} \text{ in } e_c \vdash \{ \tau, v / t, x \} \tau_e \]

**No ADT’s At Run Time!**

**Important:** there are no abstract types at run time!

- Type checking rule for clients holds representation type abstract.
- Dynamic semantics of open replaces the abstract type by its representation before executing client.
- Therefore at run time abstraction is lost; that is, abstraction is a compile-time notion!

Corollary: data abstraction does not introduce run-time overhead!
Bisimilarity

Informally, a bisimulation between two implementations of an ADT consists of

1. A “fictional” notion of equality between their representations.

2. A proof that the operations of the ADT preserve this “fiction”.

The operations preserve this equality if they yield equivalent results given equivalent arguments.

Safety

Safety is stated and proved as usual.

Theorem 1 (Preservation)
If \( e : \tau \) and \( e \rightarrow e' \), then \( e' : \tau \).

Lemma 2 (Canonical Forms)
If \( v : \exists \tau \) is a value, then \( v = \text{pack } \tau \) with \( \tau' \) as \( \exists \tau \) end for some type \( \tau \) and some value \( \tau' : (\tau/\ell) \).

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

Encoding Existentials

Existential types may be encoded in terms of universal types.

- \( \exists \tau : = \forall u ((\forall (\tau \rightarrow u) \rightarrow u) \), where \( u \notin \text{FTV}(\sigma) \).
- \( \text{pack } \tau \text{ with } v \text{ as } \exists \tau \text{ end } : = \Lambda u . \lambda x : \forall (\tau \rightarrow u) . x [\tau](v) \).
- \( \text{open as } \text{with } [\tau] \text{ in } e \text{ end } : = e [\tau'] (\Lambda t . \lambda x : \sigma . e') \), where \( \tau' \) is the type of \( e' \).

Reasoning About ADT's

A useful technique for reasoning about ADT's:

- Define a reference implementation that is “obviously” correct.
- Define a candidate implementation that is “clever” in some way.
- Define a bisimulation between the reference and candidate.
Reasoning About ADT’s

By the Parametricity Theorem,

- The reference and candidate implementations are indistinguishable.
- This may be interpreted as saying that the candidate is correct.

Example: queues two ways.

- As a list of elements, with the head of the list being the most recently enqueued value, and its last element as the next to be dequeued.
- As a pair of lists, the “back” and the “front”, with the most recently enqueued value at the head of the back list, and the next value to be dequeued at the head of the front list.

Reasoning About ADT’s

A reference implementation:

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

A candidate implementation:

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

The bisimulation relation $R : \text{int list} \rightarrow \text{int list \times int list}$ is defined as follows:

$$R = \{(l, (b,f)) : l \leq b \circ \text{rev}(f)\}$$

We must show that the operations preserve $R$. First, the implementation of empty:

Clearly

$\text{nil} = (\text{nil, nil}) : \text{int list}$

since

$\text{nil} \circ \text{rev}(\text{nil}) = \text{nil} : \text{int list}$
Reasoning About ADT’s

Next, the insert operation.

Suppose that
\[ m = n : \text{int} \]
and
\[ l R (b, f). \]

That is, \( m = n \) and \( l = b \& \text{rev}(f) \).

Reasoning About ADT’s

Finally, we consider remove.

Assume that \( l \) is related by \( R \) to \( (b, f) \). That is, \( l \) is equivalent to \( b \& \text{rev}(f) \).

Let \( \text{QL.remove}(l) \) be \( (m, l') \), and let \( \text{QFB.remove}(b, f) \) be \( (n, (b', f')) \).

We are to show that \( m = n \) and \( l' \) is equivalent to \( b' \& \text{rev}(f') \).

Reasoning About ADT’s

We are to show
\[ \text{QL.insert}(m, l) R \text{QFB.insert}(n, (b, f)) \]

The left-hand side is equivalent to \( m : l \); the right-hand side is equivalent to \( n : f \).

Note that \( n : b \& \text{rev}(f) \) is equivalent to \( n : (b \& \text{rev}(f)) \).

But then \( m : l \) is related to \( n : f \) by \( R \), as required.

Reasoning About ADT’s

Calculating from our assumptions,\[ l = l' @ [m], \]
\[ b \& \text{rev}(f) = b \& \text{rev}(n : f'), \]
\[ b \& \text{rev}(f') @ [n] = (b \& \text{rev}(f')) @ [n]. \]

Since \( l' @ [m] = (b \& \text{rev}(f')) @ [n] : \text{int list} \), it follows that \( m = n \)
and \( l' = b \& \text{rev}(f') : \text{int list} \).
Reasoning About ADT’s

We made use of several lemmas along the way.

- Associativity of append.
- Reversal of appending two lists is the append of their reversals.
- Symbolic evaluation is a valid form of equivalence.

We also relied on the Parametricity Theorem, a deep result about polymorphism.

Summary

Existential types capture the informal concept of data abstraction.

Bisimilarity is a method of reasoning about ADT’s.

- Exhibit a correspondence between representations.
- Show that the operations of the ADT preserve it.
- Apply parametricity.

Parametricity Theorem

Informally, Reynolds’ Parametricity Theorem states that polymorphic expressions preserve all relations on the quantified type.

More precisely,

- if \( \epsilon_1, \epsilon_2 : \forall \tau \) and
- if \( \sigma_1 \) and \( \sigma_2 \) are any types, then
- for any relation \( R \) between \( \sigma_1 \) and \( \sigma_2 \).
- \( \epsilon_1 \ [\sigma_1] \) is “equivalent” to \( \epsilon_2 \ [\sigma_2] \), relative to \( R \).