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 type.

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.

Existing Types

We’ll extend the syntax for $L_{\text{Polymorphism}}$ (previous lecture) with the following constructs to produce $L_{\text{Existential Types}}$:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\exists t.\tau$</td>
<td>$\exists t.\tau$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{pack}_{\rho}[\tau] e$</td>
<td>$\text{open}_{e_1} \tau_1$ as $x : \tau_2$</td>
</tr>
</tbody>
</table>

**Note**: open binds $t$ and $x$ within $e_1$!

Introductory form is a **package** where $\rho$ is its **representation type** and $e : [\rho/t] \tau$ is its implementation.

Elimination form opens $e_1$ for use within client $e_2$ by binding its representation type to $t$ and its implementation to $x$.

Existential Types

<table>
<thead>
<tr>
<th>Binding and scope:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$ is bound in $\tau$ in $\exists t.\tau$.</td>
</tr>
<tr>
<td>$t$ and $x$ are bound in $e_2$ in $\text{open}_{e_1} \tau_1$ as $x : \tau_2$.</td>
</tr>
</tbody>
</table>

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

Existential Types

The type $\exists t.\tau$ is an **existential type**. Its elements are packages of the form $\text{pack}_{\rho}[\tau] e$ that consist of:

1. Some type $\rho$, together with
2. An implementation $e$ of type $[\rho/t] \tau$.

In practice $\tau$ is another existential, or a tuple or record of function types.
Existential Types

For example, consider the classic abstract data type integer queue, typically defined by three operations:

1. Creation of an empty queue
2. Insertion of an integer at the end of the queue
3. Removal of an integer from the head of the queue

Existential Types

In SML, the interface to the integer queue abstract data type might be given by the signature

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

Existential Types

This signature corresponds to the existential type

\[ \sigma_q = \exists (q : \tau_q) \]

where
\[ \tau_q = (\text{emp} : q, \text{ins} : \text{int} \times q \rightarrow q, \text{rem} : q \rightarrow \text{int} \times q) \]

or
\[ \sigma_q = \exists (q : \text{emp} : q, \text{ins} : \text{int} \times q \rightarrow q, \text{rem} : q \rightarrow \text{int} \times q) \]

The representation type, \( q \), is abstract — all that is specified about it is that it supports the indicated operations with the indicated types.

Existential Types

An SML implementation of the integer queue abstract data type might be given by the following:

```
structure 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
```

Existential Types

Finally, the client code

```
local open Queue in
end
```

corresponds to the expression

```
open \sigma_q as \tau_q with (\text{emp}, \text{ins}, \text{rem}) \in e
```

A Different Implementation

An alternative, and possibly more efficient, implementation of the integer queue abstract data type might be given by the following SML code:

```
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
```

Existential Types

This implementation corresponds to the package \(vfb\):

\[\text{pack list list int list with } (\text{emp } = (\text{nil}, \text{nil}), \text{ins } = e_1, \text{rem } = e_2) \text{ as } \sigma_0.\]

where:

\[e_1 : \text{int } \to (\text{int list } \times \text{int list}) \to (\text{int list } \times \text{int list})\]

\[e_1 = \lambda(x: \text{int } \times (\text{int list } \times \text{int list}), e_2')\]

\[e_2 : (\text{int list } \times \text{int list}) \to \text{int } \times (\text{int list } \times \text{int list})\]

\[e_2 = \lambda(x: \text{int list } \times \text{int list}, e_2')\]

and \(e_1'\) and \(e_2'\) are analogues of the ML functions given above.

Existential Types

Again, the client code

```
local open QFB in e end
```

corresponds to the expression

```
open vfb as with (emp, ins, rem) : \sigma_0 in e
```

Static Semantics

Well-formed packages obey this typing rule:

\[
\Delta, \Gamma \vdash e_1 : \text{some}(\sigma) \\
\Delta, \Gamma \vdash e_2 : e_2' \\
\Delta \vdash \text{open}(\sigma) \Gamma \Gamma_1 \Gamma_2 : \text{some}(\tau)
\]

The bound type variable \(t\) can always be chosen not to occur in \(\Delta\) (by renaming the bound variable).

Static Semantics

The typing rule for opening a package is crucial:

\[
\Delta, \Gamma \vdash e_1 : \text{some}(\tau), \Delta, \Gamma \vdash e_2 : \tau_2 \\
\Delta \vdash \text{open}(\tau) \Gamma \Gamma_1 \Gamma_2 : \tau_2
\]

That is,

- \(e_1\) must be a package of type \(\text{some}(\tau)\).
- \(e_2\) must be of type \(\tau_2\) while holding \(t\) abstract.
- \(\tau_2\) must not involve \(t\).
Static Semantics

The restriction that $\Delta \vdash \tau$ type and $t \notin \Delta$ precludes "exporting" values of the underlying type.

For example, if $e_1 : \exists (q : q)$, then this is illegal:

$\text{open } e_1 \text{ as with } \lambda q : q. \text{int } q \Rightarrow \text{int } q^r : \tau \text{ in } n$

This expression cannot be typed due to the restriction!

The client must compute something extrinsic to the ADT (e.g., some integer value).

Properties of Typing

Lemma 1 (Regularity)
Suppose that $\Delta \vdash e : \tau$. If $\Delta \vdash \tau_i$ type for each $x : \tau_i$ in $\Gamma$, then $\Delta \vdash \tau$ type

Proof: The proof is by induction on typing. □

Dynamic Semantics

The structural semantics rules for package expressions are as follows:

$$\text{pack } [t.\tau] p \, (e) \text{ val}$$

$$\begin{cases} \vdash e \text{ val} \\ \text{pack } [t.\tau] p \, (e') \text{ val} \end{cases}$$

Bracketed rule and premise omitted for lazy semantics and included for eager semantics.

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 $\text{open}$ 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!
Safety

Safety is stated and proved as usual.

Theorem 2 (Preservation)
If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Lemma 3 (Canonical Form)
If $e : \text{some } [t.] \tau$ and $e \text{ val}$, then $e \mapsto \text{pack } [t.] \tau$ for some $\rho$ and some $e' \text{ val}$ such that $e' : \rho [t.] \tau$.

Theorem 4 (Progress)
If $e : \tau$ then either $e \text{ val}$ or there exists $e'$ such that $e \mapsto e'$.

Definability of Existentials

Existential types may be encoded in terms of universal types.

- $\exists [t.] \sigma = \forall [t.] \sigma (t. \rightarrow e')$ for some type $\rho$ and some $e' \text{ val}$ such that $e' : \rho [t.] \tau$.

- pack $\rho$ with $e$ as $\exists [t.] \sigma$.

- open $e$ with $z : \sigma$ in $e' = e' [(\tau') (\Lambda (t. \lambda (x : \sigma) e'))]$, where $\tau'$ is the type of $e'$.

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.

Bisimilarity and Parametricity

If there is a bisimulation between two packages, they are said to be bisimilar.

Reynolds’ Parametricity Theorem states that if two packages are bisimilar, then no client can distinguish between them.

- Client’s behavior does not change if one is replaced by the other.

- A consequence of the polymorphic typing of the client.

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, per our earlier examples.

- 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

The reference implementation:

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 txx rev l' end
  end

Reasoning About ADT's

The candidate implementation:

structure QFB :> QUEUE =
  struct
  type queue = int list × int list
  val empty = tnilx nilu
  fun insert (tx::bx fuu) = tx::bx fuu
  fun remove (tx::bx fuu) = remove tnilx rev bu
    | remove tnilx (x::fu) = txx tx bx fuu
  end

Reasoning About ADT's

The bisimulation relation \( R : \text{int list} \leftrightarrow \text{int list} \times \text{int list} \) is defined as follows:

\[
R = \{(l, (b, f)) \mid l = \text{ldrev}(f)\}
\]

We must show that the operations preserve \( R \).

First, the implementation of empty:

Clearly

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

since

\[
\text{nil} \text{ldrev}(\text{nil}) = \text{nil} : \text{int list}
\]
Reasoning About ADT's

Next, the insert operation.

Suppose that 

\[ m = n : \text{int} \]

and

\[ l R (b, f). \]

That is, \( m = n \) and \( l = \mathcal{B} \mathcal{R} \mathcal{W}(f) \).

Reasoning About ADT's

We are to show

\[ \mathcal{Q}L.\text{insert}(m, l) \mathcal{R} \mathcal{Q}F.\text{insert}(n, (b, f)) \]

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

Note that \( (n : b) \mathcal{B} \mathcal{R} \mathcal{W}(f) \) is equivalent to \( n : (\mathcal{B} \mathcal{R} \mathcal{W}(f)) \).

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

Reasoning About ADT's

Finally, we consider remove.

Assume that \( l \) is related by \( R \) to \( (b, f) \). That is, \( l \) is equivalent to \( \mathcal{B} \mathcal{R} \mathcal{W}(f) \).

Let \( \mathcal{Q}L.\text{remove}(l) \) be \( (m, l') \), and let \( \mathcal{Q}F.\text{remove}(b, f) \) be \( (n, (b', f')) \).

We are to show that \( m = n \) and \( l' \) is equivalent to \( \mathcal{B} \mathcal{R} \mathcal{W}(f') \).

Reasoning About ADT's

Calculating from our assumptions,

\[
\begin{align*}
 l &= \mathcal{F}[m] \\
 &= \mathcal{B} \mathcal{R} \mathcal{W}(f) \\
 &= \mathcal{B} \mathcal{R} \mathcal{W}(n : f') \\
 &= \mathcal{B} \mathcal{R} \mathcal{W}(f') \mathcal{B}[n] \\
 &= (\mathcal{B} \mathcal{R} \mathcal{W}(f')) \mathcal{B}[n]
\end{align*}
\]

Since \( \mathcal{F}[m] = (\mathcal{B} \mathcal{R} \mathcal{W}(f')) \mathcal{B}[n] : \text{intlist} \), it follows that \( m = n \) and \( l' = \mathcal{B} \mathcal{R} \mathcal{W}(f') : \text{intlist} \).
**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.

**Parametricity Theorem**

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

More precisely,

- if \( e_1, e_2 : \forall \tau . \tau \) and
- if \( \sigma_1 \) and \( \sigma_2 \) are any types, then
- for any relation \( R \) between \( \sigma_1 \) and \( \sigma_2 \),
- \( e_1 [\sigma_1] \) is “equivalent” to \( e_2 [\sigma_2] \), relative to \( R \).

**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.