510: Programming Languages

Data Abstraction and Existential Types

David Walker

Data Abstraction

Recall the central ideas of data abstraction:

- Define a **representation type** together with **operations** that manipulate values of that type.

- Hold the type 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.

Existential Types

We'll extend the syntax of PolyMinML with the following constructs:

$$
\begin{array}{llcl}
\textit{Types} & \tau & ::= & \exists t(\tau) \\
\textit{Expressions} & e & ::= & \texttt{pack}\,\tau\,\texttt{with}\,e\,\texttt{as}\,\tau'\,\texttt{end} \\
& & \mid & \texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\tau\,\texttt{in}\,e_2\,\texttt{end} \\
\textit{Values} & v & ::= & \texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\tau'\,\texttt{end}
\end{array}
$$

Note: `open` binds t within e_2!

Existential Types

Binding and scope:

- t is bound in τ in $\exists t(\tau)$.

- t and x are bound in e_2 in

$$\texttt{open}\, e_1 \,\texttt{as}\, t \,\texttt{with}\, x\texttt{:}\tau \,\texttt{in}\, e_2 \,\texttt{end}$$

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

Existential Types

The type $\exists t(\tau)$ is an **existential** type. Its elements consist of

1. some type τ', together with

2. an implementation e of type $\{\tau'/t\}\tau$.

In practice τ is another existential, or a tuple or record of function types.

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
```

Existential Types

This signature corresponds to the existential type

$$\sigma_q := \exists q(\tau_q)$$

where

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

(I will use "σ" to range over types as well as "τ")

Existential Types

Similarly, the implementation

```
structure Queue :> 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
```

corresponds to the package

$$v_q : \sigma_q := \mathbf{pack}\,\mathtt{int\,list}\,\mathbf{with}\,\langle v_e, v_i, v_r\rangle\,\mathbf{as}\,\sigma_q\,\mathbf{end}.$$

where v_e, v_n, and v_r are the PolyMinML analogues of the ML functions given above.

Existential Types

Finally, the client code

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

corresponds to the expression

open Queue as queue with τ_q in
  e
end

Static Semantics

Well-formedness of existential types is defined by this rule:

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

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

Static Semantics

Well-formed packages obey this typing rule:

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

The implementation, e, of the operations "knows" the representation type, τ, of the ADT.

Static Semantics

The typing rule for opening a package is crucial:

$$\frac{\Gamma \vdash_{\Delta} e_i : \exists t(\sigma) \qquad t \notin \Delta \qquad \Gamma, x{:}\sigma \vdash_{\Delta \cup \{t\}} e_c : \tau_c \qquad \Delta \vdash \tau_c \ \mathsf{ok}}{\Gamma \vdash_{\Delta} \mathtt{open}\, e_i \,\mathtt{as}\, t \,\mathtt{with}\, x\mathtt{:}\sigma \,\mathtt{in}\, e_c \,\mathtt{end} : \tau_c}$$

That is,

- e_i must be a package of type $\exists t(\sigma)$.
- e_c must be of type τ_c **while holding t abstract**.
- τ_c must not involve t.

Static Semantics

The restriction that $\Delta \vdash \tau_c$ ok is crucial. It implies that $t \notin \mathsf{FTV}(\tau_c)$. If we did not have this condition, the result type might have a free type variable that did not appear in Δ, and such a type would not have any meaning.

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

```
open e_i as q, <n:q,i:int*q->q,r:q->int*q> in
  n
end
```

The client must compute something defined outside the ADT (*e.g.*, some integer value).

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 package expressions are as follows:

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

An expression of the form $\mathtt{pack}\,\tau\,\mathtt{with}\,v\,\mathtt{as}\,\sigma\,\mathtt{end}$ is a value, provided that v is a value.

Dynamic Semantics

The SOS rules for `open` expressions are as follows.

First, we evaluate the package:

$$\frac{e_i \mapsto e'_i}{\begin{array}{c}\texttt{open}\, e_i \,\texttt{as}\, t \,\texttt{with}\, x\,\texttt{:}\,\sigma \,\texttt{in}\, e_c \,\texttt{end} \\ \mapsto \\ \texttt{open}\, e'_i \,\texttt{as}\, t \,\texttt{with}\, x\,\texttt{:}\,\sigma \,\texttt{in}\, e_c \,\texttt{end}\end{array}}$$

Then we run the body with the "opened" package:

$$\frac{}{\begin{array}{c}\texttt{open}\,\texttt{pack}\, \tau \,\texttt{with}\, v \,\texttt{as}\, \sigma \,\texttt{end}\,\texttt{as}\, t \,\texttt{with}\, x\,\texttt{:}\,\sigma \,\texttt{in}\, e_c \,\texttt{end} \\ \mapsto \\ \{\tau, v/t, x\}e_c\end{array}}$$

No ADT's At Run Time!

Important: there are not 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!

Safety

Safety is stated and proved as usual.

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

$$v = \mathtt{pack}\,\tau\,\mathtt{with}\,v'\,\mathtt{as}\,\exists t(\sigma)\,\mathtt{end}$$

for some type τ and some value $v' : \{\tau/t\}\sigma$.

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

Encoding Existentials

Existential types may be encoded in terms of universal types.

$$\exists t(\sigma) :=$$

$$\forall u((\forall t(\sigma \to u)) \to u)$$

where $u \notin \mathsf{FTV}(\sigma)$

$$\mathtt{pack}\,\tau\,\mathtt{with}\,v\,\mathtt{as}\,\exists t(\sigma)\,\mathtt{end} :=$$

$$\Lambda\, u.\lambda x{:}\forall t(\sigma \to u).x\ [\tau]\,(v)$$

$$\mathtt{open}\,e\,\mathtt{as}\,t\,\mathtt{with}\,x{:}\sigma\ \mathtt{in}\,e'\,\mathtt{end} :=$$

$$e\ [\tau']\,(\Lambda\, t.\lambda x{:}\sigma.e')$$

where τ' 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

A **bisimulation** between two packages

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

and

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

consists of

1. A binary relation $R : \tau_1 \leftrightarrow \tau_2$ . . .

2. such that $v_1 = v_2 : \sigma$ "relative to" R.

What does "equal relative to R" mean?

$v_1 = v_2 : \sigma$ is an equality relation indexed by σ (which has 1 free type variable, α)

- $v_1 = v_2 : \alpha$ iff $v_1 \; R \; v_2$
- $m = n : \texttt{int}$ iff m is the same integer as n
- $(v_a, v_b) = (v'_a, v'_b) : \tau_1 * \tau_2$ iff
  $v_a = v'_a : \tau_1$ and $v_b = v'_b : \tau_2$
- $v_h :: v_t = v'_h :: v'_t : \tau \; \texttt{list}$ iff
  $v_h = v'_h : \tau$ and $v_t = v'_t : \tau \; \texttt{list}$
- $f = f' : \tau_1 \to \tau_2$ iff for all v and v',
  - If $v = v' : \tau_1$ and $f v \mapsto^* v_r$ and $f' v' \mapsto^* v'_r$
  - Then $v_r = v'_r : \tau_2$

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.

Reasoning About ADT's

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:

```
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
```

Reasoning About ADT's

A **candidate** implementation:

```
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
```

Reasoning About ADT's

The **bisimulation** relation

$$R : \texttt{int list} \leftrightarrow \texttt{int list*int list}$$

is defined as follows:

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

We must show that the operations preserve R.

Reasoning About ADT's

First, the implementation of `empty`:

Clearly

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

since

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

Reasoning About ADT's

Next, the `insert` operation.

Suppose that

$$m = n : \texttt{int}$$

and

$$l \, R \, \texttt{(}b\texttt{,}f\texttt{)}.$$

That is, $m = n$ and $l = b\texttt{@rev(}f\texttt{)}$.

Reasoning About ADT's

We are to show

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

The left-hand side is equivalent to $m\texttt{::}l$.

The right-hand side is equivalent to $\texttt{(}n\texttt{::}b\texttt{,}f\texttt{)}$.

Must prove: $m\texttt{::}l = (n\texttt{::}b)\texttt{@rev(}f\texttt{)}$

Must prove: $m\texttt{::}l = n\texttt{::}(b\texttt{@rev(}f\texttt{)})$

Obvious since $m = n$ and $l = (b\texttt{@rev(}f\texttt{)})$

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@rev(f).

Let QL.remove(l) be (m,l').

Let QFB.remove$((b,f))$ be $(n,(b',f'))$.

We are to show that $m = n$ and l' is equivalent to b'@rev(f').

Reasoning About ADT's

Assuming the queue to be non-empty, we may write l as l'`@[`m`]`.

Then `QL.remove(`l`)` is equivalent to `(`m`,`l'`)`.

There are two cases for `(`b`,`f`)`, according to whether or not f is **`nil`**.

If $f =$ **`nil`**, then $b = b'$`@[`n`]` for some b' and n.

But then **`rev`**`(`b`)` is equivalent to n`::`**`rev`**`(`b'`)`, which reduces to the non-empty case.

Reasoning About ADT's

Otherwise, $f = n\texttt{::}f'$ for some n and f'.

Then $\texttt{QFB.remove((}b\texttt{,}f\texttt{))}$ is equivalent to $\texttt{(}n\texttt{,(}b\texttt{,}f'\texttt{))}$.

We must show that:

- $m = n$.
- $l' = b\texttt{@rev(}f'\texttt{)}$.

Reasoning About ADT's

Calculating from our assumptions,

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

Since $l'\texttt{@[}m\texttt{]} = (b\texttt{@rev(}f'\texttt{)})\texttt{@[}n\texttt{]}$: `int list`, it follows that $m = n$ and $l' = b\texttt{@rev(}f'\texttt{)}$: `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.

Parametricity Theorem

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

More precisely, we say $v_1 = v_2 : \forall t(\tau)$

- if σ_1 and σ_2 are any (closed) types, then

- for any relation R between σ_1 and σ_2,

- $v_1\,[\sigma_1]$ is "equivalent" to $v_2\,[\sigma_2]$, 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.