

CMPSCI 630: Programming Languages  
**Data Abstraction and Existential Types**

Spring 2009  
 (with thanks to Robert Harper)

**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.

1

**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.

2

**Existential Types**

We'll extend the syntax for  $\mathcal{L}\{\rightarrow \forall\}$  (previous lecture) with the following constructs to produce  $\mathcal{L}\{\rightarrow \forall \exists\}$ :

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{some}(t.\tau)$	$\exists(t.\tau)$
Expr	$e$	$::= \text{pack}[t.\tau][\rho](e)$ $  \text{open}[t.\tau][\rho](e_1; t, x.e_2)$	$\text{pack } \rho \text{ with } e \text{ as } \exists(t.\tau)$ $\text{open } e_1 \text{ as } t \text{ with } x : \tau \text{ in } e_2$

**Note:** open binds  $t$  and  $x$  within  $e_2$ !

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

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

3

**Existential Types**

Binding and scope:

- $t$  is bound in  $\tau$  in  $\exists(t.\tau)$ .
- $t$  and  $x$  are bound in  $e_2$  in  $\text{open } e_1 \text{ as } t \text{ with } x : \tau \text{ in } e_2$ .

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

4

**Existential Types**

The type  $\exists(t.\tau)$  is an **existential type**. Its elements are packages of the form  $\text{pack } \rho \text{ with } e \text{ as } \exists(t.\tau)$  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.

5

### 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

6

### 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 -> queue
  val remove : queue -> int * queue
end
```

7

### Existential Types

This signature corresponds to the existential type

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

where

$$\tau_q = \langle \text{emp} : q, \text{ins} : \text{int} \times q \rightarrow q, \text{rem} : q \rightarrow \text{int} \times q \rangle$$

or

$$\sigma_q = \exists(q.\langle \text{emp} : q, \text{ins} : \text{int} \times q \rightarrow q, \text{rem} : q \rightarrow \text{int} \times q \rangle)$$

The representation type,  $q$ , is **abstract** – all that is specified about it is that it supports the indicated operations with the indicated types.

8

### Existential Types

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

```
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
```

9

### Existential Types

This implementation corresponds to the package

$$v_q : \sigma_q = \text{pack int list with } \langle \text{emp} = \text{nil}, \text{ins} = e_i, \text{rem} = e_r \rangle \text{ as } \sigma_q.$$

where:

$$e_i : \text{int} \times \text{int list} \rightarrow \text{int list} = \lambda(x:\text{int} \times \text{int list}.e'_i)$$

$$e_r : \text{int list} \rightarrow \text{int} \times \text{int list} = \lambda(x:\text{int list}.e'_r)$$

and  $e'_i$  and  $e'_r$  are analogues of the ML functions given above.

10

### Existential Types

Finally, the client code

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

corresponds to the expression

```
open v_q as q with (emp, ins, rem) : \tau_q in e
```

11

## 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
```

12

## Existential Types

This implementation corresponds to the package  $v_{fb}$ :

`pack int list × int list with (emp = ⟨nil, nil⟩, ins =  $e_i$ , rem =  $e_r$ ) as  $\sigma_q$ .`  
where:

$$e_i : \text{int} \times (\text{int list} \times \text{int list}) \rightarrow (\text{int list} \times \text{int list})$$

$$e_i = \lambda(x:\text{int} \times (\text{int list} \times \text{int list}).e'_i)$$

$$e_r : (\text{int list} \times \text{int list}) \rightarrow \text{int} \times (\text{int list} \times \text{int list})$$

$$e_r = \lambda(x:\text{int list} \times \text{int list}.e'_r)$$

and  $e'_i$  and  $e'_r$  are analogues of the ML functions given above.

13

## Existential Types

Again, the client code

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

corresponds to the expression

```
open  $v_{fb}$  as  $q$  with (emp, ins, rem) :  $\tau_q$  in e
```

14

## Static Semantics

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

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \text{some}(t.\tau) \text{ type}}$$

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

15

## Static Semantics

Well-formed packages obey this typing rule:

$$\frac{\Delta \vdash \rho \text{ type} \quad \Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \Gamma \vdash e : [\rho/t]\tau}{\Delta \Gamma \vdash \text{pack } [t.\tau][\rho] (e) : \text{some}(t.\tau)}$$

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

16

## Static Semantics

The typing rule for opening a package is crucial:

$$\frac{\Delta \Gamma \vdash e_1 : \text{some}(t.\tau) \quad \Delta, t \text{ type} \Gamma, x:\tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \Gamma \vdash \text{open } [t.\tau][\tau_2] (e_1; t, x.e_2) : \tau_2}$$

That is,

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

17

### Static Semantics

The restriction that  $\Delta \vdash \tau_2$  type and  $t \notin \Delta$  precludes “exporting” values of the underlying type.

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

`open e1 as q with <n:q,i:int*q->q,r:q->int*q> :  $\tau_q$  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).

18

### Static Semantics

Observe that the client,  $e_2$ , 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.

19

### Properties of Typing

#### Lemma 1 (Regularity)

Suppose that  $\Delta \Gamma \vdash e : \tau$ . If  $\Delta \vdash \tau_i$  type for each  $x_i : \tau_i$  in  $\Gamma$ , then  $\Delta \vdash \tau$  type

**Proof:** The proof is by **induction on typing**. ■

20

### Dynamic Semantics

The structural semantics rules for package expressions are as follows:

$$\frac{\{e \text{ val}\}}{\text{pack } [t.\tau][\rho](e) \text{ val}}$$

$$\left\{ \frac{e \mapsto e'}{\text{pack } [t.\tau][\rho](e) \mapsto \text{pack } [t.\tau][\rho](e')} \right\}$$

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

21

### Dynamic Semantics

The structural semantics rules for `open` expressions are as follows.

First, we evaluate the package:

$$\frac{e_1 \mapsto e'_1}{\text{open } [t.\tau][\tau_2](e_1; t, x.e_2) \mapsto \text{open } [t.\tau][\tau_2](e'_1; t, x.e_2)}$$

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

$$\frac{\{e \text{ val}\}}{\text{open } [t.\tau][\tau_2](\text{pack } [t.\tau][\rho](e); t, x.e_2) \mapsto [\rho, e/t, x]e_2}$$

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

22

### 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!

23

## Safety

Safety is stated and proved as usual.

### Theorem 2 (Preservation)

If  $e : \tau$  and  $e \mapsto e'$ , then  $e' : \tau$ .

### Lemma 3 (Canonical Forms)

If  $e : \text{some}(t.\tau)$  and  $e \text{ val}$ , then  $e = \text{pack } [t.\tau][\rho](e')$  for some type  $\rho$  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  $e \mapsto e'$ .

24

## Definability of Existentials

Existential types may be encoded in terms of universal types.

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

25

## 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.

26

## Bisimilarity

A **bisimulation** between two packages

$\text{pack } \tau_1 \text{ with } v_1 \text{ as } \exists(t.\sigma)$

and

$\text{pack } \tau_2 \text{ with } v_2 \text{ as } \exists(t.\sigma)$

consists of

1. A binary relation  $R : \tau_1 \leftrightarrow \tau_2 \dots$
2. such that  $v_1 = v_2 : \sigma$  “relative to”  $R$ .

27

## 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.

28

## 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.

29

### 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.

30

### Reasoning About ADT's

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.

31

### 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 (x, rev l') end  
  end
```

32

### Reasoning About ADT's

The **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
```

33

### Reasoning About ADT's

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

$$R = \{ (l, \langle b, f \rangle) \mid l = b @ \text{rev}(f) \}$$

We must show that the operations preserve  $R$ .

34

### Reasoning About ADT's

First, the implementation of `empty`:

Clearly

```
nil = (nil, nil) : int list
```

since

```
nil @ rev(nil) = nil : int list
```

35

### Reasoning About ADT's

Next, the insert operation.

Suppose that

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

and

$$l R \langle b, f \rangle.$$

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

36

### Reasoning About ADT's

We are to show

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

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

Note that  $\langle n :: b @ \text{rev}(f) \rangle$  is equivalent to  $n :: \langle b @ \text{rev}(f) \rangle$ .

But then  $m :: l$  is related to  $\langle n :: b, f \rangle$  by  $R$ , as required.

37

### Reasoning About ADT's

Finally, we consider `remove`.

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

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

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

38

### Reasoning About ADT's

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

Then  $QL.\text{remove}(l)$  is equivalent to  $\langle m, l' \rangle$ .

There are two cases for  $\langle b, f \rangle$ , according to whether or not  $f$  is `nil`.

If  $f = \text{nil}$ , then  $b = b' @ [n]$  for some  $b'$  and  $n$ .

But then  $\text{rev}(b)$  is equivalent to  $n :: \text{rev}(b')$ , which reduces to the non-empty case.

39

### Reasoning About ADT's

Otherwise,  $f = n :: f'$  for some  $n$  and  $f'$ .

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

We must show that:

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

40

### Reasoning About ADT's

Calculating from our assumptions,

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

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

41

### 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.

42

### Parametricity Theorem

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

More precisely,

- if  $e_1, e_2 : \forall(t.\tau)$ , 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  $R$ .

43

### 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.

44