Existential Types
Powerful mechanism for providing data-abstraction
Useful for
Lists of Heterogeneous Values
Typed Closure Conversion
Encoding Modules

Data Abstraction
COS 441
Princeton University
Fall 2004

Heterogeneous Lists Revisited
Consider a heterogeneous list of the form
val x : ?? list = [1, true, “hello world”] :
how can we implement a generic print function?
val printList : ?? list → unit

Tagged List
Simple answer is to use tagged datatype
datatype tagged = I of int | B of bool | S of string
val x = [I 1, B true, S “hello world”]
fun tagged2s (I i) = Int.toString i
| tagged2s (B b) =  Bool.toString b
| tagged2s (S s) = s
fun printList l = …

Problems With Tagged Approach
We must know the type of every value we might want to tag.
If a user creates a new type, they have to modify the tagged2s function as well as the tagged datatype
Not a modular approach

A More Modular Approach
Create a list of “objects” which consist of a value and a function to convert it into a string
type obj = ??
val x : obj list = [(1, Int.toString),
                    (true, Bool.toString),
                    (s,(fn x => x))]
fun obj2s (x,f) = f x
Typing Problems

It doesn’t type check in SML
We can get a similar effect with HOF

```plaintext
type obj = unit → string
val x : obj list = [(fn () => Int.toString 1),
                   (fn () => Bool.toString true),
                   (fn () => s)]
fun obj2s f = f ()
```

Extending the Type System

The type obj should capture the pattern

```plaintext
obj = (τ * (τ → string))
```

for some arbitrary τ

```plaintext
∃ t (t * (t → string))
```

Provides a modular solution that avoids a fixed non-modular tagging scheme.

Closure Conversion

Remember functions represented in runtime as a pair “environment” and function body.
Consider

```plaintext
type obj = unit → string
fun IntObj i = (fn () => Int.toString i)
let val obj = IntObj 10
in obj ()
end
```

Closure Representation

Can represent environments as tuples

```plaintext
fun IntObj (_,i) = (i,fn (i,()) => Int.toString i)
let val (fenv,f) = IntObj (glbl_env,10)
in f (fenv,())
end
```

Size and type of tuple depends on number of free vars. in body of function

Typed Closure Conversion

Closure conversion done by the compiler user doesn’t care what goes on or how the environment is represented!

However, useful to be able check the output of the compiler for debugging and security reasons

Also notice that closures are just like our “object” type i.e. a piece of private data stored with something that knows how to manipulate it

We can us existential to make the result of the compiler type checkable!

Existential Types

```plaintext
Polytypes σ ::= ·.

Expressions e ::= ... 
  | pack r with e as σ
  | open e1 as i with x:x' in e2

Values v ::= ...
  | pack r with v as σ
```
Example: pack

```plaintext
type obj = Θ t((t * (t → string)))

pack int with (1, Int.toString) as Θ t((t * (t → string))

Note: type of (1, Int.toString) = (int * (int → string)) ≅ [t → int](t * (t → string))
```

Example: open

```plaintext
fun obj2s (obj: Θ t((t * (t → string)))) =
open obj as t with x:(t * (t → string)) in
let val (v:t, v2s:(t → string)) = x
in v2s v
end
```

Static Semantics

```
\[ \Delta \cup \{ t \} \vdash \sigma \text{ ok} \quad t \notin \Delta \]
\[ \Delta \vdash \exists f(\tau) \text{ ok} \]
\[ \Delta \vdash \tau \text{ ok} \quad \Delta \vdash \exists f(\tau) \text{ ok} \quad \Gamma \vdash \alpha : \{ \tau \} \sigma \]
\[ \Gamma \vdash \Delta \text{ pack with as } \exists f(\tau) \]
\[ \Delta \vdash \tau \text{ ok} \quad \Gamma, x: \sigma \vdash \Delta \vdash t : \tau \quad \Gamma, x : \sigma \vdash \exists f(\tau) \quad t \notin \Delta \]
\[ \Gamma \vdash \Delta \text{ open } \alpha \text{ as } f \text{ with } x: \sigma \text{ in } \exists f(\tau) \]
```

Dynamic Semantics

```
(\sigma = \exists f(\tau))

open (pack with as \exists f(\tau)) as \{ f \text{ with } x: \sigma \text{ in } \exists f(\tau) \} \to \{ \tau, v/v, x \} \epsilon_c,

\[ \epsilon_c \mapsto \epsilon'_c \]

open \epsilon_c as \{ f \text{ with } x: \sigma \text{ in } \exists f(\tau) \} \to open \epsilon'_c as \{ f \text{ with } x: \sigma \text{ in } \exists f(\tau) \}

\[ \epsilon \mapsto \epsilon' \]

pack with as \tau' \to pack with \tau' as \epsilon'
```

Modules and Existentials

```
signature QUEUE =

sig
  type queue
  val empty : queue
  val insert : int * queue -> queue
  val remove : queue -> int * queue
end

Can be encode with the type
θq((q * ((int * q) -> q)) * (q -> (int * q)))
```

Representation Independence

```
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 simple reference implementation of a queue
pack int list with
  (nil, fun insert _, fun remove _) as θq(q * ((int * q) -> q)) * (q -> (int * q)))
```
Client of a Module

```plaintext
local
open QL
in
...
end
open QL as q
with <e,i,r> : (q * ((int * q) → q) * (q → (int * q)))
in
...
end
```

Representation Independence

- Imagine we have to “equivalent” implementations of a module
- They behave externally the same way but may use different internal representations of data structures
- A client should work correctly assuming the implementations are equivalent

Example

Consider the slightly more efficient representation of a queue

```plaintext
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
```

Formal Definition

Let $\tau_q$ be $(q \times ((int \times q) \rightarrow q) \times (q \rightarrow (int \times q)))$
Let $\sigma_q$ be $\exists q(\tau_q)$
Let the expression that represents the module QL be $\nu_{QL}:\sigma_q$ and the expression that represents the module QFB be $\nu_{QFB}:\sigma_q$
We wish to show for any client code $e_c:\tau_c$
open $\nu_{QL}$ as $q$ with $x:\tau_q$ in $e_c:\tau_c$
behaves equivalently to
open $\nu_{QFB}$ as $q$ with $x:\tau_q$ in $e_c:\tau_c$

What Definition of Equivalent?

We can use the framework of parametricity to define equivalence at some type

```
$\sigma ::= \tau \mid \forall t \in (\sigma) \mid \exists t \in (\sigma)$
$\tau ::= bool \mid int \mid \tau_1 \rightarrow \tau_2$
```

$e \equiv_{val} e' : \sigma$ iff ($\exists v,v', (v \equiv v' \rightarrow e \equiv e') \land v \equiv_{val} e' : \sigma$)

Definition of Value Equivalence

Define a notion of equivalence of closed values indexed by type

```
v \equiv_{val} e : bool  iff  (v = v' = true) \lor (v = v' = false)
v \equiv_{val} e : int  iff  (v = v' = 0)
v \equiv_{val} e : \tau_1 \rightarrow \tau_2  iff  (\forall e_1,e_1', (e_1 \equiv_{val} e_1' : \tau_1) \Rightarrow
                        apply(v,v_1) \equiv_{val} apply(v',v_1') : \tau_2)$
```

4
Definition of Value Equivalence

Interesting case is definition of "all"

\[ v \equiv_{val} v' : \text{all } t \text{ in } (\sigma) \iff \forall \tau, \tau', R : \tau \rightarrow \tau'.(\forall v_1, v_2.(v_1 \equiv_{val} v_2 : t) \Rightarrow (v_1 R v_2)) \Rightarrow e[\tau] \equiv_{exp} e'[\tau'] : \sigma \]

Definition of Value Equivalence

Definition of "some"

\[
\begin{align*}
\text{pack } \tau \text{ with } v \text{ as some } t \text{ in } (\sigma) \equiv_{val} & \text{ pack } \tau' \text{ with } v' \text{ as some } t \text{ in } (\sigma) : \text{some } t \text{ in } (\sigma) \\
\iff & \exists R : \tau \rightarrow \tau'.(\forall v_1, v_2.(v_1 \equiv_{val} v_2 : t) \Rightarrow (v_1 R v_2)) \Rightarrow v \equiv_{val} v' : \sigma
\end{align*}
\]

Some simulation relation

Alternative View

\[
\begin{align*}
\Delta & \vdash \tau_1 : \text{ok} \\
\Gamma, x: \tau_1 \vdash \Delta \vdash e_i : \tau_2 & \Gamma \vdash \Delta \vdash e_i : \exists (\sigma) \quad t \notin \Delta \\
\hline
\Gamma \vdash \Delta \vdash \text{open } e_i \text{ as } f \text{ with } x : \sigma \vdash e_i : \tau_2 \\
\hline
\Gamma \vdash \Delta \vdash e_i \forall (t \rightarrow \tau_2) & \Delta \vdash \tau_2 : \text{ok} \\
\Gamma \vdash \Delta \vdash \text{open } e_i \text{ using } e_i : \tau_2
\end{align*}
\]

Key observation that relates data-abstraction and parametricity

Client Equivalence

So to because the client is just a polymorphic function and from the parametricity theorem we can conclude

\[
\{ v_1, v_2/t \} e_i \equiv_{exp} \{ v_1, v_2/t \} e_i : \tau_2
\]

if we know \( v_1 \) and \( v_2 \) are equivalent existential packages

Equivalence of Modules

Definition of "some"

\[
\begin{align*}
\text{pack } \tau \text{ with } v \text{ as some } t \text{ in } (\sigma) \equiv_{val} & \text{ pack } \tau' \text{ with } v' \text{ as some } t \text{ in } (\sigma) : \text{some } t \text{ in } (\sigma) \\
\iff & \exists R : \tau \rightarrow \tau'.(\forall v_1, v_2.(v_1 \equiv_{val} v_2 : t) \Rightarrow (v_1 R v_2)) \Rightarrow v \equiv_{val} v' : \sigma
\end{align*}
\]

To show modules equivalent choose some \( R \) and show implementations are equivalent under \( R \)

To show modules equivalent choose some \( R \) and show implementations are equivalent under \( R \)

\[
\begin{align*}
\forall v_{QL} \equiv_{val} v_{QFB} : \sigma_q \\
\text{First we must choose some } R \\
\text{Let } R \text{ be } \\
R = \{ (l, (b, f)) \mid l \equiv_{val} \text{brev}(f) \}
\end{align*}
\]

Then assuming that \( v_1 \equiv_{val} v_2 : q \) if \( (v_1, v_2) \in R \) show

1. \( QL.\text{empty} \equiv_{val} QFB.\text{empty} : q \)
2. \( QL.\text{insert} \equiv_{val} QFB.\text{insert} : (\text{int} * q) \rightarrow q \)
3. \( QL.\text{remove} \equiv_{val} QFB.\text{remove} : q \rightarrow (\text{int} * q) \)
Note About Proofs

Proofs are tedious
Have you ever seen an interesting proof yet? 😊
Formalisms we present in class are to make you aware that precise formal definitions exist for abstraction and modularity

What Do I Need To Know?

I had to read through Harper’s notes several times carefully to understand them!
Therefore I won’t assign a homework/test question about formal notions of parametricity/data abstraction!
However, it is educational to understand the high-level structure of the definitions and how they relate to one another

Summary

Existential type allow you to enforce data-abstraction
Data-abstraction exists because “client” code is polymorphic with respect to the actual type
Can encode modules, closures, and objects with existential types in a type safe way