CSE341: Programming Languages

Lecture 10
References, Polymorphic Datatypes, the Value Restriction, Type Inference

Ben Wood, filling in for Dan Grossman

Fall 2011

Callbacks

A common idiom: Library takes functions to apply later, when an *event* occurs – examples:

- When a key is pressed, mouse moves, data arrives
- When the program enters some state (e.g., turns in a game)

A library may accept multiple callbacks

- Different callbacks may need different private data with different types
- Fortunately, a function's type does not include the types of bindings in its environment
- (In OOP, objects and private fields are used similarly, e.g., Java Swing's event-listeners)

Mutable state

While it's not absolutely necessary, mutable state is reasonably appropriate here

- We really do want the "callbacks registered" and "events that have been delivered" to *change* due to function calls

For the reasons we have discussed, ML variables really are immutable, but there are mutable references (use sparingly)

- New types: **`t ref`** where **`t`** is a type
- New expressions:
  - **`ref e`** to create a reference with initial contents e
  - **`e1 := e2`** to update contents
  - **`!e`** to retrieve contents (not negation)

References example

```
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check)
```

- A variable bound to a reference (e.g., **`x`**) is still immutable: it will always refer to the same reference
- But the contents of the reference may change via **`:=`**
- And there may be aliases to the reference, which matter a lot
- Reference are first-class values
- Like a one-field mutable object, so **`:=`** and **`!`** don't specify the field

Example call-back library

Library maintains mutable state for “what callbacks are there” and provides a function for accepting new ones

- A real library would support removing them, etc.
- In example, callbacks have type **`int->unit`** (executed for side-effect)

So the entire public library interface would be the function for registering new callbacks:

```
val onKeyEvent : (int -> unit) -> unit
```

Library implementation

```
val cbs : (int -> unit) list ref = ref []

fun onKeyEvent f =  cbs := f :: (!cbs)

fun onEvent i =
    let fun loop fs =
            case fs of
              []      => ()
            | f::fs’  => (f i; loop fs’)
    in loop (!cbs) end
```

Clients

Can only register an **`int -> unit`**, so if any other data is needed, must be in closure's environment

- And if need to "remember" something, need mutable state

Examples:

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ =>
            timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
   onKeyEvent (fn j =>
       if i=j
       then print ("pressed " ^ Int.toString i)
       else ())
```

More about types

- Polymorphic datatypes, type constructors
- Why do we need the Value Restriction?
- Type inference: behind the curtain

Polymorphic Datatypes

```
datatype int_list =
     EmptyList
   | Cons of int * int_list

datatype ‘a non_mt_list =
     One of ‘a
   | More of ‘a * (‘a non_mt_list)

datatype (‘a,‘b) tree =
     Leaf of ‘a
   | Node of ‘b * (‘a,‘b) tree * (‘a,‘b) tree

val t1 = Node(“hi”,Leaf 4,Leaf 8)
                   (* (int,string) tree *)
val t2 = Node(“hi”,Leaf true,Leaf 8)
                   (* does not typecheck *)
```

Polymorphic Datatypes

```
datatype 'a list = [] | :: of 'a * ('a list)
                 (* if this were valid syntax *)

datatype 'a option = NONE | SOME of 'a
```

- `list`, `tree`, etc. are **not types**; they are ***type constructors***
- `int list`, `(string,real) tree`, etc. are types.
- Pattern-matching works on all datatypes.

The Value Restriction Appears ☹

If you use partial application to create a polymorphic function, it may not work due to the value restriction

- Warning about “type vars not generalized”
  - And won’t let you call the function
- This should surprise you; you did nothing wrong ☺ but you still must change your code
- See the written lecture summary about how to work around this wart (and ignore the issue until it arises)
- The wart is there for good reasons, related to mutation and not breaking the type system

Purpose of the Value Restriction

```
val xs = ref []
       (* xs : ‘a list ref *)
val _ = xs := [“hi”]
       (* instantiate ‘a with string *)
val y = 1 + (hd (!xs))
       (* BAD: instantiate ‘a with int *)
```

- A binding is only allowed to be polymorphic if the right-hand side is:
  - a variable; or
  - a value (including function definitions, constructors, etc.)
- `ref []` is not a value, so we can only give it non-polymorphic types such as `int list ref` or `string list ref`, but not `‘a list ref.`

Downside of the Value Restriction

```
val pr_list = List.map (fn x => (x,x))  (* X *)

val pr_list : int list -> (int*int) list =
     List.map (fn x => (x,x))

val pr_list =
   fn lst => List.map (fn x => (x,x)) lst

fun pr_list lst = List.map (fn x => (x,x)) lst
```

- The SML type checker does not know if the `'a list` type uses references internally, so it has to be *conservative* and assume it could.
- In practice, this means we need to be more explicit about partial application of polymorphic functions.

Type inference: sum

```
fun sum xs =
   case xs of
       [] => 0
     | x::xs' => x + (sum xs')
```

```
sum : t1 -> t2
 xs : t1
  x : t3
xs' : t4
```

```
t1 = t5  list
t2 = int
t3 = t5
t4 = t5  list
t3 = int
t1 = t4
```

Type inference: sum

```
fun sum xs =
   case xs of
         [] => 0
       | x::xs' => x + (sum xs')
```

sum : t1 -> **int**

xs : t1

x : **int**

xs' : **t1**

t1 = **int** list

t2 = int

int = t5

t1 = t5 list

t3 = int

t1 = t4

Type inference: sum

```
fun sum xs =
   case xs of
         [] => 0
       | x::xs’ => x + (sum xs’)
```

```
sum : int list -> int
 xs : int list
  x : int
xs’ : int list
```

```
 t1 = int list
 t2 = int
int = t5
 t1 = t5 list
 t3 = int
 t1 = t4
```

Type inference: length

```
fun length xs =
  case xs of
      [] => 0
    | _::xs’ => 1 + (length xs’)
```

```
length : t1 -> t2
    xs : t1
   xs’ : t3
```

```
t1 = t4 list
t2 = int
t3 = t4 list
t1 = t3
```

Type inference: length

```
fun length xs =
   case xs of
        [] => 0
      | _::xs' => 1 + (length xs')
```

```
length : t1 -> int
    xs : t1
   xs' : t1
```

```
t1 = t4 list
t2 = int
t1 = t4 list
t1 = t3
```

Type inference: length

```
fun length xs =
   case xs of
         [] => 0
       | _::xs' => 1 + (length xs')
```

`length :` **`'a list -> int`**

`xs :` **`t4 list`** `->` **`int`**

`xs' :` **`t4 list`**

`t1 = t4 list`

`t2 = int`

`t1` `= t4 list`

`t1 = t3`

`length` works no matter what `'a` is.

Type inference: compose

```
fun compose (f,g) = fn x => f (g x)
```

```
compose : t1 * t2 -> t3
      f : t1
      g : t2                    t3 = t4 -> t5
      x : t4                    t2 = t4 -> t6
                                t1 = t6 -> t7
                                t5 = t7
```

Type inference: compose

```
fun compose (f,g) = fn x => f (g x)
```

```
compose : (t6 -> t5) * (t4 -> t6) -> (t4 -> t5)
      f : t6 -> t5
      g : t4 -> t6                  t3 = t4 -> t5
      x : t4 -> t5                  t2 = t4 -> t6
                                    t1 = t6 -> t5
                                    t5 = t7
```

Type inference: compose

```
fun compose (f,g) = fn x => f (g x)
```

```
compose : (‘a -> ‘b) * (‘c -> ‘a) -> (‘c -> ‘b)
      f : t6 -> t5
      g : t4 -> t6              t3 = t4 -> t5
      x : t4 -> t5              t2 = t4 -> t6
                                t1 = t6 -> t5
                                t5 = t7

compose : (‘b -> ‘c) * (‘a -> ‘b) -> (‘a -> ‘c)
```

Type inference: broken sum

```
fun sum xs =
   case xs of
         [] => 0
       | x::xs’ => x + (sum x)
```

```
sum : t1 -> t2
 xs : t1
  x : t3
xs’ : t4
```

```
t1 = t5  list
t2 = int
t3 = t5
t4 = t5  list
t3 = int
t1 = t3
```

Type inference: sum

```
fun sum xs =
   case xs of
         [] => 0
       | x::xs’ => x + (sum x)
```

sum : **int**-> **int**
xs : **int**
x : **int**
xs’ : **int list**

int = **int** list
t2 = int
int = t5
t4 = t5 list
t1 = int
t1 = t3

Parting comments on ML type inference

- You almost never have to write types in ML (even on parameters), with some minor caveats.
- Hindley-Milner type inference algorithm
- ML has no subtyping. If it did, the equality constraints we used for inference would be overly restrictive.
- Type variables and inference are not tied to each. Some languages have one without the other.
  - Type variables alone allow convenient code reuse.
  - Without type variables, we cannot give a type to compose until we see it used.