CSE341: Programming Languages

Lecture 9
Function-Closure Idioms

Dan Grossman
Winter 2013
More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:

- Pass functions with private data to iterators: Done
- Combine functions (e.g., composition)
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming)
- Implementing an ADT with a record of functions (optional)
Combine functions

Canonical example is function composition:

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

- Creates a closure that “remembers” what \( f \) and \( g \) are bound to
- Type \( ('b -> 'c) * ('a -> 'b) -> ('a -> 'c) \)
  but the REPL prints something equivalent

- ML standard library provides this as infix operator \( \circ \)
- Example (third version best):

```ml
fun sqrt_of_abs i = Math.sqrt (Real.fromInt (abs i))
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```
Left-to-right or right-to-left

val sqrt_of_abs = Math.sqrt o Real.fromInt o abs

As in math, function composition is “right to left”
- “take absolute value, convert to real, and take square root”
- “square root of the conversion to real of absolute value”

“Pipelines” of functions are common in functional programming and many programmers prefer left-to-right
- Can define our own infix operator
- This one is very popular (and predefined) in F#

infix |
fun x |> f = f x

fun sqrt_of_abs i =
i |> abs |> Real.fromInt |> Math.sqrt
Another example

• “Backup function”

```ml
fun backup1 (f,g) = fn x => case f x of
  NONE => g x
| SOME y => y
```

• As is often the case with higher-order functions, the types hint at what the function does

  (`'a -> `'b option) * (`'a -> `'b) -> `'a -> `'b
More idioms

• We know the rule for lexical scope and function closures
  – Now what is it good for

A partial but wide-ranging list:

• Pass functions with private data to iterators: Done
• Combine functions (e.g., composition)
• Currying (multi-arg functions and partial application)
• Callbacks (e.g., in reactive programming)
• Implementing an ADT with a record of functions (optional)
Currying

• Recall every ML function takes exactly one argument
• Previously encoded $n$ arguments via one $n$-tuple
• Another way: Take one argument and return a function that takes another argument and…
  – Called “currying” after famous logician Haskell Curry
Example

```plaintext
val sorted3 = fn x => fn y => fn z =>
    z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```

• Calling `(sorted3 7)` returns a closure with:
  – Code `fn y => fn z => z >= y andalso y >= x`
  – Environment maps `x` to `7`

• Calling `that` closure with `9` returns a closure with:
  – Code `fn z => z >= y andalso y >= x`
  – Environment maps `x` to `7`, `y` to `9`

• Calling `that` closure with `11` returns `true`
Syntactic sugar, part 1

In general, \(e_1\ e_2\ e_3\ e_4\ \ldots\), means \((\ldots((e_1\ e_2)\ e_3)\ e_4)\)

So instead of \(((\text{sorted3}\ 7)\ 9)\ 11\),
can just write \text{sorted3}\ 7\ 9\ 11

Callers can just think “multi-argument function with spaces instead of a tuple expression”
  – Different than tupling; caller and callee must use same technique

\begin{verbatim}
val sorted3 = fn x => fn y => fn z =>
    z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
\end{verbatim}
Syntactic sugar, part 2

In general, `fun f p1 p2 p3 ... = e`,
means `fun f p1 = fn p2 => fn p3 => ... => e`

So instead of `val sorted3 = fn x => fn y => fn z => ...` or `fun sorted3 x = fn y => fn z => ...`,
can just write `fun sorted3 x y z = x >= y andalso y >= x`

Callees can just think “multi-argument function with spaces instead of a tuple pattern”
  – Different than tupling; caller and callee must use same technique

```
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```
As elegant syntactic sugar (even fewer characters than tupling) for:

```ml
val t1 = sorted3 7 9 11
```

```ml
fun sorted3 x y z = z >= y andalso y >= x
```

```ml
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```
Curried fold

A more useful example and a call too it
– Will improve call next

```haskell
fun fold f acc xs =
    case xs of
        [] => acc
        | x::xs' => fold f (f(acc,x)) xs'

fun sum xs = fold (fn (x,y) => x+y) 0 xs
```
“Too Few Arguments”

• Previously used currying to simulate multiple arguments

• But if caller provides “too few” arguments, we get back a closure “waiting for the remaining arguments”
  – Called partial application
  – Convenient and useful
  – Can be done with any curried function

• No new semantics here: a pleasant idiom
Example

```
fun fold f acc xs =
    case xs of
        []    => acc
        | x::xs' => fold f (f(acc,x)) xs'

fun sum_inferior xs = fold (fn (x,y) => x+y) 0 xs

val sum = fold (fn (x,y) => x+y) 0
```

As we already know, \( \text{fold} \ (\text{fn} \ (x,y) \Rightarrow x+y) \ 0 \) evaluates to a closure that given \( \text{xs} \), evaluates the case-expression with \( f \) bound to \( \text{fold} \ (\text{fn} \ (x,y) \Rightarrow x+y) \) and \( \text{acc} \) bound to 0.
Unnecessary function wrapping

\[
\text{fun sum\_inferior\ xs = fold (fn (x, y) => x+y) 0 xs}
\]

\[
\text{val sum = fold (fn (x, y) => x+y) 0}
\]

• Previously learned not to write \text{fun f x = g x}
when we can write \text{val f = g}

• This is the same thing, with \text{fold (fn (x, y) => x+y) 0} in place of \text{g}
Iterators

- Partial application is particularly nice for iterator-like functions
- Example:

```ml
fun exists predicate xs =
  case xs of
    [] => false
  | x::xs' => predicate xs
    orelse exists predicate xs'

val no = exists (fn x => x=7) [4,11,23]
val hasZero = exists (fn x => x=0)
```

- For this reason, ML library functions of this form usually curried
  - Examples: `List.map`, `List.filter`, `List.foldl`
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 code for workarounds

- Can discuss a bit more when discussing type inference
**More combining functions**

- What if you want to curry a tupled function or vice-versa?
- What if a function’s arguments are in the wrong order for the partial application you want?

Naturally, it is easy to write higher-order wrapper functions
  – And their types are neat logical formulas

```fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y```

Winter 2013

CSE341: Programming Languages
Efficiency

So which is faster: tupling or currying multiple-arguments?

• They are both constant-time operations, so it doesn’t matter in most of your code – “plenty fast”
  – Don’t program against an implementation until it matters!

• For the small (zero?) part where efficiency matters:
  – It turns out SML/NJ compiles tuples more efficiently
  – But many other functional-language implementations do better with currying (OCaml, F#, Haskell)
    • So currying is the “normal thing” and programmers read \( t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \) as a 3-argument function that also allows partial application
More idioms

• We know the rule for lexical scope and function closures
  – Now what is it good for

A partial but wide-ranging list:
• Pass functions with private data to iterators: Done
• Combine functions (e.g., composition)
• Currying (multi-arg functions and partial application)
• Callbacks (e.g., in reactive programming)
• Implementing an ADT with a record of functions (optional)
**ML has (separate) mutation**

-Mutable data structures are okay in some situations
  - When “update to state of world” is appropriate model
  - But want most language constructs truly immutable

-ML does this with a separate construct: references

-Introducing now because will use them for next closure idiom

-Do not use references on your homework
  - You need practice with mutation-free programming
  - They will lead to less elegant solutions
References

• New types: \( t \text{ ref} \) where \( t \) is a type

• New expressions:
  – \texttt{ref }e\texttt{ } to create a reference with initial contents \texttt{e}
  – \texttt{e1 := e2} to update contents
  – \texttt{!e} to retrieve contents (not negation)
References example

```ocaml
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
- References are first-class values
- Like a one-field mutable object, so `:=` and `!` don’t specify the field
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” to change when a function to register a callback is called
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 all support removing them, etc.
- In example, callbacks have type \texttt{int->unit}

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

\begin{verbatim}
val onKeyEvent : (int -> unit) -> unit
\end{verbatim}

(Because callbacks are executed for side-effect, they may also need mutable state)
Library implementation

```haskell
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 \texttt{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:

\begin{verbatim}
val timesPressed = ref 0
val _ = onKeyEvent (fn _ =>
    timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
    onKeyEvent (fn _ =>
        if i=j
        then print ("pressed " ^ Int.toString i)
        else ()
\end{verbatim}
More idioms

• We know the rule for lexical scope and function closures
  – Now what is it good for

A partial but wide-ranging list:

• Pass functions with private data to iterators: Done
• Combine functions (e.g., composition)
• Currying (multi-arg functions and partial application)
• Callbacks (e.g., in reactive programming)
• Implementing an ADT with a record of functions (optional)
**Optional: Implementing an ADT**

As our last idiom, closures can implement *abstract data types*
  
  – Can put multiple functions in a record
  – The functions can share the same private data
  – Private data can be mutable or immutable
  – Feels a lot like objects, emphasizing that OOP and functional programming have some deep similarities

See code for an implementation of immutable integer sets with operations *insert*, *member*, and *size*

The actual code is advanced/clever/tricky, but has no new features
  
  – Combines lexical scope, datatypes, records, closures, etc.
  – Client use is not so tricky