Code Reuse: Closures and Polymorphism

March 21, 2001

CS 131: Programming Languages

Code Duplication

- Often tempting to duplicate useful pieces of code in a program (possibly with minor changes).
  - Or, several independent implementations of the same task
- This has serious disadvantages (why?)

Code Duplication

"We heard firsthand of a U.S. state whose governmental computer systems were surveyed for Y2K compliance. The audit turned up more than 10,000 programs, each containing its own version of Social Security number validation."

Hunt and Thomas, *The Pragmatic Programmer*

Avoiding Duplication

- Many language mechanisms are designed to avoid the need for duplication
  - i.e., to permit code reuse
- Such as?

Functions/Procedures

- Fairly obvious that functions (and procedures and subroutines) permit code reuse.
- So today, we'll just revisit higher-order functions.
  - Abstracting control flow
  - Closures
    - first-class functions as values

Abstracting Control Flow

- Given a data structure, certain patterns of control tend to re-occur
- E.g., lists
  - Do something to every element of a list
  - Apply a transformation to every element of a list.
  - Process each list element, updating an accumulator

List Transformations

```
fun double_list ([] : int list) = []
  | double_list (n::ns) =
        (2*n) :: (double_list ns)

fun square_list ([] : real list) = []
  | square_list (r::rs) =
        (r*r) :: (square_list rs)

fun asciify_list ([] : char list) = []
  | asciify_list (c::cs) =
        (Char.ord c) :: (asciify_list cs)
```

Better List Transformations

```
fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)

val double_list = map (fn n => 2*n)

val square_list = map (fn (r:real) => r*r)

val asciify_list = map Char.ord
```

Constraints on Code Pointers

- In C one can pass/return function pointers

```
void qsort
     (void *base,  size_t nel,  size_t  width,
      int (*compar)(const void *, const void
*));
```

- But, any function passed to sort must be written completely by the programmer.
  - cannot "create" functions at run-time

```
fun compare_nth_char n =
  fn (s1:string,s2:string) =>
       String.sub(x,n) > String.sub(y,n)
```

Example: X-Windows

- The X Toolkit defines an event-driven model for X-windows
  - Can specify functions to be called when a particular event occurs (e.g, button pressed)
  - These functions called "upcalls" or "callbacks" because rather than making calls *to* the system, these functions are *called by* the system.
    - Similar structure appears in, for example, networking code where we want to specify something to do each time a message packet arrives.

Upcall Example

- Suppose we want to create several buttons
- Each button should cause a different string to be printed.
- How can we do this?

Upcalls in SML

- The button-creating function might be something like:

```
type button
make_button : (unit -> unit) -> button
```

- Given this, we could create two buttons.

```
fun f1 () = print "aha"
fun f2 () = print "oho"
val b1 = make_button f1
val b2 = make_button f2
```

Better Upcalls in SML

- This is ok since f1 and f2 are small, but what if each function had to open a new window just to display its string? Lots of duplication.
- Better structure would be:

```
fun printer s = (fn () => print s)

val b1 = make_button (printer "aha")
val b2 = make_button (printer "oho")
```

Upcalls in C

```
void f1(Widget w, XtPointer x, XtPointer y) {
   printf("aha");
}
void f2(Widget w, XtPointer x, XtPointer y) {
   printf("oho");
}
int main() {
   Widget button1, button2;
   /* …initialization code for buttons here… */
   XtAddCallback(button1, XmNactivateCallback, f1, 0);
   XtAddCallback(button2, XmNactivateCallback, f2, 0);
}
```

Better Upcalls in C

```
void printer(Widget w, XtPointer client_data,
             XtPointer y) {
   printf("%s", client_data);
}

int main() {
   Widget button1, button2;
   /* …initialization code for buttons here… */
   XtAddCallback(button1, XmNactivateCallback,
                 printer, "aha");
   XtAddCallback(button2, XmNactivateCallback,
                 printer, "oho");
}
```

POSIX Threads

- Recall: our couroutines package included

```
spawn : (unit->unit) -> unit
```

- The C call to create a POSIX thread is:

```
int pthread_create(pthread_t *new_thread_ID,
          const pthread_attr_t *attr,
          void * (*start_func)(void *), void *arg);
```

"The thread is created executing `start_func` with `arg` as its sole argument. If the `start_func` returns, the effect is as if there was an implicit call to `pthread_exit()`"

Summary: Higher-Order Functions

- Higher-order functions can be used to abstract out common patterns of control flow
  - Simplifies code
  - Particularly useful if the implementation of a data structure is hidden (abstract)
    - E.g., do something to every element of a `set`
    - Same motivation leads to *iterators* in object-oriented languages.

Summary: Higher-Order Functions

- Higher-order functions permit automatic creation of similar functions.
  - Closure is just a piece of code plus some data
    - In SML, the data is the values of its free variables
  - Even languages that don't directly support closures can often mimic them.
    - Not as convenient as in SML
    - Need to think ahead, as the Xt designers did
    - But even so, can be quite useful.
    - [In this particular case, could use macros to create the functions. But what if strings were run-time values?]
  - Note: There are definitely similarities between objects and closures. Discussion delayed until next week.

Complaints about Strong Typing

- Types get in the way
  - Too obtrusive: too many type annotations
  - Too restrictive: types inhibit code re-use

```
val compose =
  fn (g : real->string) : ((int->real)->(int->string)) =>
     fn (f : int->real) : (int->string) =>
         fn (x:int):string => g(f(x))
```

Improving Matters

- Polymorphism: generic functions

```
val compose =
  fn (g : 'b->'c) : (('a->'b)->('a->'c)) =>
      fn (f : 'a->'b) : ('a->'c) =>
           fn (x:'a):'c => g(f(x))
```

- Implicit typing: automatically inferred annotations

```
val compose =
    fn g =>
       fn f =>
          fn x => g(f(x))
```

Brands of Polymorphism

- Parametric polymorphism (today's topic)
  - Generic code
  - Algorithm stays the same even when types differ.
- Ad-hoc polymorphism (overloading)
  - Different code runs depending on types
  - Choice may be compile-time or run-time

A Puzzle

- Consider the definition

```
fun id x = x
```

- SML would say that

```
id : 'a -> 'a
```

meaning that, for any type `'a`, if `id` is given a value of type `'a` then the return value (if any) will also have the same type `'a`.

```
val x : int*real = (id 3, id 4.0)
```

A Puzzle

- Now consider the definition

```
fun apply(f:'a ->'a, x:'a) = f(x)
```

- SML is perfectly happy to then compile

```
fun increment x = x+1
val y = apply(increment, 3)
```

even though the first argument to `apply` clearly isn't polymorphic. What's going on?

Solution

- We need to distinguish between the two possible meanings of `'a -> 'a`.
  - A function mapping values of type `'a` to values of type `'a`, for *any* type `'a`.
  - A function mapping values of type `'a` to values of type `'a`, for s*ome fixed but unspecified* type `'a`.
- Traditionally distinguished by writing

  `∀'a. 'a->'a`

  vs.

  `'a -> 'a`
- Universally quantified types are called *polymorphic* types.

Understanding SML Types

- When considering the types of `val` or `fun` bindings, types are implicitly assumed to start with universal quantifiers for every type variable appearing
  - [This is almost true ... will be corrected later]
  - The type `'a->'a` of `id` is "really" `∀'a. 'a->'a`
- For all other types, no universal quantifier is assumed.
  - The type `'a->'a` of `f` is "really" `'a->'a`
  - The type of apply, however, is "really"
    `∀'a. (('a->'a) * 'a) -> 'a`

Views of Parametric Polymorphism

- Consider the code

```
fun id x = x
val x : int*real = (id 3, id 4.0)
```

what's really going on?

Boxed View

- The function `id` simultaneously has type `int->int`, and type `real->real`, and an infinite number of other types, all of which are summarized as `∀'a. 'a->'a`.
- So the same piece of code is getting called twice.

```
fun id x = x
val x : int*real = (id 3, id 4.0)
```

Template View

- If we had one copy of the code for `id` for each use of the function, we wouldn't need polymorphism.

```
fun id1 x = x
fun id2 x = x
val x : int*real = (id1 3, id2 4.0)
```

- So, being able to write the function once is just a convenience, and the type `∀'a. 'a->'a` means that for any use of the function we can (in the compiler) generate a specific version of the code for that type.

```
fun id x = x
val x : int*real = (id 3, id 4.0)
```

Template View

- Note: in contrast to C++ templates, polymorphic functions can always be typechecked in isolation.
  - Once we know its type, we can tell whether any use of the function is ok.
  - Don't have to first expand out the definition, plug in the arguments, and check that the resulting code compiles.
    - get error messages earlier.

Type Parameter View

- The type `∀'a. 'a->'a` for `id` means that the it needs a type `'a` as an implicit (run-time) argument.
- So, what is really happening is something like

```
fun id ('a:TYPE) (x:'a) = x
val x : int*real =
   (id(int)(3), id(real)(4.0))
```

where in principle, the (single) implementation for `id` could look at the type and modify its behavior appropriately.

- e.g., change the calling convention.

Which View Is Right?

- In SML you cannot tell.
  - In fact, there are compilers which use each of these strategies for compiling polymorphism.
  - However, for some useful extensions of the language, you can no longer get rid of polymorphism by duplicating functions in the compiler, so #2 and #3 are always interchangeable.

Parametricity

- Consider SML without side-effects or infinite loops.
- Suppose `f` has type `∀'a. 'a->'a`
  - What function could it be?

Parametricity

- Suppose `f` has type `∀'a. 'a list ->'a list`
- What can `f` be?

Parametricity

- If `f` has type `∀'a. 'a list ->'a list`
  - `f` could be an identity function
  - `f` could always return `nil`
  - `f` could return a fixed sub-list of its argument
  - `f` could return a permutation of its argument
  - The function *cannot* depend on the elements of the list, only on its structure.
  - The function *cannot* return elements in the output list that were not in the input list.
  - The function *cannot* work differently depending on the type of the list argument.

Parametricity

- Informally, a polymorphic function is said to be *parametric* if its behavior is independent of its type argument.
  - i.e., same algorithm for all type instances.
- This can be elegantly formalized
  - "Related arguments yield related results"
  - But not in this class.
  - Application: TAL and callee-save registers