Callbacks/Upcalls

- When "system" code calls "user" code, this is called a **callback** or **upcall**.
  - Reverses the normal case of user code making calls to the underlying system.
  - Frequently the user function to call is passed to the system via a normal function call.
- Examples:
  - Having a function run every time a GUI button is pressed
  - Having a function run every time a packet arrives on the network
  - Other "event-driven" code

Callbacks in SML

- Suppose we have an interface something like:

  ```sml
  type button
  addCallback : button * (unit -> unit) -> unit
  ```

- Then, assuming we have buttons **button1** and **button2**:

  ```sml
  fun f1 () = print "aha"
  fun f2 () = print "oho"
  val _ = addCallback(button1,f1)
  val _ = addCallback(button2,f2)
  ```

Better Callbacks 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 code duplication...
- Better structure would be:

  ```sml
  fun printer s = (fn () => print s)
  val b1 = make_button (printer "aha")
  val b2 = make_button (printer "oho")
  ```
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)

- The X Toolkit includes support for buttons
  - Can tell a button to invoke a function when pressed, by passing a function pointer to the XtAddCallback function
  - Function is specified via a function pointer

Callbacks in C

```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);
}
```

Constraints on C Function Pointers

- In C one can pass/return function pointers

```c
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. (Why?)

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

Better Callbacks?

- There is no direct equivalent in C to the printer higher-order function.
  - If we want two buttons which do similar but slightly different things, must we write two completely separate functions?
Closures vs. Code Pointers

• A function pointer in C is just the address of a piece of code generated by the compiler.

```
0x4C00: add $1, $2, $3
0x4C04: cmp $2, 3, $5
0x4C08: bgt $5, 4C16
0x4C12: ret
0x4C16: mov $2, $0
0x4C20: ...
```

Informal Closures

• We can approximate what's going on in at run-time, using SML source code.
  - A closure is a pair containing a function with no free variables (i.e., code), and some data (the "environment")
  - The function code expects to get the data as an extra argument!
  - Whenever a closure is applied, we need to pass to the code both the "actual" argument and the data part of the closure.

```
fun printer s =
  (fn () => print s)
val f1 =
  printer "aha"

fun anon_code(s,()) =
  print s
fun printer_code((),s) =
  (s,anon_code)
val printer =
  ((), printer_code)
val f1 =
  (#2 printer)
val f1 =
  (#1 printer, "aha")
val f1 =
  (#2 f1)(#1 f1, ())
```
Manual Closures in C

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

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

Comparison: POSIX Threads

• The C call to create a POSIX thread is:

```c
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()"

Continuations in C

• See handout (if interested) on an application of continuations in an operating system.

• Issue:
  - OS is handling many threads/processes at any given time.
  - Each one needs a kernel stack when executing system calls.
  - Many processes may be waiting in the kernel (e.g., for I/O) at the same time, so can't necessarily share a stack.

• Paper's suggestion:
  - Allow a thread that is about to block in the kernel to specify a continuation function, called when it's time to restart.
  - Thread responsible for saving all data the continuation will need.
  - Kernel can then safely discard/reuse the thread's kernel stack.

Comparison: Java Callbacks

```java
class AhaPrinter implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.print("aha");
    }
}
class OhoPrinter implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.print("oho");
    }
} 
JButton button1 = new JButton("B1");
JButton button2 = new JButton("B2");
button.addActionListener(new AhaPrinter());
button.addActionListener(new OhoPrinter());
```
**Improved Java Callbacks**

```java
class Printer implements ActionListener {
    String s;
    Printer(String s) {this.s = s; }
    public void actionPerformed(ActionEvent e) {
        System.out.print(this.s);
    }
}

JButton button1 = new JButton("B1");
JButton button2 = new JButton("B2");
button.addActionListener(new Printer("aha"));
button.addActionListener(new Printer("oho"));
```

**Closures vs. Objects**

- What's the difference between an object and a closure?

- What's the difference between a method and a function?

**Is SML Object-Oriented?**

```sml
fun newcell() =
    let
        val r = ref 0
        in
            {contents = r,
                get = fn() => !r,
                set = fn n => (r := n))
        end
    val mycell = newcell()
    val _ = (#contents mycell) := 3
    val x = (#set mycell) ((#get mycell)() * 2)
```

**Object Representation**

```java
class Point {
    public:
        Point (double, double);
        virtual ~Point;
        virtual void move(double, double);
        double x;
        double y;
};

p = new Point(17.0, 14.0);
q = new Point(15.3, 14.0);
```
**Other Class Components**

```cpp
class Point2 {
public:
    Point (double, double);
    virtual ~Point;
    virtual void move(double, double);
    double x, y;
    static int points;
    int getX();
};
```

```cpp```
p = new Point2(17.0, 14.0);
q = new Point2(15.3, 14.0);
```cpp```
Comments

- Object layout doesn't care about public / private / protected
  - These are solely a matter of scoping; the typechecker ensures that abstraction is not violated.
- Interface of a class uniquely determines the class layout:
  - e.g., byte offsets of fields or methods
  - This is why you have to list all the private components of the object.
  - Contributes to "fragile base class" problem.

Objects and Subclasses

class Cell {
    int contents;
    virtual int get();
    virtual void set(int);
};
class RCell : public Cell {
    int backup;
    virtual void restore();
    /* overrides set to store contents into backup */
};
Cell *c = new Cell;
RCell *rc = new RCell;

C++ Representation

Classes

- Classes in C++ and Java provide
  - Ability to create objects
  - Repository for related code (static)
  - Access control (public/private)
  - Code reuse via inheritance
  - Abstract types
  - Subtyping
Object-Based Languages

- Languages with objects don’t necessarily require classes
  - Such languages are often called object-based
  - Often untyped.
- Where do objects come from, without classes?
  - Where do tuples/records come from in SML?
  - Many object-based languages are organized around “prototypes”
    - Embedding: objects created by cloning a prototype and modifying it (adding or replacing fields/methods)
    - Delegation: objects can contain a pointer to the prototype which handles any method or field referenced they don’t know about.

Example: JavaScript

```javascript
function cell_get() {
    return this.contents;
};
function cell_set(n) {
    this.contents = n;
};
var mycell = new Object();
mycell.contents = 0;
mycell.get = cell_get;
mycell.set = cell_set;
```

Example: JavaScript

```javascript
function Cell () {
    this.contents = 0;
    this.get = cell_get;
    this.set = cell_set;
};
var mycell = new Cell();
```

Delegation

- Subclassing without classes
- Each object internally references another object, called the prototype
  - If we fail to find a field or method in an object, try looking in the object’s parent, the parent’s parent, etc.
  - Efficiency: many objects can share the same parent, so they don’t have to each have a copy of the parent’s fields and methods.
  - Adding methods to the parent causes new code to show up in all the child objects too. (Even built-in objects like strings!)