Procedures as Parameters

It can be handy to pass procedures as parameters to other, higher-order procedures. This feature is supported by many languages, including Pascal, Ada, ML, and C/C++ (but not directly by Java).

Examples:

- Parameterized algorithms (e.g. in C):

        typedef int (* leqfn) (int,int);
        void isort(int n, int a[], leqfn leq) {
            int i,j,t;
            for (i = n-1; i >= 0; i--) {
                t = a[i];
                for (j = i;
                    j < n-1 && leq(a[j+1],t);
                    j++)
                    a[j] = a[j+1];
                a[j] = t;
            }
        }

        intup(int p,int q) { return p <= q; }
        int down(int p, int q) { return p >= q; }

        int a[] = {2,1,3};
        isort(3, a, up);    /* a = {1,2,3} */
        isort(3, a, down);  /* a = {3,2,1} */

- Call-backs from surrounding system:

        typedef void (* click_handler)(int);
        void registerClickHandler(click_handler h);

        Application defines and registers call-back function:

        void handler(int button) {
            switch(button) {
                case 1: cut();
                case 2: copy();
                case 3: paste();
            }
        }

        registerClickHandler(handler);

- Parameterized data structure traversals (e.g. in ML)

        fun map (g:'a -> 'b, u: 'a list) : 'b list =
            case u of
                nil => nil
                | (h::t) => (g h)::(map (g,t))

        fun inc x = x + 1
        fun paren s = "(" ˆ s ˆ ")"
        val w = map (inc, [1,2,3])  (* yields [2,3,4] *)
        val z = map (paren, ["a","bc",""])  (* yields ["(a)","(bc)","()" ] *)

        ML also supports anonymous function values, i.e., functions that
can be defined without being named. Could do above example as:

        val w = map (fn x => x + 1, [1,2,3])
        val z = map (fn s => "(" ˆ s ˆ ")", ["a","bc",""], ["a","bc",""])

        In fact, the following declarations are identical (except that the
        second one isn’t recursive):

        fun foo x = e
        val foo = fn x => e
Nested procedure declarations

In Pascal, Ada, ML, etc., we can nest procedure declarations inside other procedure declarations. (Cannot do this in C,C++,Java!)

fun map (g:'a -> 'b, u: 'a list) : 'b list =
  let fun f (v :'a list) : 'b list =
    case v of
    | nil => nil
    | (h::t) => (g h)::(f t)
  in f u
end
val w = map (fn x => x + 1, [1,2,3])

Parameters and local variables of outer procedures are visible within inner procedures (using lexical scoping rules).

Purpose: localize scope of nested procedures, and avoid the need to pass auxiliary parameters defined in outer scopes.

Semantics of a function definition now depend on values of function’s free variables.

Key implementation question: what is the lifetime of the definition?

Semantics of Nested Procedures

Suppose procedure $p$ uses non-local variables? How are they found?

Semantically, it suffices to know the static environment surrounding the declaration of $p$ was encountered.

An interpreter can simply attach the current variable environment to its description of $p$ when it encounters $p$’s declaration and records it in the function environment.

When the interpreter applies $p$, it evaluates its body in an initial environment taken from the recorded description, which is then extended with $p$’s parameters and locals.

When the interpreter looks up a variable while executing $p$, it looks first among $p$’s locals and parameters, and then in the lexically-enclosing environment.

Here are appropriate dynamic semantic rules.

$$(	ext{fn } x \Rightarrow e, E) \Downarrow [x, e, E] \quad (\text{Fn})$$

$$
\begin{align*}
  (e_1, E) \Downarrow [x, e', E']  & \quad (e_2, E) \Downarrow v'  \\
  (e', E + \{ x \mapsto v' \}) \Downarrow v
\end{align*} \quad (\text{Appl})
$$

(The form of these rules implies that expression evaluation doesn’t change the environment; this is true, e.g., if we omit assignment from the language.)

Using Nested Procedures

- Sometimes want to pass nested functions as parameters.

  fun diffs(n:int, u:int list) =
    let fun diff (x:int) = x - n
    in map (diff,u)
  end
  ...
  diffs(3,[1,7,5]) (* yields [-2,4,2] *) ..

- Lexical scope rules apply, so function body can use data associated with outer function.

- Here $diff$ uses the value of $n$, which is a parameter of $diffs$.

- Cannot express this in C/C++/Java, which have no nested functions.

What if we wanted to compute $diffs$ on a several different lists with a fixed $n$?

- Want to create a function that represents $diffs$ specialized to a particular value of $n$.

- Solution: Write a function that returns another function!

"First-class" Procedures Example

fun diffs' (n:int) : int list -> int list =
  let fun d (u : int list) : int list =
    let fun diff (x:int) = x - n
    in map (diff,u)
  end
in d
end
val g : int list -> int list = diffs' 3
...
val x : int list = g [1,7,5] (* yields [-2,4,2] *)
val y : int list = g [2,4,6] (* yields [-1,1,3] *)
val z : int list = diffs' 3 [2,4,6]
  (* yields [-1,1,3] *)

ML also provides syntactic sugar to make such “Curried” functions easier to write. Above program is equivalent to:

fun diffs' (n:int) (u:int list) : int list =
  let fun diff (x:int) = x - n
  in map (diff,u)
end
Using Curried functions

- When defining “multi-argument” functions in ML, have a choice using a tuple argument and Currying.
- Can apply Curried version `diffs'` to either one or two arguments.
- Function application associates to the **left**, so

  \[
  \text{diffs' } 3 \ [2,4,6] = (\text{diffs' } 3) \ [2,4,6]
  \]

- Function type arrows associate to the **right**, so the type of `diffs'` is

  \[
  \text{int } \rightarrow \text{ int list } \rightarrow \text{ int list } = \\
  \text{int } \rightarrow \ (\text{int list } \rightarrow \text{ int list})
  \]

- Currying most often useful when passing partially applied functions to other higher-order functions, e.g.:

  \[
  \text{map (diffs' } 3, [[1,7,5],[2,4,6]])
  \]

  (* yields [[\text{~2,4,2}],[\text{~1,1,3}]] *)

- Note: unlike the map we’ve used here, the “built-in” definition of map in the SML standard library is itself defined as a Curried function, with type

  \[
  (\text{\textquotesingle a } \rightarrow \text{ \textquotesingle b}) \rightarrow \text{ \textquotesingle a list } \rightarrow \text{ \textquotesingle b list.}
  \]

Problems with first-class procedures

Consider activation tree for `diffs'` example:

```
main
  /
  /
  /
  /
  m\(\text{\textquotesingle d}(\text{\textquotesingle 3})\)\text{\textquotesingle g}(\text{\textquotesingle [2,4,6]})\)

map(d\(\text{\textquotesingle f}(\text{\textquotesingle [2,4,6]})\))
  |
  |
  |
  |
  (requires value \(n = 3\)) \text{\textquotesingle d}(\text{\textquotesingle 2})
```

Activation of `diffs'` is no longer live when `diff` is called!

If \(n\) is stored in activation record for `diffs'` and activation-record is stack-allocated, it will be gone at the point where `diff` needs it!

To avoid this problem:

- Pascal prohibits “upward funargs;” procedure values can only be passed downward, and can’t be stored.
- Some other languages only permit “top-level” procedures to be manipulated as procedure values (in C, this means **all** procedures!).

Heap Storage for Procedure Values

- Languages supporting first-class nested procedures (e.g., Lisp, Scheme, ML, Haskell, etc.) solve problem by using heap to store variables like \(n\).
- Simple solution: Just put all activation records in the heap to begin with! (Garbage collection is a must!)
- More refined solution: Represent procedure values by a heap-allocated **closure** record, containing the procedure’s code pointer and values of the **free** variables referenced by the procedure.
- Involves taking copies of the values of non-local variables, so only works when values are **immutable**.
- Can always introduce extra level of indirection to achieve this.
Using first-class functions

The ability to manipulate functions as first-class values is one of the hallmarks of a functional language.

Functional languages encourage sophisticated abstraction mechanisms. (Already saw use of map.)

Consider the following problems:

Sum a list of integers

\[
\text{fun sum \ l = } \\
\text{case l of } \\
\text{\ nil \Rightarrow 0 } \\
\text{\ h::t \Rightarrow h + (sum t)}
\]

Multiply a list of integers:

\[
\text{fun prod \ l = } \\
\text{case l of } \\
\text{\ nil \Rightarrow 1 } \\
\text{\ h::t \Rightarrow h * (prod t)}
\]

Folds

Copy a list (of anything):

\[
\text{fun copy \ l = } \\
\text{case l of } \\
\text{\ nil \Rightarrow nil } \\
\text{\ h::t \Rightarrow h::(copy t)}
\]

Query: How does copy differ from the identity function \(\text{fn \ x \Rightarrow x}\)?

Calculate the length of a list (of anything):

\[
\text{fun len \ l = } \\
\text{case l of } \\
\text{\ nil \Rightarrow 0 } \\
\text{\ h::t \Rightarrow 1 + (len t)}
\]

Folds (continued)

We can abstract over the common inductive pattern displayed by these examples:

\[
\text{fun foldr \ f \ n \ l = } \\
\text{case l of } \\
\text{\ nil \Rightarrow n } \\
\text{\ h::t \Rightarrow f(h,foldr \ f \ n \ t)}
\]

\[
\begin{align*}
\text{fun prod \ l = foldr \ (op *) \ 1 \ l} \\
\text{fun copy \ l = foldr \ (op ::) \ nil \ l} \\
\text{fun len \ l = foldr \ (fn \ (_,y) \Rightarrow 1+y) \ 0 \ l}
\end{align*}
\]

Function \(\text{foldr}\) computes a value working from the tail of the list to the head (from right to left). Argument \(n\) is the value to return for the nil list. Argument \(f\) is the function to apply to each element and the previously computed result.

Can view \(\text{foldr \ f \ n \ l}\) as replacing each \(::\) constructor in \(l\) with \(f\) and the nil constructor with \(n\). For example:

\[
\begin{align*}
\text{l = x1 :: (x2 :: (... :: (xn :: nil))}) \\
\text{foldr \ (op *) \ 1 \ l = } \\
\text{x1 * (x2 * (... (xn * 1)))}
\end{align*}
\]

Tail-calls Revisited

Consider the \(\text{len}\) function again:

\[
\text{fun len \ (h::t) = len \ t + 1 } \\
\text{| len nil = 0}
\]

Function \(\text{len}\) is recursive, but not tail-recursive. But we can convert \(\text{len}\) into a tail-recursive function, as follows:

\[
\begin{align*}
\text{fun len \ l = } \\
\text{let fun \ len’ \ (h::t,k) = } \\
\text{\ len’(t, fn \ i \Rightarrow k \ (i+1)) } \\
\text{\ | len’(nil,k) = k 0 } \\
\text{\ fun id \ i = i } \\
\text{\ in \ len’ \ (l, id) } \\
\text{end}
\end{align*}
\]

This rather odd code was constructed by giving \(\text{len’}\) an additional argument, \(k\), of type \(\text{int} \rightarrow \text{int}\). Instead of returning its “result” value, \(\text{len’}\) passes it to \(k\).

Notice that every call in \(\text{len’}\) is a tail-call. Note too that \(\text{len’}\) only returns after \(\text{id}\) is invoked. If it were the whole program, it wouldn’t need to return at all. This is because \(k\) is serving the same role as a return address: saying “what to do next.”
Comparison of Computations

len ["a", "b"] →
len ["b"] + 1 →
(len []) + 1 + 1 →
(0 + 1) + 1 →
1 + 1 →
2

fun len l =
  let fun len' (h::t, k) =
    | len' (nil, k) = k 0
  fun id i = i
  in len' (l, id)
  end

len ["a", "b"] →
len'(["a", "b"], id) →
len'(["b"], fn i1 => id(i1 + 1)) →
len'([], fn i2 => (fn i1 => id(i1 + 1)) (i2 + 1)) →
(fn i2 => (fn i1 => id(i1 + 1))(i2 + 1)) 0 →
(fn i1 => id(i1 + 1))(0 + 1) →
(fn i1 => id(i1 + 1))(1 + 1) 1 →
id(1 + 1) →
id 2 →
2

Continuation-passing Style

Functions like k are called continuations and programs written using them are said to be in continuation-passing style (CPS).

We may choose to write (parts or all of) programs explicitly in CPS because it makes it easy to express a particular algorithm or because it clarifies the control structure of the program.

Note that CPS programs are just a subset of ordinary functional programs that happens to make heavy use of the (existing) enormous power of first-class functions.

Remarkably, we can also systematically convert any functional program into an equivalent CPS program. (Details omitted.)

More broadly speaking, the term continuation means any representation of the program’s state at a particular point during execution. This state must contain exactly the information needed to “continue” execution of the program from that point until the program completes.

CPS programs represent each continuation as a function, which, given the value of the expression being computed at that point, returns the result of the entire program.

If we are describing execution in terms of a low-level machine, we can think of a continuation for a program point as the machine state at that point: the values of the pc, return stack, etc. These contain the same information that would go into a closure for the continuation function.