CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Closures

Free Variables

One aspect of our λ-calculus intermediate language that is quite far from what is available on a "real machine" is its notion of **function**.

Our functions definitions can be **nested**, which means that they can have **free variables**. Also, functions can be returned or stored as "first-class" values.

Any curried function provides an example, e.g.:

```
let f = λx.
          let h = λy. 2*x+3*y
          in h
in let g = f 3
   in g 2
```

Real machines represent functions by a machine-code **address**. Arguments are handled in registers or stack according to some calling convention, with the assumption that the location of each argument is **static** (possibly relative to a stack pointer). Nothing in the machine model corresponds to (non-global) free variables.

Our solution is simply to transform each λ-calculus function to take the **values** of its free variables as **additional arguments**. The function's callers are then responsible for providing the correct values when the function is invoked. These **closed** function definitions can then be lifted to top level, since they no longer depend on their surrounding context.

Closures

Although a function's free variables are obviously in scope at the point where the function is defined (i.e., where the λ-abstraction is evaluated) they generally are **not** in scope at the point of call! In fact, the caller generally doesn't even know the names or number of the callee's free variables.

In our example, `g` is bound to a function whose body references the free variable `x`, yet `x` is not in scope at the point where `f` is called.

Thus, to represent a functional value, we need to package up the function's code pointer together with the **values** of its free variables at the point where the function is defined. This package is traditionally called a **closure**. Then, whenever we say that a "function" is being passed or returned or stored in a data structure, we really mean its closure.

Since this technique involves making **copies** of the free variables' values, it only works when these variables are **immutable**. This is true in pure languages, and also in languages like ML, where only data structures (not variables) can be modified.

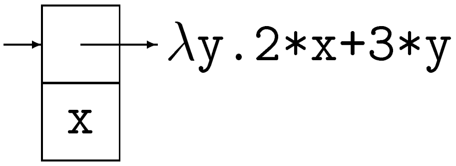
Java has inner (anonymous) classes, but their only free identifiers are outer class fields (not variables) or `final` (i.e., immutable) outer-class method local variables.

More later on other languages.

Closure Representation

A simple closure representation is just a record whose first element is a code pointer, and whose remaining elements are the values of the function's free variables.

For example, a closure for `h` might look like this:



To **call** a function represented by a closure record, the program must:

• **Extract** the code pointer from the first slot in the record.
• **Jump** to the code, passing the **closure itself** as the first argument to the function, and the ordinary arguments afterwards.

The function body thus has access to its (formerly) free variables by dereferencing its first argument.

Closures vary in size (according to how many free variables they contain), but we want to be able to manipulate them uniformly. So we actually allocate closures as records (typically on the heap) and pass pointers to them.

Example

Here's our previous example again, with the inner λ-expression named for clarity.

```
let f = λx.
          let h = λy. 2*x+3*y
          in h
in let g = f 3
   in g 2
```

Since `h` has a free variable (`x`), it must be represented by a closure record, consisting of a code pointer and the value of the free variable. The definition and uses of `h` must be altered accordingly.

We can show the result in the λ-calculus IR itself (assuming tuples are available). Note that "code pointers" are represented by λ-abstractions that have no free variables, and can be lifted to top level.

```
let h' = λ(c,y).
           let x' = #2 c
           in 2*x'+3*y
in let f = λx. (h',x)
   in let g = f 3
      in (#1 g)(g,2)
```

(To be consistent, we should represent `f` as a closure too, even though it has no free variables.)

Closure Conversion

The process of introducing explicit closure records and the corresponding code to create and select from them is called **closure conversion**. In general, it is a **global** transformation acting on all `Abs`, `Fix`, and `App` nodes:

- Transform the formal argument list of each `Abs` to include the closure argument.

- Transform the body of each `Abs`, replacing references to free variables with `Select`s from the closure parameter.

- Replace each `Abs` by a `Record` whose first field is an `Abs` for the transformed function and whose other fields are the values of (the original function's) free variables. (Similarly for `Fix`.)

- Modify each `App` to extract the first field of the "operator" expression (now record-valued) and apply it to the closure record **and** the original arguments.

- Lift all functions to top level (naming any that are anonymous).

Note that by this process, every identifier that represents a function in the original program comes to represent a closure record in the converted program.

Typing Problems

This approach works fine if we treat our intermediate languages as untyped. But if fails if we try to type the IR, treating closures as ordinary records.

The problem is that the structural type of a closure record depends on the number and kind of the function's **free variables** rather than on the function's own type. Thus source expressions with the same functional type may be transformed into closure-converted expressions with conflicting types.

For example:

```
let f = if b then λx.x+y else λx.x+y+z
in f 10
```

becomes

```
let f = if b then (λ(c,x).x+(#2 c),y)
             else (λ(c,x).x+(#2 c)+(#3 c),y,z)
in (#1 f)(f,10)
```

Now `f` has no consistently defined type (is it a pair or a triple?)

Typing Solutions

There are at least three solutions to this problem:

• Treat closures as a special sort of **abstract** record type, and the extraction and application of the function field as a special abstract operation. Ordinary `Select` operations are not permitted on closure records, so their internal structure doesn't matter.

• Use **existential types** to make the structure of the record visible, but require the extracted function to be applied to the exact same closure record from which it was extracted.

• Encode closures as elements of a **variant** type, with a distinct label for each original λ-expression. Then, explicitly `Switch` on the variant label to invoke the appropriate function.

All approaches have problems. The first makes it impossible to subject the extraction and application to ordinary optimizations. The second is complex and also fails to support some optimizations. The third requires that the **whole program** is known (at least by the time the `Switch` code is constructed), and may require that the program be **monomorphic**.

Note the similarities to our problems with typing OO languages!

More details on these approaches later.

Avoiding Closures

There are a number of circumstances under which closure construction is not needed.

If a function is **known**, i.e., we can identify all its call sites at the same time as we process its definition, **both** caller and callee can know about the closure representation.

In fact, in this case there's no need to build a closure at all! Since all call sites know the name and number of the function's free variables, we can just make each free variable a separate new parameter to the (transformed) function.

For example

```
let p = λx.let q = λy.2*x+3*y in q 2
in p 3
```

becomes

```
let q = λ(x',y).2*x'+3*y
in let p = λx.q(x,2)
   in p 3
```

Notice that the free variable set of the **calling** function is effectively augmented by the free variables of the **called** function, so long lists of arguments may cascade upwards through nested functions.

Local Tail Functions

Altough it is always possible to lift all known functions to top level, it is not always desirable.

Consider a known function f defined locally to function g and always called in **tail position** in g: i.e., after f returns to g, g itself immediately returns. Then there's no need to save g's context (local variables or PC) when executing f, since they won't be needed again.

Thus, calls to f can be implemented at machine level by some register updates (to set up f's arguments) followed by a simple jump. When f terminates, it sets a return value in a designated register and executes a normal machine `return`, which actually returns from g (with that value). This is typically much cheaper than doing a full machine function call.

Such **tail calls** are not common in user code, but they often arise as a result of optimization transformations on intermediate code, so are worth catering for.

The same principle can be applied to recursive functions, to convert "tail-recursion," which **is** very common, into iteration.

Allocation strategies

In general, if we need a closure, it must be heap-allocated, since it has arbitrary size and arbitrarily long lifetimes.

However, it is sometimes possible to allocate closures on the **stack** at the point where they are defined. Intuitively, this is possible iff the closure cannot be invoked after the procedure in which it is defined returns.

Example

```
let p = λ(f,x).f(x+1) in p (λz.z-1,3)
```

It is obviously safe to discard such a closure from the stack when the defining procedure returns. Such a function is said to **escape (only) downward**; a precise definition is slightly tricky. Moreover, a compiler cannot guarantee to locate all such functions in general.

More on Representation

Note that the **caller** knows nothing about the format of the closure record except for where to find the code pointer. (This is essential!) Thus we can be flexible about the number and positioning of the free variable slots are not important, so long as the function **body** knows what they are.

In fact, we can use any closure representation we like, so long as the callee understands it. Putting all the free variables in a single record is called a **flat** closure. It guarantees unit access time to each variable. But in general, the same value may appear in many closures, occupying lots of space.

Example

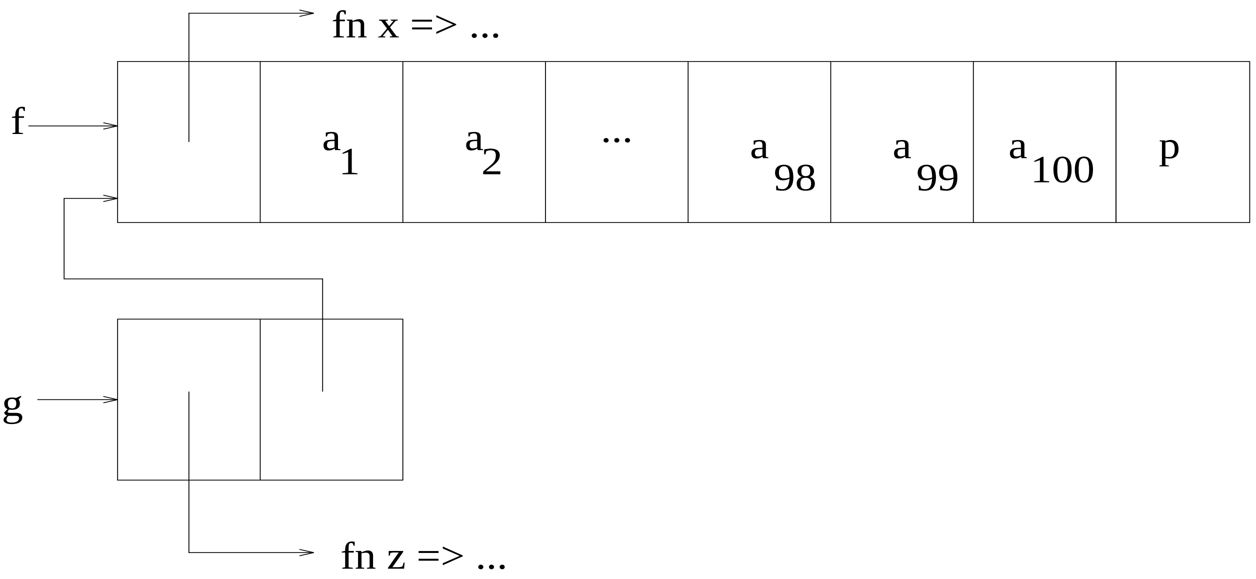
```
let a₁ = ...
    a₂ = ...
    ...
    a₁₀₀ = ...
in let f = λx.
              let g = λz.z+a₂+a₃+...+a₁₀₀
                   in p(a₁); g
in ...
```

Here the a_i are free in `g` and hence also in `f`; using flat closures, we must copy them all into the heap once when creating the closure for `f` and again when doing so for `g`.

Linked Closures

We may therefore prefer to use a **linked** closure representation, in which each closure contains only the free variables unique to its function, plus a pointer to the closure of its lexically-enclosing function. This minimizes space at the expense of longer access times.

Example again



Various tricks can be used to shorten access times to variables far up on the chain of closures.

Note that because the lifetime of the nested function (here `g`) may exceed that of the enclosing function (`f`), we may not be able to deallocate `f`'s closure as soon as we might expect or wish.

For example, suppose the value of a_1 is a very large object. It is not referenced by `g`, but it is indirectly accessible from `g`'s closure, so will remain live as long as `g` is. In general, linked closures are not **safe for space**.

Comparison with Traditional Languages

Most traditional imperative languages have an implementation model based on stack-allocated **activation records**. Each record contains the values of the arguments and local variables for a single function activation. (Real implementations may use other techniques, but they must respect this model.)

Arguments and locals in the current activation record can be accessed at known offsets from the current **stack** (or **frame**) **pointer**. Global variables live at fixed locations. But how can we access non-local (i.e., free) variables? In particular, what if we have functions as values?

This is not an issue in assembly language and C. Since these languages do not support statically nested function definitions, functions can refer only to local and (completely) global variables. Thus, even though these languages support functions as values in the form of code pointers, there is no need for anything like closures. (Note that our closure-converted programs are now suitable for direct translation to such languages!)

However, Pascal-like languages **do** support lexically-nested functions, which can access non-local variables. This requires **linking** together the activation records corresponding to lexically-enclosing functions.

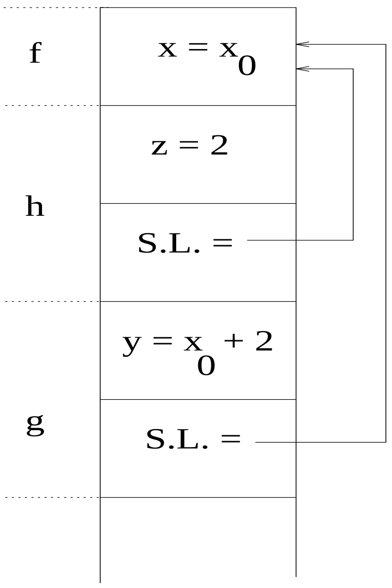
Access links

The standard method is for each activation record to contain an **access link** pointing to the activation record of the **lexically-enclosing** procedure.

Example

```
PROCEDURE f(INT x);
  PROCEDURE g (INT y);
     BEGIN RETURN(x+y) END g;
  PROCEDURE h (INT z);
     BEGIN g(x+z) END h;
BEGIN h(2) END f;
```

Stack, just before `RETURN`:



Other data structures for non-local access, such as **displays**, are essentially variations on access links.

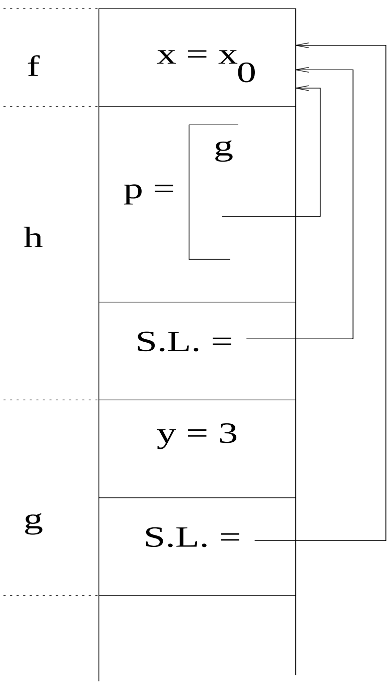
Functions as Values

Pascal-like languages sometimes allow functions to be passed as arguments to other functions. To allow such a function to access its non-local variables, the function's access link is passed along with its code pointer.

Example

```
PROCEDURE f(INT x);
  PROCEDURE g(INT y);
    BEGIN RETURN(x+y) END g;
  PROCEDURE h(PROCEDURE p(INT):INT);
    BEGIN p(3) END h;
BEGIN h(g) END f;
```

Stack, just before `RETURN`:



Access Links and Closures

Evidently, a (function pointer, access link) pair acts essentially like a **linked closure**, whose environment components happen to be stored on the stack rather than on the heap. This works because Pascal-like languages are careful **not** to allow functions to be returned or stored, i.e., they aren't "first-class" values. Thus any activation record containing a non-local variable referenced by a functional parameter will still be on the stack when the function is called.

(The Pascal runtime scheme differs from our closures—and more closely resembles our original treatment of **environments**—in that it allows access to **all** in-scope variables, not just those that are actually referenced by the code, and doesn't distinguish between locally bound and free variables.)

To support first-class functions in an Pascal-like language, it would suffice simply to allocate activation records in the heap rather than on the stack!

In languages like Scheme, **every** local variable is potentially mutable. This is fine if we use linked closures, since a given variable appears in exactly one closure. But what if we want to use flat closures?

Assignment Conversion

A standard solution is **assignment conversion**, in which each potentially mutable variable is replaced by a **pointer** to a heap cell (i.e., an ml `ref` cell).

```
(lambda (x)
   ... x ...
    ... (set! x value) ...)
```

becomes

```
(lambda (x)
   (let ((x' (make-cell x)))
        ... (contents x' ) ...
        ... (set-contents x' value)...))
```

Note that `x'` will typically end up as a field in some heap-allocated closure, which in turn points to a one-word heap cell. If `x'` appears in only one closure and is never passed as a value in its own right, the extra level of indirection can be **collapsed** as an optimization.

Free variables (i.e., those whose definitions are not available at compile time) must be assumed mutable, but they seldom actually are. Instead of doing assignment conversion on them, we can have the runtime system maintain an explicit list of all their copies, which is then used to make sure updates are globally performed.