

Closure Conversion and Defunctionalization

CS 502

Lecture 7

9/25/08

Goals

- Translate high-level abstractions
 - Procedures
 - Data structures

into a form suitable for execution on a machine.

- Expose opportunities for further optimizations.
-

Choices

- Direct translation to native code.
- Target customized virtual machine.
 - Java Virtual Machine: provides direct support for Java specific features (e.g., method invocation)
- Interpretation.

Tradeoffs:

- Native-code translation provides greater efficiency, but is more complex to implement.
 - Interpretation is easier to implement (many details can be hidden), but not particularly efficient.
 - Virtual machine approach offers a good middle ground, but requires designing a machine representation that captures essential features.
-

Other Considerations

- Interaction with runtime system:
 - Memory management (garbage collection)
 - Foreign function interfaces
 - Concurrency, distributed programming
 - Data type layout
 - How should objects be represented?
 - What about type checks, casts?
 - Procedure calling conventions
 - Continuations
-

Issues

- Implementation of static environment
 - How do we access free variables of a procedure?
 - Implementation of dynamic environment
 - How do we propagate values of actual parameters to formals?
 - How do we propagate return value to caller?
 - How do we match exceptions to handlers?
 - Implementation of tail vs. non-tail calls.
 - Implementation of dynamic data structures
 - How do we organize the heap
 - Object layout
 - Role of registers and stacks.
-

Where we are

- Static analysis for type checking
 - CPS translation to manifest control-flow
 - All temporaries named
 - Still need:
 - Optimizations to improve performance and space overheads
 - Decisions on object layout and allocation
 - Translation to simpler first-order language:
 - No higher-order functions
-

Code Generation

- Specify translation using so-called “two-level language”
 - Distinguish between parts that can be done at compile-time
 - Computing offsets of free variables
 - From parts to be executed at runtime
 - Actual code sequence
 - Memory allocation
 - Distinguish between structures manipulated by the compiler and those manipulated by the runtime system
 - Environments that associate continuation variables with offsets on a frame
 - Stack frames and heaps
-

Data Layout

- What information should heap-allocated objects record?
 - Vectors: length of vector
 - Closures: label of beginning of code sequence
 - Constructors: identity
 - In a realistic compiler, would require more information:
 - Where are pointers stored? Important for garbage collection.
 - Must also add a pointer object that distinguishes references to heap allocated object from the objects themselves.
-

Before we address these issues...

- Want a simple language to perform optimizations
 - inlining, constant propagation, common subexpression elimination, PRE, ...
 - what is the best intermediate representation in which to perform them?
 - First-order language (no first-class procedures)
 - Even optimizations on continuations can benefit

Variable Accesses

- How do we efficiently lookup free variables found within a procedure?

$\text{fn } (x) \Rightarrow \text{fn } (y) \Rightarrow x + y$

- When we evaluate the inner function, how do we determine where x is located?
 - Note: the “activation frame” for the outer function is no longer available.

Static and Dynamic Environments

- A static environment computes the layout of objects statically.
- Most importantly, the static environment identifies the location of free variables occurring in procedures:

$\text{fn } (x) \Rightarrow \text{fn } (y) \Rightarrow x + y$

$\text{fn} \Rightarrow \text{fn} \Rightarrow [1] + [0]$

The offsets [0] and [1] indicate where to find y and x resp. at runtime. A function's **closure** is the runtime representation of a static environment.

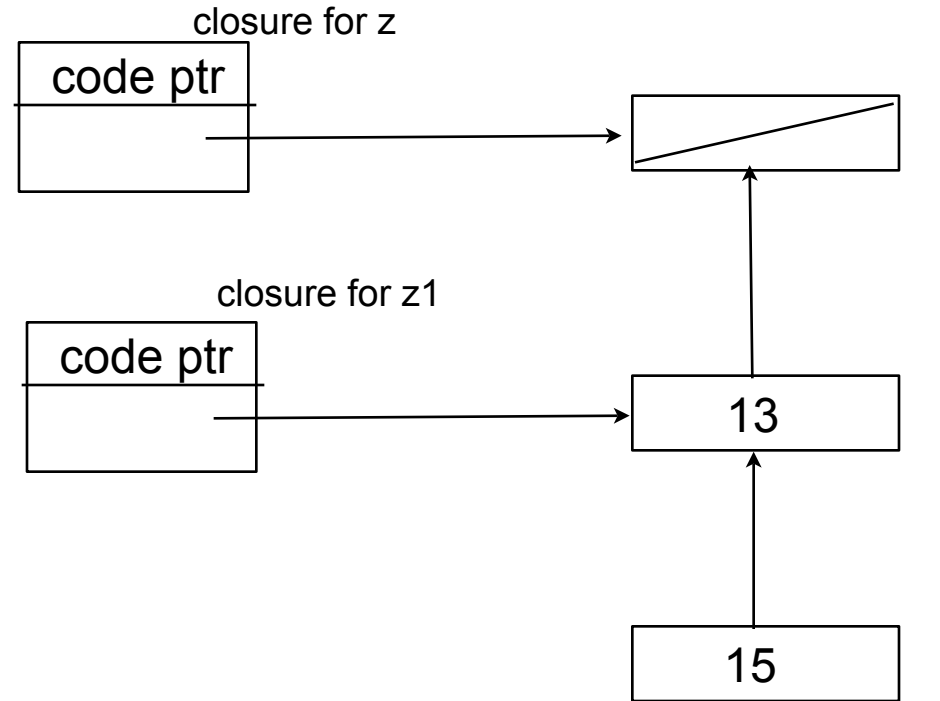
Representation

val z = fn => fn => [1] + [0]

val z1 = z(13)

val z2 = z1(15)

*offset 0 refers to 15, and
offset 1 refers to 13 based on
the structure of the environment*



Strategy

- When we evaluate a function, we build a closure that captures the current environment.
- When we perform a call, we augment the environment to include the value of argument.
 - A non-tail call also stores the current environment as part of the dynamic state
- When we return from a call, we restore the current environment from the stored dynamic state

Closure Representation

- A closure is a pair consisting of a pointer to the code representing the function and
 - a data structure used to find the values of free variables in the code
 - First-order representation of a higher-order procedure
 - How should closures be represented?
 - sharing of closure records
 - "safe-for-space"
-

Defunctionalization

- First-class functions represented using first-order datatypes:
 - a function is introduced with a constructor that holds the value of its free variables.
 - Accessed via a case dispatch over the appropriate constructor.
- Closure conversion = defunctionalization + inlining

Example

```
fun aux f = f 1 + f 10
```

```
fun main (x,y,b) =  
  aux (fn z => x + z) *  
  aux (fn z => if b then y + z else y - z)
```

Procedures aux and main have no free variables:
could be represented by a function pointer.

The two abstractions are associated with a datatype that has two constructors, one for each function abstraction.

Each constructor is closed over the environment of the associated procedure:

fn z => x + z : Environment holds x

fn z => if b then y else y - z : Environment holds b and y

Example

After defunctionalization:

```
datatype lam = Lam1 of int | Lam2 of int * bool
```

```
fun apply (Lam1 x, z) = x + z  
  | apply (Lam2 (y,b), z) = if b then y + z else y - z
```

```
(* aux: lam -> int *)
```

```
fun aux f = apply(f,1) * apply(f,10)
```

```
fun main(x,y,b) =  
  aux(Lam1 x) * aux(Lam2(y,b))
```

Example

```
fun aux(i,f) = f(i)
```

```
fun main(i,js) =  
  let fun walk nil = nil  
      | walk (j::js) = aux(i,fn i => i + j) :: (walk js)  
  in walk js  
  end
```

Given a number i and list of numbers $[j_1, \dots, j_n]$
returns $[j_1 + i, \dots, j_n + i]$.

Program creates n closures, n dynamic procedure
objects.

```
datatype lam = LAM of int
```

```
fun apply(LAM j,i) = i + j
```

```
fun aux(i,f) = apply(f,i)
```

```
fun main(i,js) =  
  let fun walk nil = nil  
      | walk(j::js) =  
        aux(i,LAM j):: (walk js)  
  in walk(js)  
  end
```

Closure Conversion

```
fun f (v,w,x,y,z) =  
  let fun g() =  
        let val u = hd(v)  
            fun h =  
              let fun i() =  
                    w + x + y + z + 3  
                  in (i,u)  
                  end  
              in h  
              end  
        in g  
        end
```

```
let big n = <a big list>  
fun loop(n,res) =  
  if n < 1 then res  
  else  
    (let val s = f(big(N),0,0,0,0)()  
     in loop(n-1,s::res)  
     end)  
val result = loop(N,[])
```

Closure Representation Choices

□ Flat closures:

- Each evaluation of $f(\dots)()$ yields a closure s for h that contains just a few integers u, w, x, y, z
- The final result contains N copies of the closure s for h : uses $O(N^2)$ space

G

g	v	w	x	y	z
-----	-----	-----	-----	-----	-----

H

h	u	w	x	y	z
-----	-----	-----	-----	-----	-----

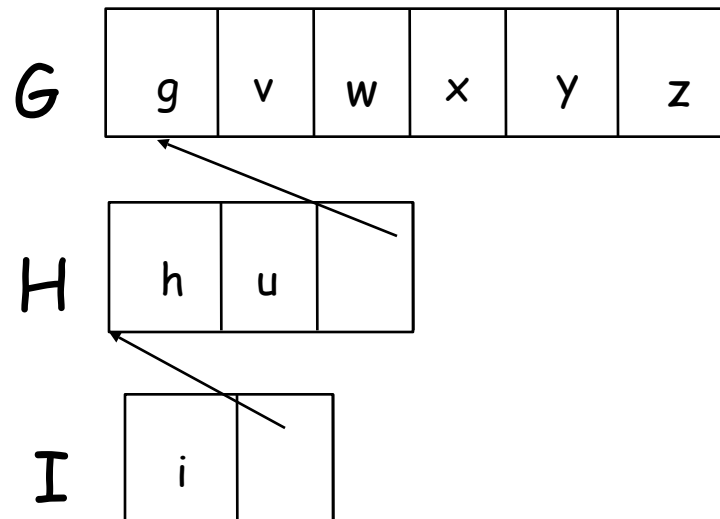
I

i	w	x	y	z
-----	-----	-----	-----	-----

Closure Representation Choices

□ Linked Closures

- Each closure s for h contains a pointer to the closure for g which contains a pointer to a list v of size N . Requires $O(N)$ space.



Tradeoffs

- Flat closures are “safe-for-space”:
 - Any local variable binding must be unreachable after its last use within its scope.
 - However, bindings may be copied many times from one closure to another.
 - Linked closures are not safe-for-space:
 - local variable bindings will stay on the stack until they exit their scope, so they may remain live even after their last use.
 - However, the same variable is not copied, and recorded only once.
-