Compiling Functional Languages

e.g. Lisp, Scheme, ML, Haskell, Miranda

Lisp and Scheme: dynamic typing
⇒ uniform "boxed" representation of all data objects, tagged pointers to encode some types (e.g. ints) cheaper

ML, Haskell, Miranda: polymorphic static typing
⇒ uniform "boxed" representation....

First-class, lexically-nested functions

- static scoping of nested functions
  ⇒ closures to represent function values
- functions can outlive defining scope
  ⇒ heap-allocated environments
- calls of computed expressions
  ⇒ (fancier) call graph analysis

Heavy use of recursion instead of iteration
⇒ tail call, tail recursion elimination

Immutable update-by-copy data structures
⇒ version arrays, compile-time reference counting

Miranda & Haskell: lazy evaluation
⇒ strictness analysis

Implementing higher-order functions

Functions are first-class data values

- passed as arguments, returned from fns, stored in data structures
- potentially anonymous
- lexically-scoped

Example:

```
(define mul-by (lambda lst n)
  (map (lambda (x) (* x n)) lst))
```

2 components of a function value (a **closure**):

- code pointer
- lexically-enclosing environment pointer

Steps in deciding how to implement a closure:

- **strategy analysis**: where to allocate closure
- **representation analysis**: how to lay out data structure

Strategy analysis

Option 1: heap allocation

+ most general option
+ simple decision to make

– expensive to create, invoke, and reclaim closure
– may require heap-allocation of lexically-enclosing env

Supports "upward funargs"

Example:

```
(define add (lambda x)
              (lambda (y)
                 (+ x y)))
(define inc (add 1))
(define dec (add -1))
(print (inc (dec 3)))
```


Stack allocation

Option 2: stack allocation

If closure's dynamic extent is contained within the extent of its lexically-enclosing activation record, then can allocate closure as part of a.r.'s stack frame (a LIFO closure)

+ faster allocation, free reclamation
+ enclosing environment can be stack-allocated

– invocation still slow
– restricted applicability

Inlining calls to closures

Option 3: represent closure in-line

If invoking a known closure, inline-expand body
If all uses of known closure inlined away, don't create closure
- closure's environment turns into local variables

+ free allocation, fast invocation, free reclamation
– limited applicability

Enables closure-based user-defined control structures

Determining LIFO extent

How to determine if closure (or other data structure) has LIFO extent?

Track flow of value, see where it goes

Has LIFO extent:
- when created
- when assigned to local variable
- when invoked

Hard case:
- passed as argument to function
  - if intraprocedural analysis: escapes
  - if interprocedural analysis: may or may not escape

Harder cases:
- returned
- stored in global/non-local variable or data structure

(Conservatively) assume escapes

Can do similar LIFO analysis for other allocated data structures

Interprocedural escape analysis

Compute for each formal parameter whether that parameter can **escape**

Construct program's call graph
Initialize all formals to "does not escape"
Initialize worklist to empty set

Process each function:
  if formal parameter labeled "does not escape"
  escapes locally within this function,
  change formal to "escapes" and put all callers on worklist

While worklist non-empty:
  remove function from worklist, reprocess
- at call site, actual escapes if corresponding formal escapes

Representation analysis

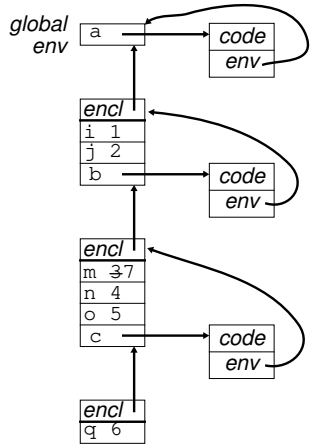
How to represent closure's lexical environment?

Option 1: **deep binding**
- represent environment as pointer to shared env record

+ simple
+ space-efficient
+ fast to create closure
– slow to access lexically-enclosing vars

Example

```
(define a (lambda (i j)
  (define b (lambda m n o)
    (define c (lambda q)
      (+ q m i)) ;; here
    (set! m 7)
    (c 6))
  (b 3 4 5))
(a 1 2)
```



Representation analysis, cont

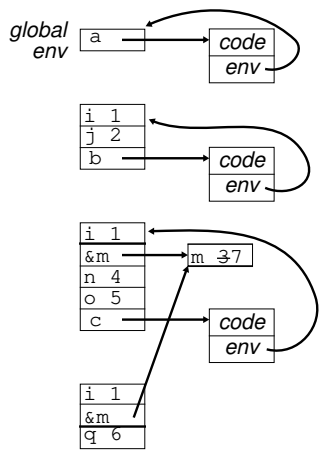
Option 2: **shallow binding**

- copy needed values into environment when created

+ fast access to enclosing vars
+ may not need to heap-allocate enclosing environment
– slower closure creation
– more space consuming, if >1 var needed
– doesn't work for mutable vars
  - apply assignment conversion first

Example

```
(define a (lambda (i j)
  (define b (lambda m n o)
    (define c (lambda q)
      (+ q m i)) ;; here
    (set! m 7)
    (c 6))
  (b 3 4 5))
(a 1 2)
```



Representation analysis, cont

Option 3: **very shallow binding**

- copy needed values into closure itself

Like shallow binding, except:

+ even faster access to enclosing vars
– even slower closure creation, if >1 var needed
– even more space consuming, if >1 var or >1 closure needed