Compiling Functional Languages

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

Lisp and Scheme: dynamic typing
⇒ tagging scheme in run-time system

First-class, higher-order functions

- static scoping of nested functions ⇒ closures
- 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

Functional data structures
⇒ in-place updates (e.g., 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 functions, stored in data structures
- potentially anonymous
- lexically-scoped

Example:

```
(define (add_n n)
   (lambda (x) (+ x n)))
(set add3 (add_n 3)
```

An issue: access of a function to its free variables which are defined in an enclosing scope (*e.g.*, `n`)

Function values represented with **closures**.

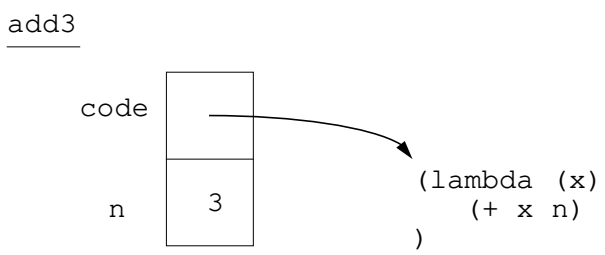
Implementing a closure:

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

Example of a closure

Consider this code again:

```
(define (add_n n)
   (lambda (x) (+ x n)))
(set add3 (add_n 3))
```



In the piece of code for `add3` :

- `x` is an argument ⇒ location determined by calling conventions
- `n` is defined in the lexically-enclosing environment

Two components of a closure:

- code pointer
- lexically-enclosing environment pointer

Strategy analysis
Option 1: Heap Allocation

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

Strategy analysis
Option 2: Inlining Calls to Closures

Represent closure in-line:

if all invocations of a closure are known,
and there are no other uses of the closure,
then can inline-expand invocations

- if all remaining uses gone, don't create closure at all
- still need to be careful with nested closures

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

Enables closure-based user-defined control structures

Strategy analysis
Option 3: Stack Allocation

Stack allocation: when cannot inline the closure, but do not need full generality of heap 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

Determining LIFO extent

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

Escape analysis: track flow of value, see where it goes

Has LIFO extent:

- when created
- when assigned to local variable
- when invoked

Doesn't when:

- returned
- stored in global variable or data structure

Hard case:

- passed as argument to function
  $\Rightarrow$ interprocedural analysis

Can do similar 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 all functions

While worklist non-empty:

- remove and process a function from worklist:
  if formal parameter labeled "does not escape" escapes within this function,
  then change formal to "escapes" and put all callers on worklist

Example

```
(define (a)
  (b (lambda () ...)
     (lambda () ...)
     (lambda () ...)))

(define (b b_1 b_2 b_3)
  (c b_2)
  (b b_2 b_1 (lambda () ...))
  (d b_3))

(define (c c_1)
  (set glob_fun c_1))

(define (d d_1)
  (d_1)
  (glob_fun))
```


Analyzing the example

Representation analysis

How to represent closure's lexical environment?

- closures are used to represent data structures

Option 1: **shallow binding**

- copy needed values into closure 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

Example:

```
(define (a i j)
  (define (b m n o)
    (define (c q)
      (+ q m i)
      (set n 3) ; ??
) ) )
```


Representation analysis, cont

Option 2: **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 i j)
  (define (b m n o)
    (define (c q)
      (+ q m i)
      (set n 3) ; OK
) ) )
```

Tradeoffs:

- time to create a closure, vs.
- time to access a variable

Also, space consumption, GC