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 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 x)
  (lambda (y) (+ x y)))
(set inc (add 1))
(set 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

(Conservatively) doesn't when:

- returned
- stored in global variable or data structure

Hard case:

- passed as argument to function

Intraprocedural analysis: escapes

Interprocedural analysis: may or may not escape

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

Representation analysis, cont

Option 2: **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
  - apply assignment conversion first

Example:

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


Control flow analysis for higher-order programs

If have first-class functions, then calls can invoke computed fns

- hard to build call graph, do interprocedural analysis

Could make worst-case assumptions

- not effective if heavy use of computed fns

Better: do interprocedural analysis,
compute set of possibly-called functions at each call site

Circularity dilemma

Problem:

- to do interprocedural analysis, need a call graph
- to construct a call graph, need to know possible values of called expressions
- to know values of expressions, need to do analysis
- ...

How to break vicious cycle?

A novel solution: optimistic iterative analysis

Set up a standard optimistic interprocedural analysis,
use iteration to relax initial optimistic solution into
a sound fixpoint solution

Simple flow-insensitive, context-insensitive analysis:

- for each (formal, local, global, instance) variable, maintain set of possible functions that could be there
  - one abstract fn for each source-level lambda, not closure
  - initially: empty set for all variables
- for each call site, set of callees derived from set associated with applied function expression
  - initially: no callees ⇒ empty call graph

worklist := {main}
while worklist not empty
    remove p from worklist
    process p:
        perform intra analysis propagating fn sets from formals
        at call site s in p, add fns of actuals to callees' formals
            if new callee possible or formals changed,
                put callee back on worklist

Example

```
(define (a)
  (b (lambda1 (f1) (f1 d))))

(define (b fb)
  (c (lambda2 (f2) (d d)))
  (fb d))

(define (c fc)
  (fc fc))

(define (d fd)
  (c (lambda3 (f3) fd)))

(a)
```