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
⇒ 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

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

Doesn't when:
- returned
- stored in global variable or data structure

Hard case:
- passed as argument to function

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 empty set

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

While worklist non-empty:
remove function from worklist, reprocess

Example

```
(define g (lambda () ...))

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

(define (b f1 f2 f3)
  (c f3)
  (b f2 f1 (lambda () ...))
  (d f2))

(define (c f4)
  (f4)
  (g))

(define (d f5)
  (set g f5))
```


Inlining calls to closures

Option 3: 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

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

Enables closure-based user-defined control structures

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

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

Example

```
(define (a i j)
  (define (b m n o)
    (define (c q)
      (+ q m i))
    (c ...))
  (b ...))
(a ...)
```


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

Alternative [Shivers 88]:

- do analysis to determine possible values of called exprs

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?

An algorithm

The task:
for each expression in the program,
compute the set of functions
that the expression might evaluate to

Analysis methodology: **abstract interpretation**

- define denotational semantics (interpreter) for language
- compute abstract property, not real value

A simple language

E ::= *const*
| *var*
| *primop*
| **(lambda (**var_{formal}*__)__ E_{body}**)**
| **(if** E_{test} E_{then} E_{else}**)**
| **(letrec ((**var_{name} E_{value}**)*)** E_{body}**)**
| **(**E_{fn} E_{arg}*__)__

Constants: `3`, `4.5`, `"hi"`, `'foo`, `'(a list of length 5)`
Primops: `+`, `eq`, `cons`, `car`, `cdr`, `...`

The abstract semantics

The domains:
Expr: domain of expressions in program
Var: domain of variables in program (assumed unique)
Fn: domain of functions in program (includes primops)

The top-level interface:
Fns: 2^{Fn}
Prog: Expr → Fns

Computing result fns of expressions

Prog: Expr → Fns

Prog [E] ≡ F [E] ∅

F: Expr × Env → Fns
Env: Var → Fns

F [*const*] ρ ≡ {}
F [*var*] ρ ≡ ρ(*var*)
F [*primop*] ρ ≡ {*primop*}
F [`(lambda (...) ...)`] ρ ≡ {`(lambda (...) ...)`}

F [`(if` E_1 E_2 E_3`)`] ρ ≡ F [E_2] ρ; F [E_2] ρ ∪ F [E_3] ρ

F [`(letrec ((`var_1 E_1`) ... (`var_n E_n`))` E_{body}`)`] ρ ≡
F [E_{body}] ρ'
where ρ' = ρ[var_i → F [E_i] ρ']

F [`(`E_{fn} E_1 ... E_n`)`] ρ ≡
callees = F [E_{fn}] ρ;
let fs_i = F [E_i] ρ **in** $\bigcup_{f \in F[E_{fn}]\rho}$ A f <fs_1...fs_n> ρ

Applying functions

A: Fn × Fns* × Env → Fns

Primops:

A [+] <fs_1 fs_2> ρ ≡ ∅

A [`cons`] <fs_1 fs_2> ρ ≡ *escaped* += $fs_1 \cup fs_2$; ∅

A [`car`] <fs_1> ρ ≡ *escaped*

User-defined fns:

A [`(lambda (`$v_1 \ldots v_n$`)` E`)`] <$fs_1 \ldots fs_n$> ρ ≡
F [E] ρ'
where ρ' = *lexical_env*[$v_i \rightarrow fs_i$]

Problems:

- *escaped*
- *lexical_env*
- not computable:
  infinite # of procedure activations

Approximation

One abstract binding per source variable,
one abstract activation record per procedure

E.g.

A [`(lambda (`$v_1 \ldots v_n$`)` E`)`] <$fs_1 \ldots fs_n$> ρ =
ρ(v_i) += fs_i; F [E] ρ

Solve iteratively:

- initially assume empty fns set for each variable
- keep enlarging sets until no more change

Example

```
(define (a)
  (b (lambda₁ (f) (c f))))

(define (b m)
  (c (lambda₂ (g) (d d)))
  (m d))

(define (c q)
  (q q))

(define (d s)
  (c (lambda₃ (w) s)))
```


Shivers's CFA hierarchy

0-CFA:
share one abstract activation record across all callers

1-CFA:
track different values of procedure's locals for each caller
\+ don't smear together info across callers

k-CFA:
use *k* enclosing procedures as specializer
− analysis time exponential in *k*

Compiling programs with higher-order functions

A source-to-source transformation-based approach

- Rabbit [Steele 76], Orbit [Kranz *et al.* 86], Shivers [88], SML/NJ compiler [Appel 89]

1. Expand away syntactic sugar
2. Convert to **continuation-passing style (CPS)**
   - linearize expression evaluation
   - make control flow explicit, model as procedure calls
3. Optimize program in CPS form
   - source-to-source optimizations
4. Optimize closure representations
   - strategy analysis
   - representation analysis
5. Generate target code

Key idea:
convert everything into a call of a closure,
then optimize calls & closures well

Source-to-source transformation style
helps reason about correctness

Example

```
(define (factorial n)
  (define (fact n ans)
    (if (= n 0) ans
        (fact (- n 1) (* n ans))))
  (fact n 1))
...
(factorial 20)
```

⇒ (remove `define` syntactic sugar)

```
(letrec (
    ...
    (factorial (lambda (n)
      (letrec ((fact (lambda (n ans)
          (if (= n 0) ans
              (fact (- n 1) (* n ans))))))
        (fact n 1))))
    ... )
  (factorial 20))
```


CPS conversion

Restrict function call expression to simple form:

- E_{fn} can only be a a var or a primop
- E_{arg} can only be a lambda, a var, or a constant

⇒ forces linearization of complex expressions

Calls don't return:
instead they invoke an explicit **continuation** arg fn,
passing it the result

```
(+ x y) ⇒ (+ x y (lambda (result) ...))
```

⇒ replaces returns with calls

Example:

```
(print (+ (* x y) (- c d)))
⇒
(* x y (lambda (t1)
  (- c d (lambda (t2)
    (+ t1 t2 (lambda (t3)
      (print t3 (lambda () 'done)) )) )) ))
```


CPS of `if`

Can make `if` a regular primop after CPS conversion

```
(if b t f)
⇒
(if b (lambda () t) (lambda () f))
```

Example

```
(letrec (
   (factorial (lambda (n)
     (letrec ((fact (lambda (n ans)
         (if (= n 0) ans
              (fact (- n 1) (* n ans))))))
       (fact n 1)))))
  (factorial 20))
```

⇒ (CPS conversion)

```
(letrec (
   (factorial (lambda (n k)
     (letrec ((fact (lambda (n ans k1)
         (= n 0 (lambda (t1)
         (if t1
           (lambda () (k1 ans))
           (lambda ()
             (- n 1 (lambda (t2)
             (* n ans (lambda (t3)
             (fact t2 t3 (lambda (t4)
             (k1 t4))))))))))))))
       (fact n 1 (lambda (t5)
       (k t5))))))
  (factorial 20 (lambda (t6) (print t6))))
```


Features of CPS form

`lambda` represents a label and an environment

- continuation arg of proc
  ≈ return addr + stack pointer

Calls don't return ⇒ implement as a jump

- also assign args, if any
- primops with lambda continuations just fall through or branch
  - closure is inlined away

Can optimize uses of `lambda` through source-level simplifications

- beta-reduction: inline expansion
- eta-reduction:
  `(lambda (r) (k r))` ⇒ `k`

Tail call optimization

Converting to CPS + eta-reduction implements tail call elimination

In traditional terms:

- if last thing before return is call, just pass caller's return address as callee's return address
- \+ optimizes away intermediate stack frame
- − debugging?

If last operation is recursive call, turns recursion into loop
⇒ tail recursion elimination

Example

```
(letrec ((factorial (lambda (n k)
      (letrec ((fact (lambda (n ans k1)
          (= n 0 (lambda (t1)
          (if t1
            (lambda () (k1 ans))
            (lambda ()
              (- n 1 (lambda (t2)
              (* n ans (lambda (t3)
              (fact t2 t3 (lambda (t4)
              (k1 t4))))))))))))))
        (fact n 1 (lambda (t5)
        (k t5))))))
  (factorial 20 (lambda (t6) (print t6))))
```

⇒ (eta reduction/tail call optimization)

```
(letrec ((factorial (lambda (n k)
      (letrec ((fact (lambda (n ans k1)
          (= n 0 (lambda (t1)
          (if t1
            (lambda () (k1 ans))
            (lambda ()
              (- n 1 (lambda (t2)
              (* n ans (lambda (t3)
              (fact t2 t3 k1)))))))))))
        (fact n 1 k)))))
  (factorial 20 print)))
```