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*

Alternative: context-sensitive analysis

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

Intermediate representation is always a legal Scheme program

- executable intermediate form, w/ clear semantics

`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)))
```


Optimizing functional data structures

If data structures are functional,
"updates" modeled by returning a copy that has the change

+ referential transparency, etc.
– slow for big data structures

E.g.

```
fun mul-by(a, n) =
  for i := 1 to (len a) do
    a := update(a, i, a[i] * n)
  end
  return a
```

Functional version copies `a` *k* times, where k = `len a`
⇒ O(n^2) time, O(n^2) space allocated

Imperative version, update in place
⇒ O(n) time, no space allocated

Single-threading

Idea: if could tell at compile-time that intermediate copy of `a` becomes garbage after the `update` call, then could reuse in place w/o copying

- "compile-time ref counting/garbage collection", "single-threading analysis"

Technique: static analysis of reference counts of data structures

For each expression (variable if CPS form)
maintain count of # of remaining uses of expression

- decrement count after each use
- decrement count when `if` resolved away from uses

For `update` operation,
if ref count of updated variable is 0 after use,
change `update` to `update-in-place`

Example

```
fun mul-by(a, n) =
  for i := 1 to (len a) do
    a := update(a, i, a[i] * n)
  end
  return a
```

Imperative nature of local variables makes this hard to deal with

Convert to functional form w/ tail recursion (mechanically):

```
fun mul-by(a, n) =
  let fun mul-rest(a, n, i) =
    if i > (len a) then
      a
    else
      let x  = a[i] * n,
          a1 = update(a, i, x) in
        mul-rest(a1, n, i+1)
    endif
  in
    mul-rest(a, n, 1)
```

Example continued

Annotate with # of remaining uses for each var:

```
fun mul-by(a, n) =
  let fun mul-rest(a, n, i) =

    if i > (len a) then

      a
    else

      let x  = a[i] * n,

          a1 = update(a, i, x) in

        mul-rest(a1, n, i+1)
    endif
  in

    mul-rest(a, n, 1)
```


Handling formals

Option 1:
assume formals have ∞ ref count

Option 2:
do interprocedural analysis

Option 3:
do interprocedural analysis and procedure specialization

Example of option 3

```
fun mul-by(a, n) =
  let fun mul-rest(a, n, i) =
    if i > (len a) then
      a
    else
      let x  = a[i] * n,
          a1 = update(a, i, x) in
        mul-rest(a1, n, i+1)
    endif
  in
    mul-rest(a, n, 1)
```

Using only "local interprocedural" analysis, `mul-rest` called
from one place with ∞ ref count
and another with 0 ref count

- if 0 calling ref count, then could optimize `update`

Apply specialization to make two copies of `mul-rest`

- one copy gets optimized

After specialization

```
fun mul-by(a, n) =
  let
    fun mul-rest∞(a, n, i) =
        if i > (len a) then a
        else let x  = a[i] * n,
                 a1 = update(a, i, x) in
            mul-rest0(a1, n, i+1)
        endif
    fun mul-rest0(a, n, i) =
        if i > (len a) then a
        else let x  = a[i] * n,
                 a1 = update(a, i, x) in
            mul-rest0(a1, n, i+1)
        endif
  in
    mul-rest∞(a, n, 1)
```

Now can optimize second `update` to `update-in-place`

Before code generation:

```
fun mul-by(a, n) =
  let
    fun mul-rest∞(a, n, i) =
      if i > (len a) then a
      else let x  = a[i] * n,
               a1 = update(a, i, x) in
          mul-rest0(a1, n, i+1)
      endif
    fun mul-rest0(a, n, i) =
      if i > (len a) then a
      else let x  = a[i] * n,
               a1 = update-in-place(a, i, x) in
          mul-rest0(a1, n, i+1)
      endif
  in mul-rest∞(a, n, 1)
```

After code generation:

```
fun mul-by(a, n) =
  if 1 > (len a) then return a;
  a1 := update(a, 1, a[1] * n);
  for i := 2 to (len a1) do
    a1[i] := a1[i] * n;
  end
  return a1;
```