Continuation-passing-style compilation

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, introduce temps
   - make control flow explicit, 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)))
```

Representation optimizations

Leroy [92], Jones [94], Shao & Appel [95], Morrisett et al. [96]

Standard implementations of polymorphic languages keep all data in standard universal representation

+ makes passing to/returning from polymorphic function easy
+ makes GC easy

– basic operations on data may be slower
– lots more heap allocation of e.g. floats
– passing data to/from other languages expensive

Idea: keep data in native format whenever of non-polymorphic type

+ makes monomorphic code fast, lower heap allocation
+ makes interaction with foreign languages easy
– makes passing to/from polymorphic routine expensive
  - arbitrarily more expensive, if passing e.g. a list
– GC?

Examples

If know static type is an int, or a float, compile code as in C

- unboxed form

If know static type is an N-tuple or record, compile as a C struct

- recursively unbox component types
- if argument to function is an N-tuple, pass arg struct in registers
- same for result

```
datatype ('a,'b) Pair = Pair of (a * b);
fun (x:int, y:(float, 'a), Pair(a:int,b:'a)) =
  ...
```


Polymorphic functions

[Leroy, Shao & Appel]

If pass unboxed data to a polymorphic function, box it first

- unbox result data similarly

```
id:'a -> 'a
val y = id(3)
  ⇒
val y = unbox_int(id(box_int(3)))
```

If pass polymorphic function to monomorphic context, wrap it with box and unbox operations first

```
map id [3,4,5]
  ⇒
map
  (fn x:int => unbox_int(id(box_int(x))))
  [3,4,5]
```


Polymorphic recursive data structures

Box and unbox operations are cheap, constant-time for:

- ints, floats, & other scalars
- tuples/records of cheap types
- datatypes whose element types are cheap

Not so good for recursive polymorphic datatypes like `list`: must box/unbox each element

A solution: always keep elements of polymorphic recursive datatypes in boxed form

- removes a lot of the opportunity to keep data unboxed

Passing type descriptors

Instead of keeping data accessed from polymorphic context in boxed (i.e. self-describing) form,

pass separate type descriptor that indicates how to access data

+ can keep all data in native format, even in lists
+ no box/unbox overhead
– extra overhead to pass, use type descriptors

TIL [Morrisett et al.]: “intensional polymorphism”

- e.g. array of ints vs. array of floats vs. array of ptrs
- polymorphic code “typecases” on the descriptor to select appropriate monomorphic version

Haskell: dictionaries

- to support type classes, pass an array of function pointers for each type variable of a polymorphic function

Specialization

Want to avoid the cost of boxing/unboxing, type/dictionary passing & accessing

when using polymorphic code in monomorphic context

Solution: produce multiple **specializations** of polymorphic code for different monomorphic contexts

TIL, ML: inline polymorphic routines, specializing them to each call site

- all polymorphism inlined away in benchmarks
  - benchmarks < 2K-line
  - assumes whole-program optimization
- 3x speedup over SML/NJ

Jones, Haskell: exhaustively specialize polymorphic functions for each instantiating dictionary value

- also remove unreachable procedures (treeshaking)
- code size *reduction* for specialization & treeshaking

– requires whole-program info