School of Computer Science

Scheme Flow Analysis Note 1 5/1/90

CPS Data-Flow Analysis Example

Olin Shivers
shivers@cs.cmu.edu

This is an informal working note intended for limited distribution. It was written 8/4/89, and typeset for LaTeX and updated 5/1/90.

Sections 10.1 and 10.2 of Aho, Sethi, and Ullman's *Compilers: Principles, Techniques, and Tools* (the Red Dragon book) trace through a series of code improvements applied to a single example: the partition loop of Quicksort. In this note, I redo the same code improvements for the same code fragment using a CPS Scheme intermediate representation.

The idea is to see how the analysis and optimisations look when cast into the CPS framework. By performing the optimisations by hand, we can get a feel for what sort of analysis is required to perform the CPS analogs of the standard data-flow based code improvements. This gives us a target to shoot for when working on analysis and optimisation techniques for CPS-based intermediate languages. We will also get to see how suitable CPS Scheme is as an intermediate representation for a compiler that pushes hard on serious optimisation.

In this note, I assume arithmetic functions like `+` and `<` are fixnum only. I don't make any attempt at type recovery or worry about generating safe code; functions are applied "dangerously" — that is, if you apply `+` to a symbol, something weird happens, as opposed to an error being reliably signalled. Also, I do not worry about maintaining GC invariants or anything — I go for array pointers whenever possible.

Type recovery, particularly for the purpose of optimising type-safe code is an interesting issue, of course; I just ignored it here to concentrate on the more traditional optimisations, using the more traditional assumptions. The novelty here is doing it all in the CPS intermediate representation.

I adapted the code fragment we are working with from a real quicksort utility written in T with the Yale loop macro. The procedure `(pick-median! v start end)` picks the median of `v[start]`, `v[end]`, and `v[(start+end)/2]`, swaps it into `v[start]`, and returns it as `pick-median!`'s value. This would be an integrated procedure. The partition code looks like this:

```
(define (quicksort-partition v start end)
  (loop (initial (value (pick-median! v start end))
                 (l start)
                 (r (+ 1 end)))

        ;; Skip past left and right elts on the correct side of the partition.
        (next (l (loop (incr l in l)
                       (while (< (vref v l) value))
                       (result l)))
              (r (loop (decr r in r)
                       (while (< value (vref v r)))
                       (result r))))
        (while (< l r))
        ;; Swap v[l] and v[r].
        (do (set (vref v l) (swap (vref v r) (vref v l))))

        ;; Swap v[start] and v[r], and return r.
        (after (set (vref v start) (swap (vref v r) (vref v start))))
        (result r)))
```

Consider the meat of the loop, expanded out into `letrec` forms:

```
(letrec (((loop1 l r)
          (let* ((l (letrec (((loop2 l)
                               (let ((l (+ l 1)))
                                 (if (< (vref v l) value)
                                     (loop2 l)
                                     l))))
                      (loop2 l)))
                 (r (letrec (((loop3 r)
                               (let ((r (- r 1)))
                                 (if (< value (vref v r))
                                     (loop3 r)
                                     r))))
                      (loop3 r))))
            (cond ((< l r)
                   (set (vref v l) (swap (vref v r) (vref v l)))
                   (loop1 l r))
                  (else
                   (set (vref v start) (swap (vref v r) (vref v start)))
                   r)))))
  (loop1 start (+1 end)))
```

Partially CPS-convert the code by introducing variable continuations (don't bother with "constant" lambda continuations) and expand out the vector refs. Assume a byte-addressable architecture, so we multiply a vector index by 4 to get the byte offset of the vector element. Most of the CPS structure in basic blocks is sugared away with `let*`. The `<clause-cont>`'s in this code mean the continuation that continues on to the next `let*` clause. It's just a way of sleazing out of writing out the full-blown (unreadable) CPS form.

```
(letrec (((loop1 l r kl1)
          (let* ((l (letrec (((loop2 l kl2)
                                (let* ((l (+ l 1))
                                       (i2 (* l 4))
                                       (ptr2 (+ i2 v))
                                       (v2 (load ptr2))) ; v[l]
                                  (if (< v2 value)
                                      (loop2 l kl2) ; loop
                                      (kl2 l)))))   ; done
                      (loop2 l <clause-cont>)))
                 (r (letrec (((loop3 r kl3)
                                (let* ((r (- r 1))
                                       (i3 (* r 4))
                                       (ptr3 (+ i3 v))
                                       (v3 (load ptr3))) ; v[r]
                                  (if (< value v3)
                                      (loop3 r kl3) ; loop
                                      (kl3 r)))))   ; done
                      (loop3 r <clause-cont>))))
            (cond ((< l r) ; swap v[l] and v[r]; loop.
                   (let* ((i4 (* r 4))
                          (ptr4 (+ i4 v))
                          (v4 (load ptr4))
                          (t1 v4)) ; t1 := v[r]
                     ;; v[r] := v[l]
                     (let* ((i5 (* r 4))
                            (ptr5 (+ i5 v)) ; v[r] lvalue
                            (i6 (* l 4))
                            (ptr6 (+ i6 v))
                            (v6 (load ptr6))) ; v[l]
                       (store v6 ptr5))
                     ;; v[l] := t1
                     (let* ((i7 (* l 4))
                            (ptr7 (+ i7 v))) ; v[l] lvalue
                       (store t1 ptr7)))
                   (loop1 l r kl1))

                  (else ; swap v[r] & v[start]; quit.
                   (let* ((i8 (* r 4))
                          (ptr8 (+ i8 v))
                          (v8 (load ptr8))
                          (t2 v8)) ; t2 := v[r]
                     ;; v[r] := v[start]
                     (let* ((i9 (* r 4))
                            (ptr9 (+ i9 v)) ; v[r] lvalue
                            (i10 (* start 4))
                            (ptr10 (+ i10 v))
                            (v10 (load ptr10))) ; v[start]
                       (store v10 ptr9))
                     ;; v[start] := t2
                     (let* ((i11 (* start 4))
                            (ptr11 (+ i11 v))) ; v[start] lvalue
                       (store t2 ptr11)))
                   (kl1 r))))))
  (loop1 start (1+ end) *c*)) ; *C* is the top-level continuation.
```

Next, we can apply some standard local optimisations, primarily β-substitution and local common subexpression elimination. Most reasonable Scheme compilers can manage this much. The effect of β-substitution on the constant continuations inside basic blocks of code is to widen the scopes of `let`-bound variables. This introduces more opportunities for local CSE. Basic blocks in CPS Scheme code are essentially `let*` expressions with simple clauses, as in Kelsey's dissertation.

As a small example, β-substitution on constant continuations gives us

```
(let ((a 4))                     (let ((a 4))
  (let ((b (* a c)))               (let ((b (* a c)))
    (foo b))             ⇒           (foo b)
  (+ 6 (* a c)))                     (+ 6 (* a c))))
```

Widening the scope of `b` allows us to spot `(* a c)` as a local CSE, and improve the code to:

```
(let ((a 4))
  (let ((b (* a c)))
    (foo b)
    (+ 6 b)))
```

This sort of scope-widening happens automatically as a result of general β-substitution code improvement. In CPS, the continuation for the `(let ((b ...)) ...)` expression is a thunk for the `(+ 6 (* a c ))` form. β-substituting this thunk for the variable to which it is bound in the inner `let`'s CPS expansion gives us the scope widening we wanted. Notice that we are relying on alphatisation to prevent variable capture when we migrate expressions down through inner scopes.

Widening scopes on the basic blocks in our partition loop exposes its local CSE's. For example, `i6` and `i7` are common subexpressions, and can be locally spotted and combined once `i7` is migrated into `i6`'s scope. Combining local CSE's wherever possible reduces the loop to the following:

```
(letrec (((loop1 l r kl1)
          (let* ((l (letrec (((loop2 l kl2)
                              (let* ((l (+ l 1))
                                     (i2 (* l 4))
                                     (ptr2 (+ i2 v))
                                     (v2 (load ptr2))) ; v[l]
                                (if (< v2 value)
                                    (loop2 l kl2) ; loop
                                    (kl2 l)))))   ; done
                      (loop2 l <clause-cont>)))
                 (r (letrec (((loop3 r kl3)
                              (let* ((r (- r 1))
                                     (i3 (* r 4))
                                     (ptr3 (+ i3 v))
                                     (v3 (load ptr3))) ; v[r]
                                (if (< value v3)
                                    (loop3 r kl3) ; loop
                                    (kl3 r)))))   ; done
                      (loop3 r <clause-cont>))))

            (if (< l r)
                ;; swap v[l] and v[r]; loop.
                (let* ((i4 (* r 4))
                       (ptr4 (+ i4 v))   ; v[r] lvalue
                       (v4 (load ptr4))  ; v[r]
                       (i6 (* l 4))
                       (ptr6 (+ i6 v))   ; v[l] lvalue
                       (v6 (load ptr6))) ; v[l]
                  (store v6 ptr4) ; v[r] := v[l]
                  (store v4 ptr6) ; v[l] := old v[r]
                  (loop1 l r kl1)); loop

                ;; swap v[r] & v[start]; quit.
                (let* ((i8 (* r 4))
                       (ptr8 (+ i8 v))  ; v[r] lvalue
                       (v8 (load ptr8)) ; v[r]
                       (i10 (* start 4))
                       (ptr10 (+ i10 v))   ; v[start] lvalue
                       (v10 (load ptr10))) ; v[start]
                   (store v10 ptr8) ; v[r] := v[start]
                   (store v8 ptr10) ; v[start] := old v[r]
                   (kl1 r)))))) ; quit

  (loop1 start (1+ end) *c*)) ; *C* is the top-level continuation.
```

Now, we can use data-flow analyses to improve the inter-basic-block control flow. Copy propagation reveals that `loop1`'s continuation `kl1` is always bound to `*c*`, so we can remove `kl1` from the program, replacing its references with `*c*`. Similarly, lambda propagation will allow us to replace `kl2` and `kl3` — the continuations for `loop2` and `loop3` — with the the actual continuation code. This removes most of the variable continuations from the CPS code, makes the control flow more explicit and opens up more opportunities for global CSE. We are left with:

```
(letrec (((loop1 l r)
          (letrec (((loop2 l)
                    (let* ((l (+ l 1))
                           (i2 (* l 4))
                           (ptr2 (+ i2 v))   ; v[l] lvalue
                           (v2 (load ptr2))) ; v[l]
                      (if (< v2 value)
                          (loop2 l) ; loop

                          (letrec (((loop3 r)
                                    (let* ((r (- r 1))
                                           (i3 (* r 4))
                                           (ptr3 (+ i3 v))   ; v[r] lvalue
                                           (v3 (load ptr3))) ; v[r]
                                      (if (< value v3)
                                          (loop3 r) ; loop

                                          (if (< l r)
                                              ;; swap v[l] and v[r]; loop.
                                              (let* ((i4 (* r 4))
                                                     (ptr4 (+ i4 v))
                                                     (v4 (load ptr4))
                                                     (i6 (* l 4))
                                                     (ptr6 (+ i6 v))
                                                     (v6 (load ptr6)))
                                                (store v6 ptr4)
                                                (store v4 ptr6)
                                                (loop1 l r))

                                              ;; swap v[r] & v[start]; quit.
                                              (let* ((i8 (* r 4))
                                                     (ptr8 (+ i8 v))
                                                     (v8 (load ptr8))
                                                     (i10 (* start 4))
                                                     (ptr10 (+ i10 v))
                                                     (v10 (load ptr10)))
                                                (store v10 ptr8)
                                                (store v8 ptr10)
                                                (*c* r))))))) ; quit
                            (loop3 r))))))
            (loop2 l))))
  (loop1 start (1+ end)))
```

Next we apply induction-variable elimination. All the `l`'s form a BIVF[1], with $\{\texttt{ptr2}, \texttt{ptr6}\}$ as a DIVF[2] $(= \texttt{l} * 4 + \texttt{v})$; `i2` and `i6` become useless and can be removed. Similarly, all the `r`'s form a BIVF, with $\{\texttt{ptr3}, \texttt{ptr4}, \texttt{ptr8}\}$ as a DIVF $(= \texttt{r} * 4 + \texttt{v})$; `i3`, `i4`, and `i8` are useless and can be removed. Doing this code improvement gives us with the shorter and faster:

[1] Basic induction-variable family
[2] Dependent induction-variable family

```
(letrec (((loop1 l r p2 p3) ; p2 = &v[l]   p3 = &v[r]
          (letrec (((loop2 l p2)
                    (let* ((l (+ l 1))
                           (p2 (+ p2 4))
                           (v2 (load p2)))
                      (if (< v2 value)
                          (loop2 l p2)

                          (letrec (((loop3 r p3)
                                    (let* ((r (- r 1))
                                           (p3 (- p3 4))
                                           (v3 (load p3)))
                                      (if (< value v3)
                                          (loop3 r p3)

                                          (if (< l r)
                                              ;; swap v[l] and v[r]; loop.
                                              (let* ((v4 (load p3))
                                                     (v6 (load p2)))
                                                (store v6 p3)
                                                (store v4 p2)
                                                (loop1 l r p2 p3))

                                              ;; swap v[r] & v[start]; quit.
                                              (let* ((v8 (load p3))
                                                     (i10 (* start 4))
                                                     (ptr10 (+ i10 v))
                                                     (v10 (load ptr10)))
                                                (store v10 p3)
                                                (store v8 ptr10)
                                                (*c* r))))))) ; quit
                            (loop3 r p3))))))
            (loop2 l p2))))
  (loop1 start (1+ end) (+ v (* 4 start)) (+ v (* 4 (1+ end)))))
```

Note that the IVE analysis depended on realising that `v` is a loop-invariant, even though it is not a constant. My current analysis doesn't deal with this.

We can improve this even further if we are willing to rewrite the uses of `l` and `r` in the loop to use the pointer variables `p2` and `p3` instead. The only real use of `l` and `r` in the loop is in the conditional test `(if (< l r) ...)`. This is equivalent to `(if (< p2 p3) ...)` and so we can completely eliminate `l` and `r` from the loop, along with all the arithmetic operations that incremented these indexes. We have to insert code in the epilog of the loop to recompute `r` from `p3`, since `r` is the value of the loop. This leaves us with:

```
(letrec (((loop1 p2 p3) ; p2 = &v[l]   p3 = &v[r]
          (letrec (((loop2 p2)
                    (let* ((p2 (+ p2 4))
                           (v2 (load p2)))
                      (if (< v2 value)
                          (loop2 p2)

                          (letrec (((loop3 p3)
                                    (let* ((p3 (- p3 4))
                                           (v3 (load p3)))
                                      (if (< value v3)
                                          (loop3 p3)

                                          (if (< p2 p3)
                                              ;; swap v[l] and v[r]; loop.
                                              (let* ((v4 (load p3))
                                                     (v6 (load p2)))
                                                (store v6 p3)
                                                (store v4 p2)
                                                (loop1 p2 p3))

                                              ;; swap v[r] & v[start]; quit.
                                              (let* ((v8 (load p3))
                                                     (i10 (* start 4))
                                                     (ptr10 (+ i10 v))
                                                     (v10 (load ptr10)))
                                                (store v10 p3)
                                                (store v8 ptr10)
                                                (*c* (/ (- p3 v) 4))))))))
                            (loop3 p3))))))
            (loop2 p2))))
  (loop1 (+ v (* 4 start)) (+ v (* 4 (1+ end)))))
```

The `v[r]` fetch — `(load p3)` — is a global CSE, so we can replace `v4` and `v8` with `v3`. This requires our analysis to track effects on the store, so we know that `v[r]` isn't set between the initial reference and the subsequent ones. In this case, there are no intervening stores at all, so we're OK.

```
(letrec (((loop1 p2 p3) ; p2 = &v[l]   p3 = &v[r]
          (letrec (((loop2 p2)
                    (let* ((p2 (+ p2 4))
                           (v2 (load p2)))
                      (if (< v2 value)
                          (loop2 p2)

                          (letrec (((loop3 p3)
                                    (let* ((p3 (- p3 4))
                                           (v3 (load p3)))
                                      (if (< value v3)
                                          (loop3 p3)

                                          (if (< p2 p3)
                                              ;; swap v[l] and v[r]; loop.
                                              (let* ((v6 (load p2)))
                                                (store v6 p3)
                                                (store v3 p2)
                                                (loop1 p2 p3))

                                              ;; swap v[r] & v[start]; quit.
                                              (let* ((i10 (* start 4))
                                                     (ptr10 (+ i10 v))
                                                     (v10 (load ptr10)))
                                                (store v10 p3)
                                                (store v3 ptr10)
                                                (*c* (/ (- p3 v) 4))))))))
                            (loop3 p3))))))
            (loop2 p2))))
  (loop1 (+ v (* 4 start)) (+ v (* 4 (1+ end)))))
```

This is about as tight as you can get it. Most C compilers would do much worse. We have replaced all vector indexing with pointer stepping, completely merged local and global CSEs, and removed useless code (UVE).

This lays out the CPS-based compiler agenda pretty clearly.

References

Aho, Sethi, Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley (1986).

Richard Kelsey. *Compilation by Program Transformation.* YALEU/DCS/RR-702. Ph.D. dissertation, Yale University (May 1989). (A conference-length version of this appears in *POPL 89*.)

David Kranz. *ORBIT: An Optimizing Compiler for Scheme.* YALEU/DCS/RR-632. Ph.D. dissertation, Yale University (February, 1988). (A conference-length version of this is in *SIGPLAN 86*.)

Olin Shivers. Control-flow analysis in Scheme. *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation* (June 1988). (Also available as Technical Report ERGO-88-60, CMU School of Computer Science, Pittsburgh, Penn.)

Olin Shivers. Super-β: Copy, constant, and lambda propagation in Scheme. Working note #3 (May 1, 1990).

Olin Shivers. Useless-variable elimination. Working note #2 (April 27, 1990).

Guy Lewis Steele Jr. *Rabbit: A Compiler for Scheme.* MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.