Carnegie Mellon

School of Computer Science

Scheme Flow Analysis Note 3 5/1/90

Super-β: Copy, Constant, and Lambda Propagation in Scheme

Olin Shivers
shivers@cs.cmu.edu

This is an informal note intended for limited distribution. The text will reappear in my dissertation; it appears here for interested parties who don't want to wait. The explication of the technique is a rough sketch, and assumes you are familiar with my paper "Data-Flow Analysis and Type Recovery in Scheme."

1 Copy propagation

Copy propagation is a standard code improvement technique that is based on data-flow analysis. It is, for example, treated in chapter 10 of the Dragon book. Copy propagation essentially is removing variables that are just redundant copies of other variables. That is, if a reference `x+3` to a variable `x` has only defs of the form `x := y`, and `y` is unchanged along intervening control paths, then we can substitute `y` for its copy `x`: `y+3`. If we can do this with all the references to `x`, then we can remove the assignment to `x` completely.

In Scheme, copy propagation is handled in some cases by simple β-substitution. We can substitute `y` for `x` in:

```
(let ((x y))      ⇒     (foo y)
  (foo x))
```

However, β-substitution does not handle cases involving circular or join dependencies. For example, consider the loop which multiplies each element in `vec` by `x`:

```
(letrec ((lp (λ (y i)
               (cond ((>= i 0)
                      (set (vref vec i) (* (vref vec i) y))
                      (lp y (- i 1)))))))
  (lp x 20))
```

Clearly, references to `y` can be replaced by `x`, although simple β-substitution will not pick this up.

Note that this sort of copy propagation can frequently and usefully be applied to the continuation variable in the CPS expansions of loops. For example, consider the iterative factorial, partially CPS-expanded:

```
(define (fact n k)
  (letrec ((lp (λ (ans m c)
                 (if (= m 0) (c ans)
                     (lp (* ans m) (- m 1) c)))))
    (lp 1 n k)))
```

The loop's continuation `c` is always bound to the procedure's continuation `k`, and is redundantly passed around the loop until it is invoked on loop termination. Applying copy propagation (and one round of useless-variable elimination to clean up afterwards) removes the iteration variable `c` entirely:

```
(define (fact n k)
  (letrec ((lp (λ (ans m)
                 (if (= m 0) (k ans)
                     (lp (* ans m) (- m 1))))))
    (lp 1 n)))
```

This sort of situation crops up all the time in CPS-expanded loops.

The general form of Scheme copy propagation, then, is:

> For a given variable v, find all lexically inferior variables that are always bound to the value of v, and replace their references with references to v.

With this definition, Scheme copy propagation acts as a sort of "super-β" substitution rule. (If we allow side-effects to variables, this can disallow some possible substitutions.

If, on the other hand, we use an intermediate representation that has no variable side-effects, we can substitute at will. This is one of the attractions of the particular CPS Scheme representation used by Orbit, the Transformational Compiler, and my work. ML elegantly avoids the issue altogether: variables cannot be side-effected at the source level.)

Note that the environment problem rears its head here. For example, we cannot substitute `x` for `y` in the following code:

```
(let ((f (λ (x g) (if (null? g) (λ () x)
                          (let ((y (g))) ... y ...)))))
  (f 3 (f 5 nil)))
```

Even though `y` is certainly bound to `x` (because `g` is bound to `(λ () x)` at that point), it's the wrong binding of `x` — we need the one that is in the `y` reference's context. Scheme allows us to have multiple bindings of the same variable around at the same time; we have to be careful when we are doing copy propagation to keep these multiple environments separate.

1.1 Copy propagation and reflow

We can handle the environment problem with an analysis similar to the one in "Data-Flow Analysis and Type Recovery in Scheme."

Suppose we have some lambda $\ell =$ `(λ (x y z) ...)`, and we wish to perform copy propagation on `x`. That is, we want to find the lexically inferior variables that are always bound to the (lexically apparent) value of `x` so we can replace them with `x`. We do a reflow, starting with each entry to ℓ. As we enter ℓ, we create a perfect contour for ℓ's variables. All the lexically inferior lambdas in this scope will be closed over the perfect contour, and so won't get their references to `x` confused with other bindings of `x`.

In order to perform the analysis, we need an abstract value domain of two values: a value to represent whatever is bound to the binding of `x` that we're tracking, and a value to represent everything else in the system. Call `x`'s value 0 and the other value 1. The initial system state when we begin a reflow by entering ℓ is that

- the current binding of `x` has value 0
- all other bindings in the environment — including the current bindings of `y` and `z` and other extant bindings of `x` — and all values in the abstract store have value 1.

As we proceed through the abstract interpretation, whenever we enter a lambda ℓ' that is lexically inferior to ℓ, we check to see if ℓ' is closed over the perfect ℓ contour we're tracking. If it is, we save away the value sets passed in as arguments to the variables of ℓ'.

After we've reflowed ℓ from all its call sites, we examine each lexically inferior variable. Consider the value set accumulated for variable v. If it is $\emptyset$, then v's lambda was never called — it's dead code. If it is $\{0\}$, then it is always bound to the value of `x` in its lexical context, and so is a candidate for copy propagation — we can legitimately replace all references to v with `x`. If it is $\{0, 1\}$ or $\{1\}$, then it can potentially be bound to another value, and so is not a candidate for copy propagation.

Notice the conservative nature of the analysis: the presence of 1 in a value set marks a variable as a non-copy. This is because after setting up the initial perfect contour, all the other variable bindings to value sets happen in approximate contours. This means that different bindings can get mapped together. If our analysis arranges for this to affect the analysis in the safe direction, this identification is OK. Ruling out a possible copy is erring on the safe side, and that is the effect we get by unioning together the value sets and looking for 1's .

The general algorithm I outline here is quite inefficient; I've kept it as simple as possible to get the basic idea across. A real implementation could be tuned up for greater efficiency.

2 Constant propagation

Constant propagation is another standard code improvement based on data-flow analysis. In brief, if the only def of `x` that reaches `sin(x)` is `x:=3.14`, then we can replace `sin(x)` with `sin(3.14)` (and, subsequently, -1.0).

In the Scheme and ML world, where we bind variables instead of side-effecting them, this corresponds to finding variables that are always bound to constants, and replacing their references with the constant. Again, β-substitution spots the easy cases. *E.g.*:

```
(let ((x 3.14))
  (sin x))          ⇒     (sin 3.14)
```

Again, we need something more powerful to spot circular and join dependencies.

Constant propagation is much easier (and cheaper) than copy propagation because we don't have to worry about the environment problem. Constants are not sensitive to their environmental context. They can be freely migrated around a program in ways that variables cannot. The analysis procedure for constant propagation can therefore be based on the simple control-flow analysis I present in “The Semantics of Scheme Control-Flow Analysis.” We just augment the variable environment (and the store) to include constants as well as procedures. When we perform a call with a constant argument a, we just pass the singleton set $\{a\}$ as the corresponding value set. When the analysis converges, we look up the data that was bound to a given variable v under all its contours. If there’s only one such datum, and it’s a constant, not a procedure, we can go ahead and replace all references to v with the constant.

3 Lambda propagation

In the CPS Scheme intermediate representation I use, a call argument can be a variable, a constant, or a lambda. Copy and constant propagation are essentially just finding out when a variable is bound to a known variable or constant, and doing the appropriate substitution. Clearly there’s one important case left to be done: binding a variable to a known lambda, which we can call “lambda propagation.”

With lambda propagation, the environment problem returns in force. Like variables, lambda expressions are not context-independent. A procedure is a lambda plus its environment. When we move a lambda from one control point to another, in order for it to evaluate to the same procedure we must be sure that environment contexts at both points are equivalent. This is reflected by the “environment consonance” condition in the following statement of lambda propagation:

> For a given lambda l, find all the places it is used in contexts that are environmentally consonant with its point of closure.

By “environmentally consonant,” I mean that all the lambda’s free variables are lexically apparent at the point of use, and are also bound to the same values as they are at the lambda’s point of closure. That is, if we determine that lambda $\ell =$ `(`λ `(x) (+ x y))` is called from call $c =$ `(g z)`, we must also determine that c appears in the scope of `y`, and further, that it is the same binding of `y` that pertained when we closed ℓ. In this sense, the more free variables a lambda has, the more constraints there are on it being “environmentally consonant” with other control points. At one extreme, combinators — lambdas with no free variables — are environmentally consonant with any call site in the program, and can be substituted at any point, completely unconstrained by the lexical context.

3.1 Simplifying the consonance constraint

Suppose we want to find all the places we can substitute lambda expression $\ell =$ `(`λ `(x) ... y ... z ... w)`. ℓ is closed over three free variables: `y`, `z`, and `w`.

Let us suppose that y is the innermost free variable, that is, ℓ's lexical context looks something like this:

```
(λ (w)
  ...
  (λ (z)
    ...
    (λ (y)
      ...
      (λ (a)
        ...
        ℓ:(λ (x) ... y ... z ... w)
        ...)
      ...)
    ...)
  ...)
```

We can substitute ℓ only within the scope of y. (Notice that since a is not free in ℓ, occurrences of ℓ can migrate outside of its scope.) The key to ensuring environment consonance is to realise that if we ensure that ℓ is consonant with its innermost free variable — y — then it is consonant with the rest of its free variables. For example, if we call ℓ from a call context that binds y in the same run-time contour as ℓ is closed over, then we can be sure the same goes for z and w.

Given this observation, we can see that a lambda propagation analysis really only needs to be careful about a single contour: the one over the innermost free variable's lambda. In a reflow-based analysis procedure, this means we need to track only one perfect contour at a time.

3.2 Lambda propagation and reflow

Given a lambda μ, we will do lambda propagation for all inferior lambdas ℓ whose innermost free variable is bound by μ. We perform a reflow from all calls to an abstract closure over μ. As we enter μ beginning the reflow, we allocate a perfect contour for μ's variables. As we proceed through the abstract interpretation, we will make closures, some over the perfect contour that we're tracking. Just as in copy propagation, when we enter a lambda μ' that is lexically inferior to μ, we check to see if μ' is closed over the perfect μ contour. If it is, we save away the value sets passed in as arguments to the variables of μ'.

When we're done with our reflows, we examine each variable v that is lexically inferior to μ. Consider the value set accumulated for v. If (1) it consists entirely of closures over a single lambda ℓ, (2) ℓ is lexically inferior to μ, (3) ℓ's innermost free variable is bound by μ, and (4) each closure over ℓ has the perfect contour for μ's scope, then we win: ℓ can be substituted for any reference $r\!:\!v$ to v — the reference is guaranteed to evaluate to a closure over ℓ and the environment that pertains at $r\!:\!v$ is consonant with the one that ℓ is closed over.

3.3 Constraints on lambda substitution

Once we've found the variables for which we can substitute a given lambda, we must decide which cases are appropriate. If a lambda can be substituted at multiple points in a program, performing all the substitutions can lead to code blow-up. We might wish to limit our substitutions to cases where there is only one possible target, or the lambda is below a certain size. We might choose some subset of the references to substitute for, where the compiler thinks there is payoff in expanding out the lambda, leaving other references unchanged.

We have to be careful when substituting a lambda for multiple variable references if more than one of the references is a call argument (instead of a call procedure), lest we break the eqness of procedures. For example, we cannot substitute for `x` in

```
(let ((x (λ () #f))) (eq? x x))
```

Duplicating the lambda will cause the `eq?` test to return false. (ML avoids this problem by disallowing eq tests on procedures.)

One case definitely worth treating with caution is when a lambda is used in its own body, *e.g.*:

```
(letrec ((fact (λ (n)
                 (if (= n 0) 1
                     (* n (fact (- n 1)))))))
  (fact m))
```

Lambda propagation tells us we can legitimately substitute the lambda definition of `fact` for the variable reference `fact` in its own body, giving us (after one round of substitution):

```
(letrec ((fact (λ (n)
                 (if (= n 0) 1
                     (* n ((λ (n) (if (= n 0) 1
                                      (* n (fact (- n 1)))))
                           (- n 1)))))))
  (fact m))
```

Clearly, blindly performing this substitution can get us into an infinite regress. This sort of situation can happen without `letrec`, in cases involving self-application

```
(let ((lp0 (λ (lp1) (lp1 lp1)))) (lp0 lp0))
```

or indirection through the store:

```
(let* ((pair (cons #f #f))
       (lp (λ () ((car pair)))))
  (set-car! pair lp)
  (lp))
```

Substituting a lambda for a variable inside the lambda's body is not necessarily a bad thing to do. It amounts to loop unrolling, which can be desirable in some contexts. But the code optimiser must be aware of this case, and avoid it when it is undesirable.

Most of these substitution constraints also pertain to code improvers that use simple β-substitution; detailed discussions are found in the Rabbit, Orbit and Transformational Compiler theses. In any event, all these caveats and this backing and filling should indicate to the reader that lambda propagation is not such a trivial thing. With the exception of procedural eqness, these caveats are all issues of implementation, not semantics (code blow-up is not, for example, a semantic concern). This makes me somewhat suspicious of the procedural eqness requirement.

A final note on lambda propagation. As we've just seen, there are many constraints determining when we can substitute a lambda for a variable known to be bound to a closure over that lambda: the variable's reference context must be environmentally consonant with the lambda's point of closure; the variable must be referenced only once (to avoid code blow-up); and so forth. However, even when these constraints are not satisfied, it is useful to know that a variable reference must evaluate to a closure over a particular lambda. The compiler can compile a procedure call as a simple branch to a known location instead of a jump to a run-time value which must be loaded into a register. If all calls to a lambda are known calls, the compiler can factor the lambda out of the run-time representation of the closure, allowing for smaller, more efficient representations. This is exactly what Orbit's strategy analysis does, for the simple cases it can spot.

Note that the sort of information required for the general version of Orbit's strategy analysis is just the basic control-flow information provided by the analyses developed in "Control-Flow Analysis in Scheme" and "The Semantics of Scheme Control-Flow Analysis."

4 Cleaning up

When we substitute an expression — a variable, a constant or a lambda — for all references to some variable v, then v becomes a useless variable. It is therefore a candidate for being removed from its lambda's parameter list. This transformation is the one used by the second phase of useless-variable elimination; my note on UVE describes the particulars. The following sequence of transformations shows the role of UVE in post-substitution cleanup.

```
;;; Original loop
(letrec ((lp (λ (sum n c)
               (if (= 0 n) (c sum)
                   (lp (+ sum n) (- n 1) c)))))
  (lp 0 m k))

;;; After copy propagation
(letrec ((lp (λ (sum n c)
               (if (= n 0) (k sum)
                   (lp (+ sum n) (- n 1) k)))))
  (lp 0 m k))

;;; After UVE
(letrec ((lp (λ (sum n)
               (if (= n 0) (k sum)
                   (lp (+ sum n) (- n 1))))))
  (lp 0 m))
```

5 Summary

Copy and constant propagation are classical code improvements based on data-flow analysis. Standard imperative languages typically use side-effects to associate variables with values, and this is what the standard formulations of copy and constant propagation focus on. In Scheme, however, variables are typically associated with values by binding them. With this change of emphasis, copy and constant propagation become simply special cases of an interesting generalisation of the λ-calculus β-substitution rule. This is particularly true in the CPS Scheme internal representation, where side-effects are not allowed to interfere with the substitution rules. This "super-β" model of code improvement gives us one more alternative to copy and constant propagation: "lambda propagation" — substituting lambda expressions for variable references.

In the case of copy and lambda propagation, the context-dependence of variable references and lambda expressions requires our analysis to be sensitive to the environment that exists at the source and target points of the expression being substituted. This is the environment problem mentioned in “Control-Flow Analysis in Scheme;” its solution is based on the general technique presented in “Data-Flow Analysis and Type Recovery in Scheme.”

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 appears 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. CPS data-flow analysis example. Working note #1 (May 1, 1990).

Olin Shivers. Data-flow analysis and type recovery in Scheme. Technical Report CMU-CS-90-115, CMU School of Computer Science, Pittsburgh, Penn. (March 1990). (Also to appear in *Topics in Advanced Language Implementation*, ed. Peter Lee, MIT Press.)

Olin Shivers. The semantics of Scheme control-flow analysis. *(In preparation)*

Olin Shivers. The semantics of Scheme control-flow analysis (Prelim). Technical Report ERGO-90-090, CMU School of Computer Science, Pittsburgh, Penn. (November 1988).

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.