Useless-Variable Elimination

Olin Shivers
Carnegie Mellon University
Pittsburgh, Penn. 15213-3890
shivers@cs.cmu.edu

1 Introduction

Useless-variable elimination (UVE) is a technique for removing unnecessary computations and variables from Scheme programs. In my 1988 SIGPLAN paper "Control-Flow Analysis in Scheme," [CFlow] I demonstrated the optimisation without explaining how to perform it. In this note, I will describe the optimisation in detail, and describe an algorithm for performing it. This technique is generally applicable to other higher-order languages, such as ML.

2 Useless-variable elimination

A *useless variable* is one whose value contributes nothing to the final outcome of the computation. We can remove a useless variable and the computation producing its value from our program. For example, consider the following code fragment:

```
(let ((sum  (+ a b))                (let ((sum (+ a b)))
      (prod (* a b)))     ⇒           (f sum a))
  (f sum a))
```

`Prod` is never used in the program, so we can remove it and its binding computation `(* a b)`. This example is fairly easy to detect; most Scheme compilers would find and optimise it.

On the other hand, some useless variables involve circular dependencies or multiple (join) dependencies. The simple lexical analysis that suffices for the above example won't spot these cases. For example, consider the factorial loop:

```
(letrec ((lp (λ (ans j bogus)
               (if (= 0 j) ans
                   (lp (* ans j)
                       (- j 1)
                       (sqrt bogus))))))
  (lp 1 n n))
```

Although the variable `bogus` doesn't contribute anything at all to the final result, it appears to be used in the loop. Useless-variable elimination is used to spot these cases.

UVE is often useful to clean up after applying other code transformations, such as copy propagation or induction-variable elimination. When we introduce a new variable to track an induction function on some basic induction variable, the basic variable frequently becomes useless. For an example produced by a working implementation, compare parts (d) and (e) of figure 4 in "Control-Flow Analysis in Scheme" [CFlow].

In continuation-passing style (CPS) intermediate representations, continuation variables are frequently passed around loops needlessly. Copy propagation [Diss] plus useless-variable elimination can transform these cases. An example is the following series of transformations on a loop:

```
;;; Original loop -- c refs can be replaced by k refs.
(λ (k)
  (letrec ((lp (λ (sum n c)
                 (if (= 0 n) (c sum)
                     (lp (+ sum n) (- n 1) c)))))
    (lp 0 m k)))

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

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

3 Finding useless variables

Detecting these cases requires a simple backwards flow analysis on the CPS intermediate form. (I use a CPS representation essentially identical to the one used in the ORBIT Scheme compiler [Orbit].) We actually compute a conservative approximation to the inverse problem — finding the set of all useful variables. We start with variables that must be assumed useful (*e.g.*, a variable whose value is returned as the value of the procedure being analysed). Then we trace backwards through the control-flow structure of the program. If a variable's value contributes to the computation of a useful variable, then it, too, is marked useful. When we're done, all unmarked variables are useless. This gives us a mark-and-sweep algorithm for a sort of "computational gc."

To be specific, in the implementation of UVE that I have written, a variable is *useful* if it appears

- in the function position of a call:
  `(f 5 0)`
- as the predicate in a conditional primop:
  `(if p (λ () ...) (λ () ...))`
- as the continuation of a primop:
  `(+ 3 5 c)`
- as an argument in a call to
  - a side-effecting operation (output or store):
    `(print a)`, `(set-car! x y)`
  - an external procedure, or a primop whose continuation is an external procedure.
  - a primop whose continuation binds a useful variable:
    `(+ metoo 3 (λ (used) ...))`

– a lambda whose corresponding parameter is useful:
`((λ (x used y) ...) 3 metoo 7)`

The first three conditions spot variables used for control flow. The next two mark variables whose value escapes, and must therefore be assumed useful. The final two recursive conditions are the ones that cause the analysis to chain backwards through the control-flow graph: if a variable is useful, then all the variables used to compute its value are useful. The control-flow graph needed for the backward chaining can be recovered with a preliminary control-flow analysis [CFlow, CFASem, Diss].

4 Optimising useless variables

Once we have found the useless variables in a program, we can optimise the program in two steps. In the first step, we remove all references to useless variables and eliminate all useless computations (primop calls). In the second step, we remove useless variables from their lambda lists where possible.

4.1 Removing useless variables from calls

In this phase, we globally remove all references to useless variables in our program. If a useless variable appears as an argument to a primop call, we remove the entire primop computation. For instance, suppose the useless variable `x` appears as an argument in the primop call `(+ x y k)`. By the definition of a useless variable, we know that the continuation `k` must bind the result of the addition to a useless variable, as well (if it didn't, we'd have marked `x` as useful). This renders the addition operation useless, so we can remove it, replacing `(+ x y k)` with `(k #f)`. The actual value passed to the continuation (we used `#f` here) is not important — remember that we are globally deleting all references to useless variables. Since the continuation `k` must bind the value `#f` to a useless variable, the value is guaranteed never to be referenced.

If a reference to a useless variable appears in a non-primop call, we simply replace it with some constant. If `x` is useless, we convert `(f x y)` to `(f #f y)`. Similar reasoning applies in this case: if `x` is useless, it must be the case that `f`'s corresponding parameter is useless as well. All references to this parameter will be deleted, so we can pass any value we like to it.

4.2 Removing useless variables from lambda lists

Suppose we have determined that `x` is useless in lambda ℓ = `(`λ` (x y) ...)`. After applying the transformation of the previous subsection, we can be sure that ℓ contains no references to `x`. The only remaining appearance of `x` is in ℓ's parameter list. Consider the places this lambda is called from, *e.g.*, `(f a 7)`. We can delete both the formal parameter `x` from its lambda list and the corresponding argument `a` from its call:

`(`λ` (x y) ...)` $\Rightarrow$ `(`λ` (y) ...)` and `(f a 7)` $\Rightarrow$ `(f 7)`.

However, we can't apply this optimisation in all cases. Suppose ℓ is called from two places, the external call and `(f a 7)`. We can't simply delete `x` from ℓ's parameter list, because the external call, which we have no control over, is going to pass a value to ℓ for `x`. Or, suppose that our lambda is only called from one place, `(f a 7)`, but that call site calls two possible lambdas. Again, we can't delete the argument `a` from the call, because the other procedure is expecting it (unless it, as well, binds a useless variable).

We have a circular set of dependencies determining when it is safe to remove a useless variable from its lambda list and its corresponding arguments from the lambda's call sites:

- We can delete a variable from a lambda list only if we can delete its corresponding argument in all the calls that could branch to that lambda.
- We can delete an argument from a call only if we can delete the corresponding formal parameter from every lambda we could branch to from that call.

It is not hard to compute a maximal solution to these constraints given control-flow information. We use a simple algorithm that iterates to a fixed point. We compute two sets: the set RV of removable useless variables, and the set RA of removable call arguments. Initialise RV to be the set of all useless variables. For each useless variable v , find all the call sites that could branch to v's lambda, and put the call's corresponding argument into RA. Then iterate over these sets until we converge:

```
ITERATE until no change
    FOR each argument a in RA
        IF a's call could branch to a lambda whose
           corresponding parameter is not in RV
        THEN remove a from RA
    FOR each variable v in RV
        IF v's lambda can be called from a call site whose
           corresponding argument is not in RA
        THEN remove v from RV
```

For the purposes of the first loop, the external lambda counts as a disqualifying branch target; for the purposes of the second loop, the external call counts as a disqualifying branch source. When we're done, we're left with RV and RA sets that satisfy the circular criteria above. We can safely remove all the arguments in RA from their calls and all the variables in RV from their lambda lists.

5 Results

Applying these two phases of optimisation will eliminate useless variables and their associated computations wherever possible. For example, we can remove the useless `bogus` variable and its square-root calculation from the factorial loop of section 2, leaving only the necessary computations:

```
(letrec ((lp (λ (ans j)
               (if (= 0 j) ans
                   (lp (* ans j)
                       (- j 1))))))
  (lp 1 n))
```

Note that as a special case of UVE, all unreferenced variables in a program are useless. The second phase of the UVE optimisation will remove these variables when possible.

References

[CFlow] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of the 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.

[CFASem] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the First ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1991. Published as *SIGPLAN Notices* 26(9):190–198, Association for Computing Machinery, September 1991. Also available as Technical Report CMU-CS-91-119, CMU School of Computer Science, Pittsburgh, Penn. An early version was available as Technical Report ERGO-90-090.

[Diss] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* Ph.D. dissertation, Carnegie Mellon University, May 1991. Technical Report CMU-CS-91-145, School of Computer Science.

[Orbit] David Kranz. *ORBIT: An Optimizing Compiler for Scheme.* Ph.D. dissertation, Yale University, February 1988. Research Report 632, Department of Computer Science. A conference-length version of this dissertation appears in *SIGPLAN 86*.

[TRec] 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*, Peter Lee (editor), MIT Press.

Note: This note and all of the above-cited papers authored by myself can be retrieved by anonymous ftp from CMU in Postscript and .dvi format. Ftp to any CMU host with access to the `/afs` network file system (almost any host will do; some possibilities are `a.gp.cs.cmu.edu` and `cs.cmu.edu`). Login as `anonymous`. Cd to

```
/afs/cs.cmu.edu/user/shivers/lib/papers
```

and retrieve files from this directory. Note that anonymous ftp at CMU has access only to certain specific directories, so you must cd straight to `.../lib/papers`.