ACM SIGPLAN

Functional Programming

Editor: *Philip Wadler, Bell Laboratories, Lucent Technologies*; *wadler@research.bell-labs.com*

SSA is Functional Programming

Andrew W. Appel

Static Single-Assignment (SSA) form is an intermediate language designed to make optimization clean and efficient for imperative-language (Fortran, C) compilers. Lambda-calculus is an intermediate language that makes optimization clean and efficient for functional-language (Scheme, ML, Haskell) compilers. The SSA community draws pictures of graphs with basic blocks and flow edges, and the functional-language community writes lexically nested functions, but (as Richard Kelsey recently pointed out [9]) they're both doing exactly the same thing in different notation.

SSA form. Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression. The *def-use chain* is a data structure that makes this efficient: for each statement in the flow graph, the compiler can keep a list of pointers to all the *use* sites of variables defined there, and a list of pointers to all *definition* sites of the variables used there. But when a variable has N definitions and M uses, we might need $N \cdot M$ pointers to connect them.

The designers of SSA form were trying to make an improved form of def-use chains that didn't suffer from this problem. Also, they were concerned with "getting the right number of names:" the programmer might use some variable i for several unrelated purposes in the same procedure – for example, as the loop counter for two different loops – and we can do more optimization if we split i into different variables i_1 and i_2.

In SSA, each variable in the program has only one definition – it is assigned to only once. The assignment might be in a loop, which is executed many times; so single-assignment is a *static* property of the program text, not a dynamic property of program execution.

$$\begin{array}{lcl} a & \leftarrow & x+y \\ b & \leftarrow & a-1 \\ a & \leftarrow & y+b \\ b & \leftarrow & x\cdot 4 \\ a & \leftarrow & a+b \end{array} \qquad\qquad \begin{array}{lcl} a_1 & \leftarrow & x+y \\ b_1 & \leftarrow & a_1-1 \\ a_2 & \leftarrow & y+b_1 \\ b_2 & \leftarrow & x\cdot 4 \\ a_3 & \leftarrow & a_2+b_2 \end{array}$$

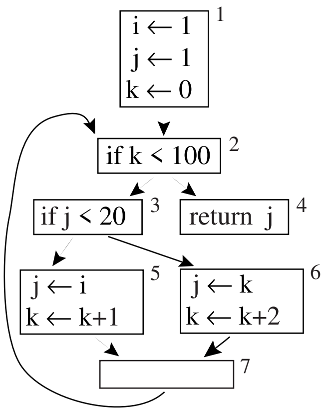
To achieve single-assignment, we make up a new variable name for each assignment to the variable. For example, we convert the program at left into the single-assignment program at right. At left, a use of a at any point refers to the most recent definition, so we know where to use a_1, a_2, or a_3, in the program at right.

For a program with no jumps this is easy. But where two control-flow edges join together, carrying different values of some variable i, we must somehow merge the two values. In SSA form this is done by a notational trick, the ϕ-function. In some node with two in-edges, the expression $\phi(a_1, a_2)$ has the value a_1 if we reached this node on the first in-edge, and a_2 if we came in on the second in-edge.

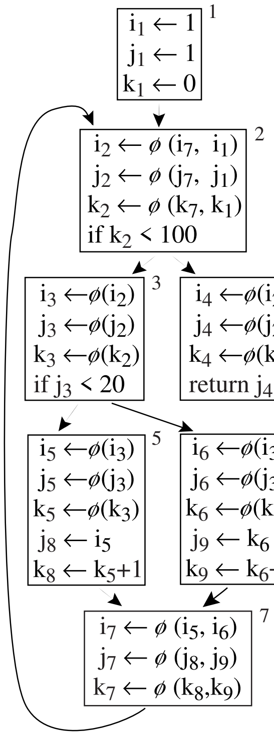
Let's use the following program to illustrate:

$$\begin{array}{l} i \leftarrow 1 \\ j \leftarrow 1 \\ k \leftarrow 0 \\ \textbf{while } k < 100 \\ \qquad \textbf{if } j < 20 \\ \qquad\quad j \leftarrow i \\ \qquad\quad k \leftarrow k+1 \\ \qquad \textbf{else} \\ \qquad\quad j \leftarrow k \\ \qquad\quad k \leftarrow k+2 \\ \text{return } j \end{array}$$

First we turn this into a control-flow graph (CFG):



Now, the question is, where to put the ϕ-functions and how to rename the variables. A *really crude approach* is to split every variable at every basic-block boundary, and put ϕ-functions for every variable in every block:



Yuck! This isn't "the right number of names!" There are too many variables and useless copies. More about this later.

Meanwhile, we can view this program as a set of mutually recursive functions, where each function takes arguments i, j, k:

function $f_1() =$
  let $i_1 = 1,\ j_1 = 1,\ k_1 = 1$ in $f_2(i_1, j_1, k_1)$
function $f_2(i_2, j_2, k_2) =$
  if $k_2 < 100$ then $f_3(i_2, j_2, k_2)$ else $f_4(i_2, j_2, k_2)$
function $f_3(i_3, j_3, k_3) =$
  if $j_3 < 20$ then $f_5(i_3, j_3, k_3)$ else $f_6(i_3, j_3, k_3)$
function $f_4(i_4, j_4, k_4) = j_4$
function $f_5(i_5, j_5, k_5) =$
  let $j_8 = i_5,\ k_8 = k_5 + 1$ in $f_7(i_5, j_8, k_8)$
function $f_6(i_6, j_6, k_6) =$
  let $j_9 = k_6,\ k_9 = k_6 + 1$ in $f_7(i_6, j_9, k_9)$
function $f_7(i_7, j_7, k_7) = f_2(i_7, j_7, k_7)$

This gives us some insight into what, exactly, is a "ϕ-function." Compare the expression $j_2 \leftarrow \phi(j_7, j_1)$ (in the really crude SSA program) with the function-declaration

$$f_2(\ldots, j_2, \ldots) = \ldots$$

and function-calls

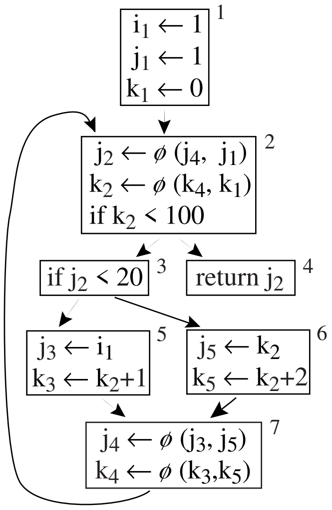
$$f_2(\ldots, j_7, \ldots) \quad f_2(\ldots, j_1, \ldots)$$

in the functional program. We see that the *left-hand side* of the ϕ assignment is the *formal parameter* of the corresponding function; and each *right-hand side* argument of the ϕ assignment is the *actual parameter* of some call to the corresponding function. That's what I mean when I say that SSA form is a kind of functional programming. The "ϕ-functions" are not really functions, but they do correspond (in an inside-out way) to the real functions.

We can express this functional program in a nicer way using the idea of nested scope. Then the inner-nested functions won't all need so many parameters; they can use non-local variables from the functions in which they are nested. This idea will be familiar to Pascal programmers (and Scheme, ML, Haskell programmers), and (if there are any of you left) Algol-60 programmers as well.

let $i_1 = 1,\ j_1 = 1,\ k_1 = 0$
 in let function $f_2(j_2, k_2)$=
   if $k_2 < 100$
   then let function $f_7(j_4, k_4) =$
     $f_2(j_4, k_4)$
    in if $j_2 < 20$
     then let $j_3 = i_1,\ k_3 = k_2 + 1$
      in $f_7(j_3, k_3)$
     else let $j_5 = k_2,\ k_5 = k_2 + 1$
      in $f_7(j_5, k_5)$
   else return j_2
 in $f_2(j_1, k_1)$

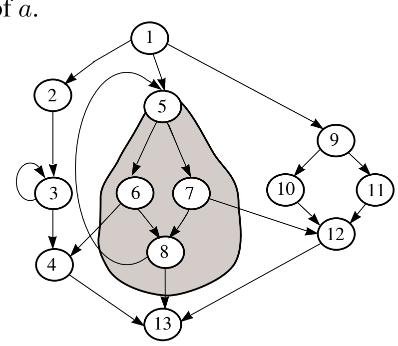
But what's the algorithm for finding the best way of nesting the functions to eliminate unnecessary argument-passing? The algorithm is the one for converting programs to SSA form!



This is the Static Single-Assignment form of the program with optimal placement of ϕ-functions. It's much nicer than the crude version that had too many variables and too many ϕ-functions. This program has "the right number of names." And notice how it corresponds exactly to the nested functional program – function f_i corresponds to block i, parameter j_i corresponds to variable j_i, and so on. Wherever there is a formal parameter of a function (in the functional form), there is a ϕ (in the SSA form). Wherever the functional form refers to a non-local variable, the SSA form has avoided the need for a ϕ.

Algorithm for optimal placement of ϕ's. The only place we really need a ϕ-function in SSA form is where two different definitions reach (along control-flow edges) the same point. For example, in the original CFG (the first diagram above), only one definition of i reaches block 2, so we don't need a ϕ-function for i in that block. This is true even though there are two edges leading into block 2 – it's because the definition of i (in block 1) *dominates* block 2. Any path to block 2 must go through block 1.

We use the notion of dominance and *dominance frontiers* to calculate the minimum set of ϕ-functions. In general, node a in a flowgraph dominates node b when any path from the start node to b must go through a. Now, consider the region of the graph dominated by a; imagine that this region has a "border" or "frontier" separating it from the rest of the graph. We call this the dominance frontier of a. In particular, whenever there is an edge $b \rightarrow c$ from a node b dominated by a to a node c not strictly dominated by a, we say that c is in the dominance frontier of a.



For example, in this graph node 5's dominated region is shown in grey, and the border of that region is crossed by edges $6 \rightarrow 4$, $8 \rightarrow 5$, $8 \rightarrow 13$, and $7 \rightarrow 12$. So we say that nodes $4, 5, 12, 13$ form the dominance frontier of node 5.

The importance of dominance frontiers is this: If node 5 contains a definition of variable x, then any node in the dominance frontier of 5 is reachable from two different definitions of x; one in node 5 and one in the start node. (We assume that every variable has an initializing definition in the start node.) Therefore, the rule for placing ϕ functions is: *Whenever node n contains a definition of some variable x, then any node in the dominance frontier of n needs a ϕ-function for x.*

Efficient algorithms for computing the dominator tree and dominance frontiers can be found in any good compiler textbook [3, 4, 5, 10, 15]

Once we have the SSA form, we can make appropriate linked data structures connecting the uses of each variable to the definition, and the definition to all the uses. Then we can run efficient optimization algorithms: instead of using costly bit-vector dataflow analysis, we can follow links to quickly find the uses for each definition, and vice versa, as needed.

Functional programming in Fortran? So now we know that the SSA conversion algorithm is really doing with its dominance frontiers: it is automatically converting a Fortran or C procedure into a well-structured functional program with nested scope. Actually, I've only shown what to do with the scalar variables. Arrays are handled in high-powered (parallelizing) compilers using sophisticated dependence analysis techniques [15], which is another way of extracting the functional program hiding inside the imperative one.

What SSA users can learn from functional programming. An important property of SSA form is that the definition of a variable dominates every use (or, in the case of a uses within a ϕ-function, dominates the a predecessor of the use node). This property is often unstated in explanations of SSA, but it is necessary for many of the analyses and optimizations on SSA – it is part of SSA's semantics. In a functional program with nested scope, this restriction is explicitly and statically encoded into the structure of function nesting. The notion of *scopes of variables* helps us to structure the intermediate form.

What functional programmers can learn from SSA. People who use SSA tend to draw flowcharts with boxes, assignments, conditionals, and control-flow edges. This notation, while subject to abuse, is often better for explaining ideas and for intuitive visualization of algorithms and transformations. Functional programmers often get lost in the notation of functional programming, which is a shame.

History and literature. SSA form was developed by Wegman, Zadeck, Alpern, and Rosen [1, 11] for efficient computation of dataflow problems such as global value numbering, congruence of variables, aggressive dead-code removal, and constant propagation with conditional branches [14]. Cytron et al. [7] describe the efficient computation of SSA form using dominance frontiers.

Wolfe [15] describes several optimization algorithms on SSA (which he calls *factored use-def chains*).

Church [6] invented λ-calculus, a language of functions with nested scope. Strachey [13] showed how to encode control flow as function calls to *continuation* functions. Steele [12] showed how to use continuations as the intermediate representation of a compiler. Kelsey [9] showed the correspondence between SSA and continuation-passing style (CPS), and gave algorithms for converting each to the other.

Appel [2] improved upon CPS by binding every nontrivial value explicitly to a variable. Flanagan et al. [8] showed Administrative-Normal Form (A-Normal Form or ANF), which binds every nontrivial value to a variable without being full CPS. The functional notation I have used in this paper is a variant of ANF or CPS.

Acknowledgment. Kenneth Zadeck improved my understanding of SSA form through many conversations, and told me all along that SSA is a functional program.

References

[1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 1–11, New York, January 1988. ACM Press.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[3] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, New York, 1998.

[4] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, 1998.

[5] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, New York, 1998.

[6] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, New York, 1993. ACM Press.

[9] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations*, vol. 30, pages 13–22, March 1995.

[10] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[11] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pp. 12–27, New York, Jan. 1988.

[12] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[13] C. Strachey and C. Wadsworth. Continuations: A mathematical semantics which can deal with full jumps. Technical Monograph PRG-11, Programming Research Group, Oxford University, 1974.

[14] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, 1991.

[15] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, Redwood City, CA, 1996.

Andrew Appel is Professor of Computer Science at Princeton University. His research interests include programming languages and compilers, functional programming, and language support for modularity and security.