

# Program Analysis with Presburger Integer Formulae

Josh MacDonald

**Abstract.** A procedure employing Presburger integer formulae for a constraint style program analysis is described. Presburger integer constraints are used to locate dead code fragments in a style similar to *conditional constant propagation* after conversion to a form similar to SSA. Then, an algorithm for proving strong loop termination is presented based on the generation of constraints on the program in this form after reducing the loop's variation to Presburger relations.

## 1 Introduction

First discovered by Presburger in 1929, there exists a decidable subset of logic over the integers with addition. Oppen proved the best known deterministic time for an algorithm originally published by Cooper [1, 5]. The decision procedure has a time complexity of  $2^{2^{2^{pn}}}$  where  $n$  is the length of the sentence and  $p$  is a constant greater than 1.

To gain an understanding of the decision procedure, a naïve implementation of Oppen's algorithm was first tested in SML. Preliminary results suggested that this implementation did in fact exhibit the intractable nature of the algorithm. The SML implementation was replaced with a higher-performance implementation due to Pugh [8, 9]. Pugh's library, known as *Omega*, turned out to be quite practical. Current work in the field of formal verification has turned up even more effective methods [10].

The procedure described here employs Presburger formulae for a constraint style program analysis, inspired by the set and term constrain system employed in Bane [3]. Presburger integer constraints are used to locate dead code fragments in a style similar to *conditional constant propagation* after conversion to a form similar to the static single assignment (SSA) form which I will call the conditional static single assignment (CSSA) form [2, 11]. Then, an algorithm for proving strong loop termination is presented based on the generation of constraints on the program in CSSA form after reducing the loop's variation to Presburger relations.

## 2 Presburger Formulae

A Presburger *term* is defined as follows:

$$t = 0 \mid |x|t + t \mid -t \tag{1}$$

A Presburger *atom* is defined as follows:

$$a = t < t \tag{2}$$

A Presburger *formula* is defined as follows:

$$f = a | \exists x. f | f \wedge f | f \vee f | \neg f \tag{3}$$

From these, formulas containing  $=, \geq, \leq, >, \forall, \subseteq, true,$  and  $false,$  can be generated. A constant integer  $k$  represents  $(1 + \dots + 1)$  repeated  $k$  times. Multiplication by a constant,  $kt,$  is represented as  $(t + \dots + t)$  repeated  $k$  times. Finally, the *stride* constraint:  $k|t$  abbreviates  $\exists x.(x + \dots + x) = t.$

The decision procedure for Presburger formulae may be implemented by a quantifier elimination, though most toolkits use more advanced techniques. Omega extends its capabilities with other, non-decidable features in its interface [6, 7]. I have not used any of these features.

### 3 The SSA Form

The SSA form of a program is one in which program variables have been labeled with an identifier indicating the unique assignment which yields the value of the variable at each program point. Where two program control edges join, a  $\phi$  function is used to join the value of the variable in each branch, and is considered a assignment. Figure 1 contains an example of a program and its SSA form.

$\begin{aligned} &\text{if } (i < k) \{ \\ &\quad i = 3k; \\ &\} \text{ else } \{ \\ &\quad k = 3i; \\ &\} \end{aligned}$	$\begin{aligned} &\text{if } (i_0 < k_0) \{ \\ &\quad i_1 = 3k_0; \\ &\} \text{ else } \{ \\ &\quad k_1 = 3i_0; \\ &\} \\ &i_2 = \phi(i_0, i_1); \\ &k_2 = \phi(k_0, k_1); \end{aligned}$
---	---

Figure 1: Simple program and its SSA form

Algorithms for computing the SSA form are given in Cytron *et al*, and are based on the concept of a *dominance frontier* [2]. A node  $X$  in the control flow graph of a program dominates another node,  $Y,$  if all paths leading to  $Y$  must pass through  $X.$  Domination is both reflexive and transitive. If  $X$  dominates  $Y$  and  $X \neq Y$  then  $X$  strictly dominates  $Y.$  The *immediate dominator* of  $Y$  is the closest strict dominator of  $Y$  on any path leading to  $Y.$  A *dominance frontier*  $DF(X)$  of a node  $X$  is the the set of all nodes  $Y$  such that  $X$  dominates a predecessor of  $Y$  but does not strictly dominate  $Y.$  Intuitively,  $DF(X)$  are the nodes which join a path which  $X$  dominates with another flow of control. SSA places  $\phi$  functions for a  $X$  at the dominance frontier of every assignment to  $X.$  Since a  $\phi$  function introduces a new assignment, this process must be iterated until a least

solution is found. The dominance frontiers of a graph can be computed in  $O(E\alpha(E, N))$ , or by a more sophisticated algorithm  $O(E)$  [2]<sup>1</sup>.

Though the SSA form of a program can be exponential in the program's size, Cytron *et al* give empirical results stating that the size of the minimal SSA program computed by their algorithm and the time to compute it are typically linear in the program size.

The SSA construction is done in three steps:

- a. Construct a *dominance frontier*.
- b. Locate  $\phi$ -functions.
- c. Rename variables.

The next section develops a similar form, CSSA, the construction of which is a modification of the SSA construction.

#### 4 The CSSA Form

The CSSA form arises in this context because for the purposes of integer constraint analysis, program variables are not values but rather sets of possible values. Let the set of possible values for a variable  $x$  be written  $\psi(x)$ . Branches in control flow typically constrain the possible values of program variables, so these constrained variables must be considered distinct. For this reason, variables used in a predicate that branches control flow are renamed much like an assignment in the SSA formulation. Figure 2 contains an example of a program and its CSSA form. The difference to note between the SSA example, is that the variables  $i_0$  and  $k_0$  are relabeled  $i_1, k_1$  in the true branch and  $i_3, k_3$  in the false branch. The relabeling of a variable after this control split is written:

$$\mathbf{x}_1 = \gamma_P(\mathbf{x}_0) \Rightarrow \psi(\mathbf{x}_1) \subseteq \psi(\mathbf{x}_0) \wedge P(\mathbf{x}_1) \quad (4)$$

$$\mathbf{x}_2 = \overline{\gamma}_P(\mathbf{x}_0) \Rightarrow \psi(\mathbf{x}_2) \subseteq \psi(\mathbf{x}_0) \wedge \neg P(\mathbf{x}_1) \quad (5)$$

where  $\mathbf{x}$  is a vector of program variables used in the predicate  $P$ . The relabeled values are at least as constrained as the previous values, but additionally constrained by the result of the predicate.

The CSSA form is constructed by incorporating program dependence. The *program dependence graph* (PDG) [4] introduces a very similar program form to SSA, in which *postdominance*, the dual relation of dominance, defines *control dependency*. A node  $Y$  is control dependent on  $X$  if both of the following hold [2]:

- a. There is a nonnull path  $p : X \xrightarrow{+} Y$  such that  $Y$  postdominates every node after  $X$  on  $p$ .

---

<sup>1</sup>It is perhaps of interest to the enthusiastic Bane reader, given the various attempts to do a control flow analysis this semester in Bane, to note that much of Cytron's construction is based on a few set formulae which compute the dominance frontier of each program point

<pre> if (<math>i &lt; k</math>) {     <math>i = i - k</math>;     <math>k = k - 3</math>; } else {     <math>k = k - i</math>;     <math>i = i - 3</math>; } </pre>	<pre> if (<math>i_0 &lt; k_0</math>) {     <math>i_1 = \gamma_{i_0 &lt; k_0}(i_0)</math>;     <math>k_1 = \gamma_{i_0 &lt; k_0}(k_0)</math>;     <math>i_2 = i_1 - k_1</math>;     <math>k_2 = k_1 - 3</math>; } else {     <math>i_3 = \bar{\gamma}_{i_0 &lt; k_0}(i_0)</math>;     <math>k_3 = \bar{\gamma}_{i_0 &lt; k_0}(k_0)</math>;     <math>k_4 = k_3 - i_3</math>;     <math>i_4 = i_3 - 3</math>; } <math>i_5 = \phi(i_2, i_4)</math>; <math>k_5 = \phi(k_2, k_4)</math>; </pre>
--	--

Figure 2: Simple program and its CSSA form

- b. The node  $Y$  does not strickly postdominate the node  $X$ .

Clearly,  $X$  is control dependent on  $Y$  if  $X$  is on the dominance frontier of  $Y$  in the *reverse* graph. The CSSA form is computed by following the SSA transformation with a slightly modified dual transformation; rerun SSA, computing the *control frontier*, reversing the direction of each edge, the sense of read and assign, and using a  $\gamma$ -function rather than a  $\phi$ -function, as above, before each branch in control flow. The only modification to the second application of SSA is to only rename variables which clearly involved in the branch of control flow: those that appear in the branching predicate.

So far, looping constructs have not been considered. Proceed assuming that code is loop-free.

**THEOREM 1.** *After the CSSA transformation, in loop-free code, all appearances of a variable represent the same set of possible values.*

$$x = y \Rightarrow \psi(x) \equiv \psi(y)$$

**PROOF.** Since  $x$  and  $y$  are the same variable, they must have resulted from the same, unique assignment as a result of the SSA transformation. Suppose the reference to  $x$  is at node  $X$ ,  $y$  at  $Y$ , and the assignment at  $A$ . For  $x$  and  $y$  to have different constrained values, they must reside on control paths which constrain their values differently. The *control frontier* of the two references,  $x$  and  $y$  is that predicate, and since the assignment uniquely determines both  $x$  and  $y$  the assignment must dominate the predicate. Thus by the *postdominance* condition, were  $x$  and  $y$  differently constrained,  $x$  and  $y$  would have been labeled differently by the *reverse*-SSA.

□

## 5 Conditional Constraint Propagation

With the program in the above CSSA form, detection of dead code with Presburger constraints is straightforward. For each assignment, generate an equality constraint relating the assigned variable to the assigned value. For each predicate and set of  $\gamma$ -functions, generate constraints as in (1) and (2). Done in the order of program evaluation, predicates with definite, static values can be determined and dead code can be announced, skipped, or eliminated. For each function  $z = \phi(x, y)$ , generate a constraint  $z = x \vee z = y$ .

## 6 Loop Reduction

Though none of the preceding discussion has mentioned the analysis of loops, treating the program in terms of general control flow graphs has paid off. The CSSA construction still yields the a program in which each program variable has the value of a unique assignment, and in which no program variable is control dependent, but theorem 1 does not apply to loops. Consider the following code sample in SSA form:

```
 $x_0 = 0;$ 
loop {
   $x_1 = \phi(x_0, x_2);$ 
  if ( $x_1 = 16$ ) break;
   $x_2 = x_1 + 2;$ 
}
```

Figure 3: Simple loop and its SSA form

Now translate to CSSA form:

```
 $x_0 = 0;$ 
loop {
   $x_1 = \phi(x_0, x_2);$ 
  if ( $x_1 = 16$ ) break;
   $x_3 = \overline{\gamma}_{x=16}(x_1);$ 
   $x_2 = x_3 + 2;$ 
}
```

Figure 4: Simple loop and its CSSA form

Since the variables do not encode the number of times the loop body has been executed, the variables cannot be said to represent the same set of possible values. It is sometimes possible to express the value of a variable as a function of the number of times through the loop and break the cyclic structure of the program. These cases are examined.

The  $\phi$ -functions encode the looping nature of variable constraints. To begin with, introduce a new variable  $j \geq 0$  to count loop iterations. After putting the loop body into CSSA form as in figure 4, examine each  $\phi$  in the order they are encountered. If the assignment  $C = \phi(A, B)$  is encountered where  $B$  was assigned *before* the  $\phi$  and  $A$  is assigned *after*, temporarily unconstrain  $C$  ( $C$  becomes temporarily free, the  $\phi$  function is removed) and continue. In the opposite order that the  $\phi$  functions were discovered, compute the difference  $\Delta = A - C$  using a Presburger solver.

- a. If  $\Delta$  is a single valued constant, then replace  $C = \phi(A, B)$  with  $C = B + \Delta j$ .
- b. If  $\Delta \geq 0$ , then replace  $C = \phi(A, B)$  with the constraint  $C \geq B$ .
- c. If  $\Delta \leq 0$ , then replace  $C = \phi(A, B)$  with the constraint  $C \leq B$ .
- d. Otherwise, unconstrain  $C$ .

LEMMA 1. *Unconstraining a variable is sound. Replacement (d) is sound.*

Unconstraining a variable only weakens any conclusions that may be made from the system of constraints.

LEMMA 2. *If there are no  $\phi$ -functions referring to forward variables on a path from entry to a node  $X$ , theorem 1 is valid.*

$\phi$ -functions referring to forward variables introduce uncertainty in the number of iterations.

LEMMA 3. *Replacement (a) is sound.*

If the system can prove a constant increment or decrement for the variable, then the value of that variable may be expressed as multiplication by the loop increment.

LEMMA 4. *Replacements (b) and (c) are sound.*

If the variable's loop increment is positive or negative, semi-definite, these variables will be always increasing or decreasing.

THEOREM 2. *The resulting system of constraints is correct.*

PROOF. Follows from the preceding lemmas and theorem 1.

□

## 6.1 Testing Loops for Strong Correctness

The resulting system of constraints can be used to generate a formula which is satisfiable if and only if the loop is *strongly correct*, that is, the loop is guaranteed to terminate.

By inserting an artificial node at each exit point of a loop which claims to reference all variables,  $\gamma$ -functions will be inserted at each exit point with values constrained to exit the loop. For all program variables in scope on entrance to the loop, assert that one of the exit conditions must be met.

## 7 Future Work

This work is hardly complete. A prototype system is currently implemented which locates dead code in non-looping code using Omega and a *non-minimal* CSSA construction. With this system I proved absolutely nothing. Only my contrived examples contained any dead code—these Omega proved satisfiable or unsatisfiable quickly (on the order of tens of milliseconds for equations with 50-100 nodes).

After this was completed, I began writing this paper to make sure I had a clear understanding of what to do with loops, and that took longer than expected, partly because I had to go invent a program representation (which was quite new to me, though in hindsight it looks fairly obvious).

This work is (hopefully) being demonstrated at the Design Automation Conference (DAC) on June 15th, so I should have some data by then.

## 8 Conclusion

Very sketchy, preliminary results using Omega suggest that it is practical for analysis of local integer variables in Java programs. A very convenient program representation was developed for generating these types of constraints, and a conservative loop reduction was presented which allows certain forms of loops to be proven terminating. It has not been tested.

## References

- [1] COOPER, D. C. *Machine Intelligence*, vol. 7. American Elsevier, 1972, ch. Theorem Proving in Arithmetic without Multiplication, pp. 91–99.
- [2] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct. 1991), 451–490.
- [3] FÄHNDRICH, M. Bane: Analysis programmer interface, Apr. 1998.
- [4] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349.
- [5] OPPEN, D. C. A  $2^{2^{2^n}}$  upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences* 16, 3 (June 1978), 323–332.
- [6] PUGH, W. The omega test: a fast and practical integer programming algorithm for dependence analysis. Technical report, University of Maryland, College Park, Aug. 1992.
- [7] PUGH, W. Counting solutions to presburger formulas: How and why. Technical Report CS-TR-3234, University of Maryland, College Park, Apr. 1993.

- [8] PUGH, W., KELLY, W., MASLOV, V., ROSSER, E., SHPEISMAN, T., AND WONNACOT, D. The omega calculator and library, version 1.1.0, Nov. 1996.
- [9] PUGH, W., KELLY, W., MASLOV, V., ROSSER, E., SHPEISMAN, T., AND WONNACOT, D. The omega library, version 1.1.0, interface guide, Nov. 1996.
- [10] QUADEER, S. Personal communication. May 1998.
- [11] WEGMAN, M. N., AND ZADECK, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* 13, 2 (Apr. 1991), 181–210.