Range Analysis of Whole Programs

Homer Sympson Luck Skywalker Robin Hood Frodo Beggins
Computer Science Department – UFMG – Brazil
{hsympson,skywalker,rbhood,frodo}@dcc.ufmg.br

Abstract

This paper describes a range analysis algorithm that is linear on the program size. We handle comparisons between variables via a two-phases dataflow analysis, and achieve flow sensitiveness by using the Extended Static Single Assignment form as the intermediate representation. We have implemented an inter-procedural version of our technique into LLVM, an industrial strenght compiler.

Categories and Subject Descriptors D - Software [*D.3 Programming Languages*]: D.3.4 Processors - Optimization

General Terms Languages, Performance, Experimentation

Keywords Range Analyisis, Dataflow Analysis, Integer arithmetics

1. Introduction

2. Background

Following Gawlitza *et al.*'s notation, we shall be performing arithmetic operations over the complete lattice $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$, where the ordering is naturally given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. For any $x > -\infty$ we define:

$x + \infty = \infty, x \neq -\infty$ $\quad$ $x - \infty = -\infty, x \neq +\infty$
$x \times \infty = \infty$ if $x > 0$ $\quad$ $x \times \infty = -\infty$ if $x < 0$
$0 \times \infty = 0$ $\quad$ $(-\infty) \times \infty =$ not defined

From the lattice $\mathcal{Z}$ we define the product lattice $\mathcal{Z}^2$, which is partially ordered by the subset relation $\sqsubseteq$. $\mathcal{Z}^2$ is defined as follows:

$$\mathcal{Z}^2 = \emptyset \cup \{[z_1, z_2] |\ z_1, z_2 \in \mathcal{Z},\ z_1 \leq z_2,\ -\infty < z_2\}$$

Given an interval $\iota = [l, u]$, we let $\iota_\downarrow = l$, and $\iota_\uparrow = u$. We let $\mathcal{V}$ be a set of variables, and $I : \mathcal{V} \mapsto \mathcal{Z}^2$ a mapping from variables to intervals in $\mathcal{Z}^2$. Our objective is to solve a constraint system C, formed by instances of the five different types of constraints seen in Figure 1 (left). Figure 1 (right) defines a valuation function e that computes $Y = f(\ldots)$, given I. Armed with these concepts, we define the range analysis problem as follows:

DEFINITION 2.1. RANGE ANALYSIS PROBLEM
Input: *a set C of constraints ranging over a set $\mathcal{V}$ of variables.*
Output: *a mapping I such that, for any variable $V \in \mathcal{V}$, e(V) = I[V].*

$Y = [l, u]$	$e(Y) = [l, u]$
$Y = \phi(X_1, X_2)$	$\dfrac{I[X_1] = [l_1, u_1] \quad I[X_2] = [l_2, u_2]}{e(Y) = [\min(l_1, l_2), \max(u_1, u_2)]}$
$Y = X_1 + X_2$	$\dfrac{I[X_1] = [l_1, u_1] \quad I[X_2] = [l_2, u_2]}{e(Y) = [l_1 + l_2, u_1 + u_2]}$
$Y = X_1 \times X_2$	$\dfrac{\begin{array}{c} I[X_1] = [l_1, u_1] \\ I[X_2] = [l_2, u_2] \quad L = \{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\} \end{array}}{e(Y) = [\min(L), \max(L)]}$
$Y = aX + b$	$\dfrac{I[X] = [l, u] \quad k_l = al + b \quad k_u = au + b}{e(Y) = [\min(k_l, k_u), \max(k_l, k_u)]}$
$Y = X \sqcap [l', u']$	$\dfrac{I[X] = [l, u]}{e(Y) \leftarrow [\max(l, l'), \min(u, u')]}$

Figure 1. The valuation function.

We will use the program in Figure 2(a) as the running example to illustrate our range analysis. Figure 2(b) shows the same program in SSA form, and Figure 2(c) outlines the constraints that we extract from this program. There is a clear correspondence between instructions and constraints. A possible solution to the range analysis problem is given in Figure 2(d). This is a very conservative solution. As we will see shortly, we can improve this solution substantially by using a more sophisticated program representation.

3. Polynomial Time Range Analysis

In this section we explain the algorithm that we use to solve the range analysis problem for an entire program. This algorithm involves a number of steps. First, we convert the program to a suitable intermediate representation that makes it easier to extract constraint. From these constraints, we build a dependence graph that allows us to do range analysis sparsely. Finally, we solve the constraints applying different fix-point iterators on this dependence graph. Figure 3 gives a global view of this algorithm. Some of the steps in the algorithm are optional. They improve the precision of the range analysis, at the expenses of a longer running time.

3.1 Choosing a Program Representation

The solution that we found to the range analysis problem in Figure 2 is imprecise because we did not take conditional tests into considerations. Branches gives us information about the ranges that some variables assume, but only at *specific* program points. For instance, given a test such as $(k_1 < 100)$? in Figure 2(b), we know that $I[k_1] \sqsubseteq [-\infty, 99]$ whenever the condition is true. In order to encode this information, we might split the live range of k_1 right after the branching point; thus, creating two new variables, one at

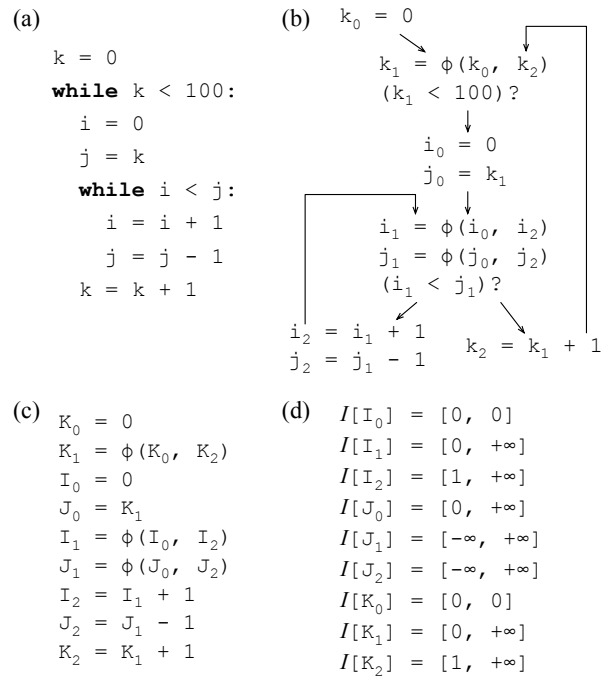


Figure 2. (a) Example program. (b) Control Flow Graph (CFG) in SSA form. (c) Constraints that we extract from the program. (d) A possible solution to the range analysis problem.

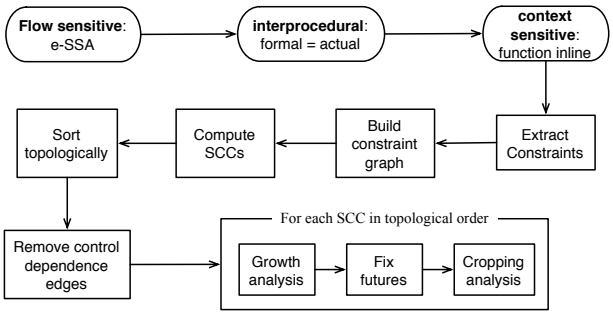


Figure 3. Our algorithm to solve the range analysis problem. Rounded boxes are optional steps.

the path where the condition is true, and another where it is false. There is a program representation, introduced by Bodik *et al.* [2], that performs this live range splitting: the *Extended Static Single Assignment* form, or e-SSA for short. In this paper, we have experimented with two different flavors of e-SSA form, which we call *standard* and *non-pruned*.

We will show first how we convert a program into standard e-SSA form. Let $(v < c)?$ be a conditional test, and let l_t and l_f be labels in the program, such that l_t is the target of the test if the condition is true, and let l_f is the target when the condition is false. If l_f dominates any use of v, then we insert at l_f a copy $v_f = v \sqcap [-\infty, c-1]$, where v_f is a fresh name. We then rename every use of v that is dominated by l_f to v_f. Dually, if l_t dominates any use of v, then we insert at l_t a copy $v_t = v \sqcap [c, +\infty]$, and rename variables

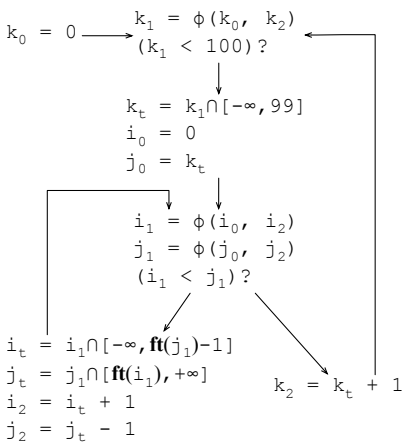


Figure 4. The control flow graph from Figure 2(b) converted to standard e-SSA form.

accordingly. If the test uses two variables, e.g., $(v_1 < v_2)?$, then we create intersections bound to *futures*. We insert, at l_f, $v_{1f} = v_1 \sqcap [\mathbf{ft}(v_2), +\infty]$, and $v_{2f} = v_2 \sqcap [-\infty, \mathbf{ft}(v_1) - 1]$. Similarly, at l_v we insert $v_{1v} = v_1 \sqcap [-\infty, \mathbf{ft}(v_2) - 1]$ and $v_{2f} = v_2 \sqcap [\mathbf{ft}(v_1), +\infty]$. We use the notation $\mathbf{ft}(v)$ to denote the *future* bounds of a variable. As we will show in Section 3.5, once the growth pattern of v is known, we can replace $\mathbf{ft}(v)$ by an actual value. Once we are done placing copies, we insert ϕ-functions into the transformed program to convert it to SSA form. This last step prevents that two different names given to the same original variable be simultaneously alive at the program code. Figure 4 shows our running example changed into standard e-SSA form.

The standard e-SSA form serves well analysis that need range information at the use site of variables, such as conditional constant propagation, redundant code elimination and detection of buffer overflows. On the other hand, there exist compilation algorithms that require range information at the whole live range of variables. Examples of such algorithms include the family of bitwidth aware register allocators [1, 7, 9] and hardware synthesizers [3, 6, 8]. In order to provide extra precision to these algorithms, we may work with a non-pruned version of e-SSA form, which we produce by simply inserting copies after every conditional test, and then reconverting the entire program to SSA form. Figure 5 illustrates the difference between these two program representations. Notice that the non-pruned flavor might introduce dead variables in the program code, which can be removed by standard dead code elimination.

3.2 Extracting Constraints

The constraints shown in Figure 1 let us handle many different assembly instructions. We use ϕ assignments, such as $Y = \phi(X_1, X_2)$ to represent the ϕ-functions typically found in SSA form programs. As we have discussed in Section 3.1, we use intersection constraints to bound the variables created due to live range splitting at conditionals. The widening analysis that we will introduce in Section 3.5 require monotonic transfer functions. Many assembly operations, such as modulus or division, do not afford us this property. However, several of these non-monotonic instructions have a conservative translation to our constraint system, as we show in Figure 6.

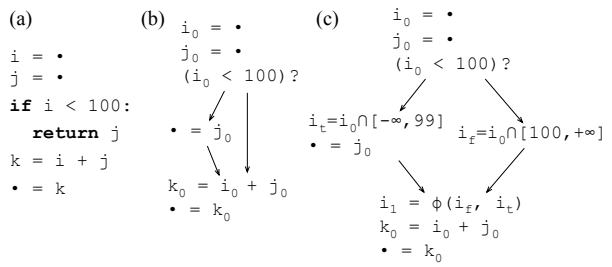


Figure 5. (a) Example program. (b) Standard e-SSA form. (c) Non-pruned e-SSA form.

Description	Operation	Constraint
Constant	$v = c$	$V = [c, c]$
Assignment	$v_1 = v_2$	$V_1 = V_2$
Multiplication	$v_1 = v_2 * c$	$V_1 = cV_2$
Int. division	$v_1 = v_2 \,/\, v_3$	$V_1 = V_2$
		$V_1 = -V_2$
Modulus	$v_1 = v_2 \,\%\, c$	$V_1 = [0, c-1]$
Bitwise and	$v_1 = v_2 \,\&\, c$	$V_1 = v_2 \sqcap [0, c]$
Bitwise or	$v_1 = v_2 \mid v_3$	$V_1 = V_2 + V_3$
Left shift	$v_1 = v_2 \ll c$	$V_1 = 2^c V_2$

Figure 6. Example of constraint derivation rules. We let c denote a positive constant, $\{v, v_1, v_2\}$ are program variables and $\{V, V_1, V_2\} \subset \mathcal{V}$.

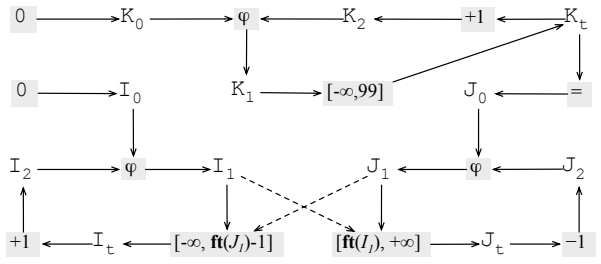


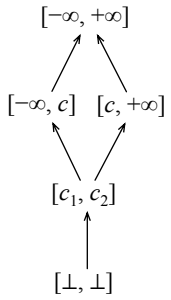
Figure 7. The dependence graph that we build to the program in Figure 4.

3.3 The Dependence Graph

The main data structure that we use to solve the range analysis problem is a variation of Ferrante *et al.*'s *program dependence graph* [5]. For each constraint variable V we create a variable node N_v. For each constraint C we create a constraint node N_c. We add an edge from N_v to N_c if the name V is used in C. We add an edge from N_c to V if the constraint C defines the name V. Figure 7 shows the dependence graph that we build for the e-SSA form program given in Figure 4. If V is used by C as the input of a future, then the edge from N_v to N_c represents a *control dependence* [5, p.323]. All the other edges denote *data dependences* [5, p.322].

3.4 Finding Ranges: the Macro Algorithm

Having built the dependence graph, we proceed to solve the range analysis problem by finding all the strongly connected components (SCCs) of the dependence graph, and reducing this graph to a directed acyclic graph. We perform this last step by collapsing every SCC into a single node. We then sort the resulting DAG topologically, and apply the analyses from Section 3.5 on every SSC in topological order. Once we solve the range analysis problem for a SCC, we propagate the intervals that we found to the variable nodes at the *frontier* of this SCC. A variable node N_v is said to be in the frontier of a strongly connected component S if: (i) $N_v \notin S$; and (ii) there exists a variable node $V' \in S$, and a constraint node N_c, such that $V' \leftarrow N_c$, and $N_c \leftarrow V$. This propagation gives us a very modular algorithm: when analyzing a strongly connected component S we can rest assured that any influence that S might suffer from nodes outside it has been already taken into consideration.



$$\frac{I[Y] = [\bot, \bot]}{I[Y] \leftarrow e(Y)}$$

$$\frac{e(Y)_\downarrow < I[Y]_\downarrow \qquad e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [-\infty, +\infty]}$$

$$\frac{e(Y)_\downarrow < I[Y]_\downarrow}{I[Y] \leftarrow [-\infty, I[Y]_\uparrow]}$$

$$\frac{e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, +\infty]}$$

Figure 8. (Left) The lattice of the widening analysis. (Right) Cousot and Cousot's widening operator. We evaluate the rules from top to bottom, and stop upon finding a pattern matching.

3.5 Finding Ranges: the Micro Algorithm

Given a strongly connected component of the dependence graph with N nodes, we solve the range analysis problem in three-steps:

1. Run widening analysis: $O(N)$.
2. Fix intersections: $O(N)$.
3. Run narrowing analysis: $O(N^2)$.

However, before we start, we remove the control dependence edges from the strongly connected component, as they have no semantics to our transfer functions.

Widening Analysis. The first step of our algorithm consists in determining the growth behavior of the interval bound to each constraint variable. We accomplish this task via Cousot and Cousot's widening operator [4, p.247]. The possible behaviors of an interval are: (i) does not change; (ii) grows towards $+\infty$; (iii) grows towards $-\infty$; and (iv) grows in both directions. The lattice of this dataflow analysis, plus its meet operator is given in Figure 8. Because the lattice has height three, the intervals bound to each variable can change at most three times.

Fixing futures. The ranges found by the widening analysis tells us which variables have fixed bounds, independent on the intersections in the constraint system. Thus, we can use actual limits to replace intersections bounded by futures. Figure 9 shows the rules to perform these substitutions. In order to correctly replace a future $\mathbf{ft}(V)$ that limits a variable V', we need to have already applied the widening analysis onto V. Had we considered only data dependence edges, then it would be possible that V be analyzed before V'. However, because of control dependence edges, this case cannot happen. The control dependence edges ensure that any topological ordering of the constraint graph either places N_v before $N_{v'}$,

$$\frac{Y = X \sqcap [l, \mathbf{ft}(V) + c] \qquad I[V]_\uparrow = u}{Y = X \sqcap [l, u + c]}, u, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

$$\frac{Y = X \sqcap [\mathbf{ft}(V) + c, u] \qquad I[V]_\downarrow = l}{Y = X \sqcap [l + c, u]}, l, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

Figure 9. Rules to replace futures by actual bounds. S is the interval bound to each variable after the widening analysis.

$$\frac{I[Y]_\downarrow = -\infty \qquad e(Y)_\downarrow > -\infty}{I[Y] \leftarrow [e(Y)_\downarrow, I[Y]_\uparrow]}$$

$$\frac{I[Y]_\downarrow > e(Y)_\downarrow}{I[Y] \leftarrow [e(Y)_\downarrow, I[Y]_\uparrow]}$$

$$\frac{I[Y]_\uparrow = +\infty \qquad e(Y)_\uparrow < +\infty}{I[Y] \leftarrow [I[Y]_\downarrow, e(Y)_\uparrow]}$$

$$\frac{I[Y]_\uparrow < e(Y)_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, e(Y)_\uparrow]}$$

Figure 10. Cousot and Cousot's narrowing operator.

or places these nodes in the same strongly connected component. For instance, in Figure 7, variables J_1 and I_t are in the same SCC only because of the control dependence edges.

Narrowing Analysis. The widening analysis associates very conservative bounds to each variable. Thus, the last step of our algorithm consists in narrowing these intervals. We accomplish this step via Cousot and Cousot's narrowing operator [4, 248], which we show in Figure 10.

Example Continuing with our example, Figure 11 shows the application of our algorithm on the last strongly connected component of Figure 7. Notice that upon meeting this SCC, we have already found the interval $[0, 0]$ to I_0 and the interval $[100, 100]$ to J_0, as we show in Figure 7(a). We are not guaranteed to find the least fix point of a constraint system. However, in this example we did it. We emphasize that finding this tight solution was only possible because of the topological ordering of the constraint graph in Figure 7. Had we applied the widening operator onto the whole graph, then we would have found out that variable J_0 is bound to $[-\infty, +\infty]$, because (i) it receives its interval directly from variable K_t, which is upper bounded by $+\infty$, and (ii) it is part of a negative cycle. On the other hand, because we only analyze the J's SCC after we have analyzed K's, K only contribute the constant range $[0, 99]$ to J_0.

3.6 Alternative Narrowing Operators

Cousot and Cousot's narrowing operator, given in Figure 10 is not guaranteed to find the least fixpoint of a system of constraints. In order to improve the bounds, trading time for precision, we have experimented with two other operators. The first, is the evaluation function itself, which is given in Figure 1. By iterating it until reaching a fixpoint, we are guaranteed to find the optimal solution to the narrowing analysis. However, this approach is not practical, because it is equivalent to concretely interpreting the program. The second alternative that we have tested consists in applying the evaluation function on each constraint at most once per intersection. We call this approach *depth-first narrowing*, because, starting from an intersection we visit every node that might be influenced by it. After

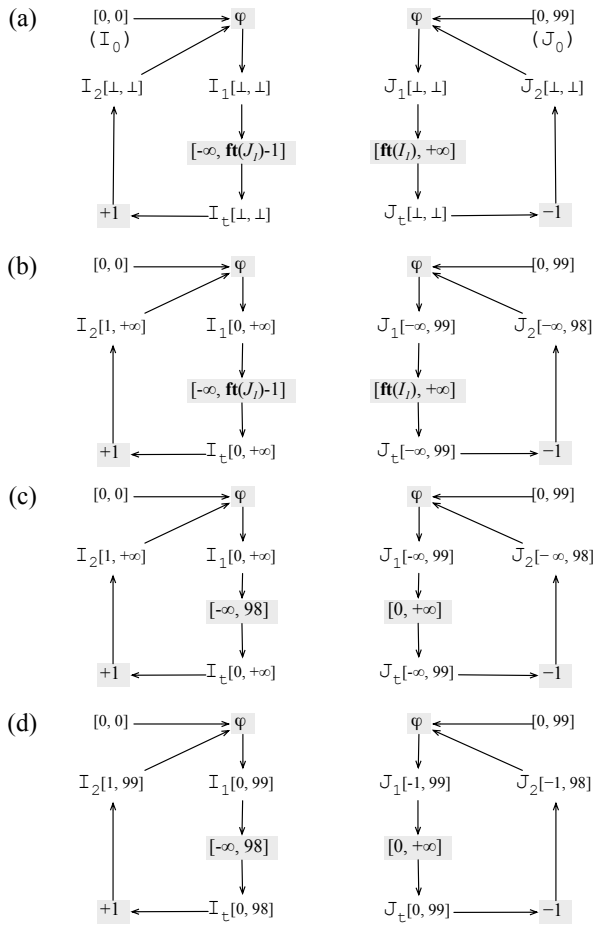


Figure 11. Four snapshots of the last SCC of Figure 11. (a) After removing control dependence edges, immediately before applying Cousot and Cousot's operators. (b) After running the widening analysis. (c) After fixing the intersections. (d) After running the narrowing analysis.

removing control dependence edges, if we are left with a strongly connected component containing only *ascencing constraints*, that is, constraints that only increase the upper bounds of intervals, then this approach is guaranteed to find the least fixpoint. The same is true if we have a SCC with only *descending constraints*.

References

[1] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2006.

[2] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[3] Jason Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu, Zhiru Zhang, and Xu Cheng. Bitwidth-aware scheduling and binding in high-level synthesis. *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2:856–861, 18-21 Jan. 2005.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of

fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[6] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371, 2001.

[7] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226. ACM, 2008.

[8] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.

[9] Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *POPL*, pages 85–96, New York, NY, USA, 2003. ACM.