Static Analysis by Elimination

Pavle Subotic[1], Andrew E. Santosa[2], and Bernhard Scholz[3]

[1] Uppsala University, Sweden
[2] Oracle Labs, Australia
[3] University of Sydney, Australia

Abstract. In the past, elimination-based data flow analysis algorithms have been proposed as an alternative to iterative algorithms for solving dataflow problems. Elimination-based algorithms exhibit a better worst-case runtime performance than iterative algorithms. However, the implementation of elimination-based algorithms is more challenging and iterative algorithms have been sufficient for solving standard data-flow problems in compilers. For more generic abstract interpretation frameworks, it has not been explored whether elimination-based algorithms are useful. In this paper we show that elimination-based algorithms are useful for implementing abstract interpretation frameworks for low-level programming languages. We demonstrate the feasibility of our approach by a range analysis developed in the LLVM framework. We supplement this work by a range of experiments conducted through several test suites.

1 Introduction

Range analysis has various applications in embedded systems *program analysis* including assertion elimination [1], determining the numerical stability of algorithms [6], eliminating array out of bound checks [7], and integer overflow detection [15]. The majority of embedded systems are written in low-level languages such as assembly or C. Implementing a range analysis for low-level languages, is not entirely straightforward in the presence of loops as they are often implemented using goto-like statements. In such cases the static program analysis framework must identify potential program points that may delay the termination of the analysis. Existing methods [4, 6, 2] rely either on structured control-flow imposed by the syntax of a programming language to identify loops, or require additional and complicated control flow analysis to mark program loop headers [11]. In this paper we present a range analysis for the LLVM [10] low-level language which uses an elimination-based data-flow analysis [14]. We demonstrate our technique through an implementation in the industrial strength LLVM compiler framework. We have also conducted experiments with a suite of test programs. Our approach demonstrates that elimination-based data flow algorithms are (1) useful in implementing generic abstract interpretation frameworks, (2) provide a more precise and flexible alternative to iterative schemes, and (3) integrates into compilers with ease as shown with the LLVM compiler framework. The contributions of this paper are as follows: (1) adopting an elimination-based algorithm for an abstract interpretation framework – in particular range analysis, (2) intrinsically discovering widening points and accelerating convergence, and (3) implementing our techniques in the LLVM compiler framework and demonstrating feasibility. The paper is organized as follows: In Section 2 we explain the background, in Section 3 we discuss our approach, in Section 4 we present our results, and in Section 5 we draw our conclusions.

2 Background

Data flow analysis. Data flow analysis [9] is a unified framework for static program analysis. It takes as input a program in the form of a control-flow graph (CFG) and gathers information about the possible set of program values. In this work, we assume that CFG nodes are basic blocks, i.e., a sequence of consecutive statements, and edges represent transfer of control from one basic block to another. A basic block is assigned a unique number i, $(0 \le i \le n-1)$, where n is the number of basic blocks in the graph. The node 0 denotes the start node and all other nodes are denoted by 1 to $n-1$. The set of predecessors for node i is denoted by $pred(i)$. We say a path p is a sequence of basic blocks $0 \dots i \dots n-1$ such that $(i, i+1)$ is an edge in the control-flow graph. A *constraint system* has the following initial form that directly corresponds to the control-flow graph:

$$\begin{gathered} x_0 \sqsupseteq f_0(\{x_k | k \in pred(0)\}), \\ \vdots \\ x_{n-1} \sqsupseteq f_{n-1}(\{x_k | k \in pred(n-1)\}). \end{gathered}$$

The *data-flow variables* x_i denote abstracted program properties at the entry of basic block i, and functions f_i's are monotonic data-flow functions whose arguments are the data-flow variables of the predecessors of i. The symbol $\sqsupseteq$ denote a partial order. Several techniques exist for obtaining a solution for a data-flow constraint system. They can be broadly classified as either iterative [9, 8, 3] or elimination-based approaches [12, 13].

Elimination-based algorithms. Elimination-based algorithms are derivatives of the Gaussian elimination approach for solving constraints. Modern approaches are more efficient than the Gaussian algorithm because they exploit the underlying dependency structure of the simultaneous equation system. Modern approaches include Allen-Cocke interval analysis, Hecht-Ullman T1-T2 analysis, Tarjan interval analysis, Graham-Wegman analysis, which are surveyed in [12], Scholz-Blieberger [13], and Sreedhar-Gao-Lee algorithm [14]. Allen-Cocke interval analysis derives the set of interval graphs and uses them to choose an evaluation order for the data-flow equations. This results in a CFG that leads to simplification. It has a worst-case run-time complexity of $O(n^2)$ compared to $O(n^3)$ for the Gaussian elimination approach, where n is the number of nodes in the graph. Hecht-Ullman T1-T2 analysis non-deterministically substitutes terms in the equations with respect to an ordering. The substitutions are stored and possible common factors are exploited in subsequent calculations. This approach improves on the Allen-Cocke algorithm by having a worst-case complexity of $O(n \log n)$. Tarjan interval analysis imposes linear variable ordering and eliminates variables from the system of equations in that ordering, delaying some calculations; a path-compressed tree is used to remember sequences of reduced equations for these delayed calculations. It has an improved worst-case complexity in $O(n\alpha(n))$ where α is a very slow growing function. Graham-Wegman analysis establishes an order of substitutions for each term in the system that avoids duplication of common substitution sequence calculations. It uses a transformed version of the original graph to remember previous substitutions. It has a worst-case complexity of $O(n \log n)$. The Scholz-Blieberger algorithm [13] uses annotated decomposition trees which describes reducible flowgraphs with means of binary trees. However, their algorithm is limited to reducible flowgraphs.

In this work, we employ the Sreedhar-Gao-Lee algorithm that makes the implementation of elimination algorithms simpler by using a data structure called DJ-graphs [14]. The DJ-graph of a program is the dominator tree of its control-flow graph, we refer to these edges as D-edges. This graph is augmented with join edges called J-edges. In the first phase of the algorithm, J-edges are eliminated in a bottom-up manner, and using a *loop breaking* rule cyclic J-edges are instantiated. The algorithm also performs substitution along D-edges when necessary. At the end of the bottom-up elimination phase, all the J-edges will be eliminated. Once the solution is determined for the root node, this information is propagated in a top-down fashion on the dominator tree to compute the solution for every other node. It has a complexity in $O(m \log n)$ where m is the number of edges and n is the number of nodes in the flow graph. The Sreedhar-Gao-Lee's algorithm has another advantage that it is readily implementable and is therefore employed in our experiments in Section 4. Our approach is however, equally applicable to other algorithms. An elimination-based data-flow solver requires two operations that are applied to the set of constraints, i.e., *substitution* ($sub_{i,j}$) and *loop-breaking* (lb_i). Substitution is defined as a transformation of the set of constraints, where an occurrence of a data-flow variable x_i, in the right-hand side of constraint x_j, is replaced by its associated term. Loop-breaking eliminates the occurrences of variable x_i on the right-hand side of constraint i. Substitution and loop-breaking is applied until a solution for a single variable is found and this solution is back-propagated to determine the solutions of the remaining variables.

To illustrate the elimination-based approach we present an example which consists of three constraints, i.e., $n = 3$, and where we assume a Boolean domain, i.e., variables can assume values in the Boolean lattice. The following constraint system is constructed, which corresponds to a control-flow graph which has a loop between nodes 1 and 2 denoted by the data-flow variables x_1 and x_2:

$$x_0 \sqsupseteq \texttt{false}, \qquad x_1 \sqsupseteq x_0 \wedge x_2, \qquad x_2 \sqsupseteq x_1.$$

By following the Gaussian algorithm, the operation $lb_0(cs)$ is performed in the first loop. This operation has no effect since the constraint $x_0 \sqsupseteq \texttt{false}$ has no recurrence of x_0 on the right-hand side. Within the inner loop, the operations $sub_{0,1}(cs)$, $sub_{0,2}(cs)$ are performed, resulting in the following constraint system:

$$x_0 \sqsupseteq \texttt{false}, \qquad x_1 \sqsupseteq \texttt{false} \wedge x_2, \qquad x_2 \sqsupseteq x_1.$$

In the second iteration of the outer loop $lb_1(cs)$ is performed, which has no effect. However, performing $sub_{1,2}(cs)$ in the inner loop, produces the following constraint system:

$$x_0 \sqsupseteq \texttt{false}, \qquad x_1 \sqsupseteq \texttt{false} \wedge x_2, \qquad x_2 \sqsupseteq \texttt{false} \wedge x_2.$$

In the last iteration of the outer loop, *lb*$_2$*(cs)* is performed. The loop-breaking operation finds a fixpoint solution for x_2, which in this case has the fixpoint solution x_2 = `false`. For this iteration the inner loop is not executed and the program exits the outer loop. At this point we have the following constraint system:

$$x_0 \sqsupseteq \texttt{false}, \qquad x_1 \sqsupseteq \texttt{false} \wedge x_2, \qquad x_2 \sqsupseteq \texttt{false}.$$

We execute the second loop, which executed the operations, *sub*$_{2,1}$*(cs)*, *sub*$_{2,0}$*(cs)*, and *sub*$_{1,0}$*(cs)*. After these operations are executed, we have the following constraint system:

$$x_0 \sqsupseteq \texttt{false}, \qquad x_1 \sqsupseteq \texttt{false} \wedge \texttt{false}, \qquad x_2 \sqsupseteq \texttt{false}.$$

The constraint system now has no variables left and can be solved, giving us the result of:

$$x_0 \sqsupseteq \texttt{false}, \qquad x_1 \sqsupseteq \texttt{false}, \qquad x_2 \sqsupseteq \texttt{false}.$$

Widening and narrowing. We let the widening (∇) and narrowing (Δ) operators be defined as in [5] and extend them to environments. Let σ_1 and σ_2 be environments on the same variable domain, $\sigma_1 \nabla \sigma_2 = \sigma_3$ such that if $\sigma_1(x) = [l_1, u_1]$ and $\sigma_2(x) = [l_2, u_2]$, then $\sigma_3(x) = ([l_1, u_1] \nabla [l_2, u_2])$ for all variables x. Similarly, $\sigma_1 \Delta \sigma_2 = \sigma_4$ such that if $\sigma_1(x) = [l_1, u_1]$ and $\sigma_2(x) = [l_2, u_2]$, then $\sigma_4(x) = ([l_1, u_1] \Delta [l_2, u_2])$ for all variables x. We define a widening and narrowing iteration, denoted *widen_narrow_iter*, on environments. This definition is defined in terms of a widening iteration, denoted as *widen_iter(f)* and defined as follows:

Definition 1. ***(Widening Iteration)*** *Given a monotonic function f that is a denotation of a program, we say that widen_iter(f) is the environment σ_n, where n is the least natural number such that $\sigma_{n+1} = \sigma_n$, and where σ_i is inductively defined as $\sigma_0(x) = \bot$ for all variable x, and $\sigma_{i+1} = \sigma_i \nabla f(\sigma_i)$.*

We note that *widen_iter*(f) is a post fixpoint that is obtained after finitely many steps. In the definitions the data-flow functions are representations of the abstract semantic transformation functions. The *widen_narrow_iter(f)* is defined as follows:

Definition 2. ***(Narrowing/Widening Iteration)*** *Given a monotonic function f that is a denotation of a program, we say that widen_narrow_iter(f) is an environment σ_n where n is the smallest natural number such that $\sigma_{n+1} = \sigma_n$, and where σ_i is defined as: $\sigma_0 = \mathit{widen\_iter}(f)$, and $\sigma_{i+1} = \sigma_i \Delta f(\sigma_i)$.*

3 Approach

Our approach uses elimination-based data-flow analysis and embeds widening and narrowing inside its algebraic loop-breaking operation. The elimination-based approach automatically discovers program loops in the CFG, regardless of the control-flow graph topology. In this section we formally define our proposed framework and illustrate it by an example.

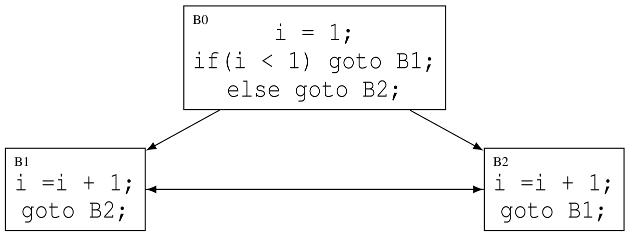

Fig. 1. An Irreducible CFG of a Diverging Program

3.1 Preliminaries

We define a set $\mathcal{X} = \{x_0, \ldots, x_n\}$ of *data-flow* variables, where n is the number of basic blocks in the program. We also define a set of function symbols $\mathcal{F}$ which contains a special symbol F^*. A *constraint* is of the form $x \sqsupseteq e$, where $x \in \mathcal{X}$ and e is an element of set $T \subseteq (\mathcal{X} \cup \mathcal{F})^*$ such that for all $t \in T$ either $t \in \mathcal{X}$ and or $t = f(t_1, \ldots, t_n)$ such that $f \in \mathcal{F}$ and $t_1, \ldots, t_n \in T$. Now, $e = f(t_1, \ldots, t_m)$ such that $f \in \mathcal{F}$ and $t_1, \ldots, t_k \in T$. Further, we say a constraint *system* for simply a set of constraints.

Elimination Phase. The elimination-based framework's operations transform the constraint system. There are two kinds of transformations that are defined on a constraint system: *substitution* and *loop breaking*.

Definition 3. ***(Substitution Operation)*** *Given* $(x_i \sqsupseteq e_i) \in cs$, *and for all* j *such that* $(x_j \sqsupseteq e_j) \in cs, sub_{i,j}(cs) = \{x_i \sqsupseteq e_i[e_j/x_j] | i{\neq}j\} \cup \{x_i \sqsupseteq e_i | i = j\} \cup \{x_k \sqsupseteq e_k | k \neq i, (x_k \sqsupseteq e_k) \in cs\}$. *Now,* $syssub_i(cs) = (sub_{n,i}(cs) \cdot \ldots \cdot sub_{0,i})(cs)$.

Intuitively, in the substitution operation $sub_{i,j}(cs)$, i is an index to a constraint q in cs that will be acted upon, and j is an index to a constraint in cs that will be substituted into q. In *syssub*$_i$*(cs)*, i is an index to a constraint in cs that will be substituted into all other constraints in cs. We next define the loop breaking operation.

Definition 4. ***(Loop-Breaking Operation)*** *Let* $(x_i \sqsupseteq e_i) \in cs$, *and*
$lb_i(cs) = \{x_i \sqsupseteq F^*(e_i, x_i') | x_i \in var(e_i)\} \cup \{x_i \sqsupseteq e_i | x_i \notin var(e_i)\} \cup \{x_k \sqsupseteq e_k | k \neq i\}$.

Given a constraint $x_i \sqsupseteq e_i$ of cs with $x_i \in var(e_i)$, the loop-breaking operation $lb_i(cs)$ wraps a binary function F^* around e_i. The resulting expression $F^*(e_i, x_i')$ denotes an environment σ, such that $\sigma = e_i[\sigma/x_i]$. We note here that σ need not be the least (wrt. $\sqsupseteq$ extended to environments) of such solution. As we later see, F^* is implemented as a sequence of widening and narrowing. By repeatedly applying these operations, following some algorithm, the constraint system is rewritten to its normal form, i.e., a system of constraint that cannot be further reduced.

We describe through an example how our proposed framework performs a range analysis using an example in Figure 1 of an irreducible graph, that may occur in intermediate representations. The example initializes the value i to one and enters the B2 block, as the condition to B1 is invalid. The block B2 increments i by one and

Block	1st Iter.	2nd Iter. (Widen)	3rd Iter. (Narrow)
x_2	$\perp$	[2,2]	[2,∞]
f_0	[1, 1]	[1, 1]	[1, 1]
A : f_1	$\perp$	[3, 3]	[3, ∞]
f_2	[2, 2]	[2, 4]	[2, ∞]
x_2'	[2, 2]	[2, 2]∇[2, 4]=[2, ∞]	[2, ∞]Δ[2, ∞]=[2, ∞]

Fig. 2. Solving an Irreducible CFG of a Diverging Program

unconditionally branches to B1. The block B1, also increments i by one and unconditionally branches back to B2 and this process continues. The corresponding data-flow constraint system of the example in Figure 1 contains $x_0 \sqsupseteq f_0(\perp)$, $x_1 \sqsupseteq f_1(x_0,x_2)$, and $x_2 \sqsupseteq f_2(x_0,x_1)$. To solve this constraint system, we must eliminate all variables so that the constraint system is in a normal form. We note that the functions are defined by an appropriate abstract semantics function $\mathcal{S}^{*\sharp}[\![\cdot]\!]$ as follows:

$$\begin{aligned} f_0(\perp) &= \mathcal{S}^{*\sharp}[\![\texttt{i=1}]\!]\{\texttt{i} \mapsto \perp\} = \{\texttt{i} \mapsto [1,1]\}, \\ f_1(x_0,x_2) &= \mathcal{S}^{*\sharp}[\![\texttt{i=i+1}]\!](x_0 \sqcup x_2), \\ f_2(x_0,x_1) &= \mathcal{S}^{*\sharp}[\![\texttt{i=i+1}]\!](x_0 \sqcup x_1). \end{aligned}$$

The elimination framework will perform a sequence of transformations, with the $sub_{i,j}$ and lb_i operations. Ultimately, any cyclic graph will result in a loop-breaking operation being called, regardless of the reducibility of the graph. In this example we follow the general elimination algorithm of [12].

A series of algebraic operations are performed to reduce the system of constraints to a normal form. In the first transformation, x_0 is substituted for all its occurrences in the constraint system. This is followed by substituting x_1 for all of its occurrences. Next, we find a recursive definition in the definition of x_2. We perform a loop breaking operation which wraps the right-hand side of x_2 with a F^* operations. Finally, we backward propagate the definition of x_2 to x_2's predecessors (i.e., constraints x_1 and x_0) and obtain a normal form. The algorithm when implemented using DJ-graphs has a asymptotic complexity of $O(m \log n)$ as shown in [14]. The naive implementation that follows the Gaussian elimination-like pattern has cubic complexity.

Solving Phase. Once the constraint solution is normalized, it can be solved to determine the variable ranges at each program point. When resolving a recurrence with the F^* function, i.e. for the function $F^*(f_i(\ldots x_i \ldots),x_i')$, we say that x_i' is the fixpoint value that shall replace x_i. The next step is to find its value. For this we implement F^* as $F^*(e) = narrow\_widen\_iter(e)$,, where $narrow\_widen\_iter$ is as given by Definition 2. We present an example of solving one constraint from the system of constraints. In Figure 2 we distinguish the two instances of f_1 using the letters A and B. Our algorithm performs widening and narrowing on the interval values for x_2' at each traversal of the tree. In that sense the constraint system is solved in a bottom-up manner. Before each iteration, x_2 is assigned the value of x_2' (initially $\perp$). Note that since we always discuss the value of the variable `i`, we simply write x_j whenever we mean $x_j(\texttt{i})$ for any j. It also is important to note that there is a condition `i<1` on the edge (f_0, f_1) and another condition `i>=1` on the edge (f_0, f_2) obtained from the control-flow graph of the original program. The conditions restrict the interval values obtained via the edges. The first iteration starts with x_2 having the value of $\bot$, and f_0 has the constant value of $[1,1]$. The condition on the edge $(f_0, A : f_1)$ restricts the value of `i` to $\bot$ and since the initial value of x_2 is $\bot$, $A : f1$ (which is an increment operation) propagates the value $\bot$ to f_2. This is merged with the constant value $[1,1]$ of f_1 resulting in $[1,1]$ which is incremented by f_2 to $[2,2]$, which then becomes the next value for x'_2. In the next iteration, we assign this value to x_2. In a similar process, we obtain the interval $[2,4]$ from f_2. Here we apply the widening operator on $[2,2]$ and $[2,4]$ resulting in the interval value $[2,\infty]$, which is stable. The next iteration tries to improve on the fixpoint value of $[2,\infty]$ via narrowing. This, however, for this program, is not possible and we reach a fixpoint. We can conclude that the value of `i` may diverge. By inspecting the program in Figure 1 and the program analysis results in Figure 2 it is easy to see that these results are accurate. The variable i is incremented twice as both functions f_1 and f_2 increment i, and it causes a divergent value resulting from the infinite loop. Although this example considers the case of a single loop, our approach can naturally be applied to more complex control-flow graphs.

Term-Size Limits. In some constraint systems the structural depth of right-hand side terms may increase rapidly as a result of substitutions. To mitigate this problem we limit the term sizes. After a substitution or loop-breaking we determine the structural size (i.e. number of atomic sub-terms) of the right-hand side of a constraint. If the total size is larger than a threshold K, we replace the term with an interval that is an over-approximation of its value. There are various methods to over-approximate terms. One approach is to replace the entire term with $[-\infty,\infty]$ and hope to recover some bounds through future narrowing.

4 Experiments

In this section we summarize the results of our program analysis tool. In particular we highlight the execution-time/precision trade-off with various term-size limits. In Table 1 we present our results of the bounds of variables in looped LLVM programs from our test suite. The characteristics of these test programs are summarized in left most columns. Programs T6-T8 produce terms with a large structural size and we show how precision and execution-times vary as the threshold value of K is changed. For programs T0-T5, we found that limiting the value of K made no major improvements in execution-time. We present the types of variable bounds observed in each test program. Exact results, have the form $[l,u]$ where $l = u$. Bounded intervals are of the form $[l,u]$ where $l < u$. This type of result indicates that no widening occurred or the widened bounds were recovered by narrowing. Fully widened intervals have the form $(-\infty,\infty)$, where both directions are widened. Partially widened intervals are of the form $[l,\infty)$ or $(-\infty,u]$, indicating widening occurred in a single direction or a widened bound was recovered by narrowing. Note that in programs where a variable diverges this is often a

Test	K	Exact	Bounded	Part	Full	Time	LOC	Max	Loops
		(%)	(%)	(%)	(%)	(Sec)	(LLVM)	Nest	
T0	–	15	55	30	0	0.016	48	3	2
T1	–	20	30	50	0	0.013	30	2	1
T2	–	8	3	2	0	0.024	53	3	2
T3	–	100	0	0	0	0.048	44	3	0
T4	–	5	95	0	0	0.039	28	2	1
T5	–	25	70	0	0	0.066	45	3	3
T6	25	6	94	0	0	1.441	88	3	2
	10	6	24	0	70	0.839			
T7	30	80	20	0	0	1.979	130	2	1
	10	35	2	0	63	1.469			
T8	30	79	20	1	0	5.385	144	3	7
	10	60	10	1	29	3.779			

Table 1. Variable Bounds Per Test Case

legitimate result, e.g., unbounded loops. In Table 1 our program analysis attains mainly bounded intervals. The next common results are exact and partially widened values. The two cases of full widening occur in programs where variables assume values from an external function call and are not constrained by any conditional control-flow. As previously explained, certain programs can produce constraint systems where the right-hand side of a constraint has a large structural depth. Programs T6-T8 have this property. In the experiments in Table 1 we find that $K = 20$ is a good trade-off between precision and execution-time.

5 Conclusion

We have presented an elimination-based static analysis framework for range analysis of program variables values in low-level languages. We have shown that it is possible to extend elimination-based data-flow analysis algorithms to incorporate analyses with highly bound information lattices. We demonstrated the effectiveness of our framework by an implementation as an LLVM compiler pass. Our technique extends the elimination-based data-flow analysis approach by combining it with abstract interpretation and introducing the term limits. We have shown this approach performs fast and precise static program analysis on low-level code. Some prospective future work is to improve the limit-size technique to obtain better a execution-time / precision trade-off. Additionally, it would be interesting to embedded other acceleration techniques in the framework by refining the semantics of the loop-breaking operation.

References

1. J. Berdine, C. Calcagno, and P. W . O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *4th FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer, 2005.

2. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*, volume 735 of *LNCS*, pages 128–141. Springer, 1993.
3. Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM TOPLAS*, 12:341–395, July 1990.
4. B. Le Charlier and P. Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, 1992.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252. ACM, 1977.
6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTREÉ static analyzer. In *11th ASIAN*, LNCS, pages 272–300. Springer, 2006.
7. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI '03*, pages 155–167. ACM, 2003.
8. J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23:158–171, January 1976.
9. G. A. Kildall. A unified approach to global program optimization. In *1st POPL*, pages 194–206. ACM, 1973.
10. C. Lattner and V. Adve. The LLVM instruction set and compilation strategy, 2002.
11. G. Ramalingam. Identifying loops in almost linear time. *ACM TOPLAS*, 21(2):175–188, 1999.
12. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM C. Surv.*, 18:277–316, September 1986.
13. B. Scholz and J. Blieberger. A new elimination-based data flow analysis framework using annotated decomposition trees. In *ETAPS '07*, LNCS. Springer, 2007.
14. V. C. Sreedhar. *Efficient program analysis using DJ graphs*. PhD thesis, McGill University, Montreal, Canada, 1995.
15. T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS '09*, 2009.

A Correctness Proof

Theorem 1. ***(Correctness)*** *If* $cs \rightsquigarrow_i cs'$ *for some* i, *and* $F^*(e,v) = narrow\_widen\_iter(e)$ *then* $SOL(cs') \sqsupseteq SOL(cs)$.

Here we prove the correctness of our approach, that is, Theorem 1. We can intuitively understand correctness as:

1. Maintaining the invariant that every constraint transformation is also in the solution space.
2. Narrowing and widening results in a solution, that is, implementing F^* using *narrow_widen_iter* in such a way that $F^*(e,v) = narrow\_widen\_iter(e)$ results in a solution.

We first prove that the substitution operation of the elimination framework is correct.

Lemma 1. ***(Substitution Correctness)*** $SOL(sub_{i,j}(cs)) \sqsupseteq SOL(cs)$.

Proof. In case $i = j$, $sub_{i,j}(cs) = cs$ (see Definition 3) and therefore trivially $SOL(sub_{i,j}(cs)) \sqsupseteq SOL(cs)$. In case $i \neq j$, let us assume a solution $\psi \in SOL(cs)$ such that given $(x_i \sqsupseteq e_i) \in cs$ where $e_i = g_i(x_0, \ldots, x_n)$, we have that $\psi(x_i) \sqsupseteq g_i(\bigsqcup\{\psi(x_k) \mid k \in pred(i)\})$ due to commutativity of $\sqcup$. Here we assume that several transformations have been performed from the initial set of constraints, resulting in g_i, which is some some composition of f_ks and F^*. From the properties of these operators, g_i is monotonic. Now, due to commutativity of $\sqcup$,

$$\psi(x_i) \sqsupseteq g_i(\bigsqcup\{\psi(x_k) \mid k \in pred(i) - \{j\}\} \sqcup \psi(x_j)). \tag{1}$$

Since $\psi \in SOL(cs)$, it should be the case that $\psi(x_j) \sqsupseteq g_j(\bigsqcup\{\psi(x_k) \mid k \in pred(j)\})$ for any $j \in pred(i)$, and therefore the following holds:

$$\begin{gathered}\bigsqcup\{\psi(x_k) \mid k \in pred(i) - \{j\}\} \sqcup \psi(x_j) \\ \sqsupseteq \\ \bigsqcup\{\psi(x_k) \mid k \in pred(i) - \{j\}\} \sqcup \\ g_j(\bigsqcup\{\psi(x_k) \mid k \in pred(j)\}).\end{gathered}$$

Now due to monotonicity of g_i,

$$\begin{gathered}g_i(\bigsqcup\{\psi(x_k) \mid k \in pred(i) - \{j\}\} \sqcup \psi(x_j)) \\ \sqsupseteq \\ g_i(\bigsqcup\{\psi(x_k) \mid k \in pred(i) - \{j\}\} \sqcup \\ g_j(\bigsqcup\{\psi(x_k) \mid k \in pred(j)\})).\end{gathered}$$

The right-hand side of the above reflects the effect of $sub_{i,j}$. Using this and (1), we have:

$$\begin{gathered}\psi(x_i) \sqsupseteq \\ g_i(\bigsqcup\{\psi(x_k) \mid k \in pred(i) - \{j\}\} \sqcup \\ g_j(\bigsqcup\{\psi(x_k) \mid k \in pred(j)\})).\end{gathered}$$

ψ is therefore also a solution of $SOL(sub_{i,j}(cs))$.

We also establish that the loop breaking maintains a solution.

Lemma 2. ***(Loop Breaking Correctness)*** *$SOL(lb_i(cs)) \sqsupseteq SOL(cs)$ when $F^*(e, v) = narrow\_widen\_iter(e)$.*

Proof. Since *narrow_widen_iter* is a sequence of widening until fixpoint is reached followed by narrowing until fixpoint is reached, we can rely on the classical result of [5].

Following is Theorem 1 re-stated here.

Theorem 2. ***(Correctness)*** *If $cs \rightsquigarrow_i cs'$ for some i, and $F^*(e, v) = narrow\_widen\_iter(e)$ then $SOL(cs') \sqsupseteq SOL(cs)$.*

Proof. Immediate from the fact that $cs \rightsquigarrow_i cs'$ is either a composition of $sub_{k,i}$ for all k, or lb_i given that $F^*(e, v) = narrow\_widen\_iter(e)$, and from Lemmas 1 and 2.