TECHNICAL REPORT

Identifying Unnecessary Bounds Checks Through Block-Qualified Variable Elimination

Jeffery von Ronne

Tech. Report CS-TR-2010-008
Department of Computer Science
The University of Texas at San Antonio

August 2010

1 Overview

Java's memory-safety relies on the Java Virtual Machine checking that each access to an array does not exceed the bounds of that array. When static analysis can determine that some array access instruction will never use an out of bounds index, the cost of dynamically checking the index each time the instruction is executed can be avoided.

This report introduces Block-Qualified Variable Elimination (BQVE) as a program analysis technique that extends Fourier-Motzkin Elimination [5, 3], so that it can be applied directly to programs in Static Single Assignment Form (SSA) [2] in order to determine whether array accesses of such programs are in bounds. This is accomplished by performing the variable elimination on a constraint system, where each constraint consists of a linear inequality the program's variables and a *block-validity qualifier* that specifies in what program regions the linear inequality holds. Additional modifications handle SSA's ϕ-functions.

In summary, the BQVE algoirthm works as follows: BQVE operates on SSA programs for which a control flow graph and a dominator tree have been computed. From such a program, a set of block-qualified linear program constraints consistent with the program's semantics is extracted from the program code. These constraints are combined with additional constraints derived from properties that are to be tested (e.g., that a particular array access never exceeds its upper bounds) to form a system of constraints. The BQVE algorithm then operates by eliminating non-ϕ variables from the constraint system and introducing new constraints derived from the old constraints that involved the eliminated variables or were related to ϕ-functions. After all variables have been eliminated and the ϕ-functions have been processed, the resulting constraint system can be examined to determine whether the property in

question holds.

The BQVE algorithm described here is simpler and more precise than our previous Constraint Analysis System (CAS) algorithm [6]. The main differences from the prior CAS algorithm is the use of block-validity qualifiers (as described in Section 4) instead of CAS's direction flags to limit the validity of constraints and BQVE's explicit representation of constraints holding for ϕ-functions on different predecessor blocks allowing for more constraints to be discovered to hold on ϕ-result variables (Section 6).

The concepts underlying the BQVE algorithm are presented the following sections, and a full specification of the algorithm is found in Appendix A.

2 Linear Inequality Constraints

BQVE begins after the system has derived the control-flow graph and dominator tree, put the code into SSA form, and traversed the program extracting a set of constraints that are known to hold in certain program regions. It is further assumed that all critical edges have been split.

The basic constraints considered by BQVE are integer linear inequalities. These can be expressed in the form $c_1v_1 + \ldots + c_nv_n + c \leq 0$ where $c, c_1, \ldots, c_n$ are integer constants and each of $v_1, \ldots, v_n$ is a symbolic variable representing either the value of an integer program variable or the length of the array referenced by a program variable. For linear inequalities of this form, we call c_i the coefficient on v_i in the inequality and c the constant of the inequality. In the case where the coefficient c_i is positive, the constraint is said to be an upper bound on v_i, and when c_i is negative, the constraint is said to be a lower bound on v_i.

These inequality constraints can be derived from particular instructions based on the semantics of the operations involved in that instruction. For example, the property that a SSA variable x_0 is zero, which would hold after the assignment x_0 `= 0`, can be encoded by the following two linear inequalities that place upper and lower bounds on x_0, respectively:

$$x_0 + 0 \leq 0$$
$$-x_0 + 0 \leq 0$$

Similarly, the property that the value of x_2 is one more than the value of x_1 could be expressed using the constraints:

$$-x_1 + x_2 - 1 \leq 0$$
$$x_1 + -x_2 + 1 \leq 0$$

As a special case, in the discussion that follows, we will refer to constraints that have no non-zero coefficients (i.e., are of the form $c \leq 0$) as *reduced constraints*. A reduced constraint is unsatisfiable if and only if its constant c is greater than zero.

3 Fourier-Motzkin Elimination

The BQVE algorithm extends the well-known Fourier-Motzkin Elimination (FME) algorithm [5, 3]. Given a system of linear inequality constraints, FME eliminates a variable v by removing from the system all upper and lower bound constraints on that variable. These are replaced with new constraints derived from the removed constraints on v by combining each removed upper bound on v with each removed lower bound on v. Each pair of lower-bound and upper-bound constraints, is scaled so that their coefficients are opposite, and then they are added together to produce a new constraint. As an example, consider the system:

$$\begin{aligned} 2x + -3y + 4 &\leq 0 \\ x + -2y + 2 &\leq 0 \\ -x + z + -3 &\leq 0 \\ y + -z + 1 &\leq 0 \end{aligned}$$

To eliminate x from this system, each of the upper-bound constraints on x ($2x + -3y + 4 \leq 0$ and $x + -2y + 2 \leq 0$) is combined with the lower-bound constraint on x ($-x + z - 3$). The first combination is achieved by scaling the lower-bound constraint by 2 and then adding:

$$\begin{array}{rcrcrcrcl} 2x & + & -3y & & & + & 4 & \leq & 0 \\ -2x & & & + & 2z & + & -6 & \leq & 0 \\ \hline & & -3y & + & 2z & + & -2 & \leq & 0 \end{array}$$

The second combination is just the sum of the second upper-bound constraint and the lower bound constraint:

$$\begin{array}{rcrcrcrcl} x & + & -2y & & & + & 2 & \leq & 0 \\ -x & & & + & z & + & -3 & \leq & 0 \\ \hline & & -2y & + & z & + & -1 & \leq & 0 \end{array}$$

After replacing the constraints on x with the new derived constraints, the system for example would be:

$$\begin{aligned} -3y + 2z + -2 &\leq 0 \\ -2y + z + -1 &\leq 0 \\ y + -z + 1 &\leq 0 \end{aligned}$$

The result of each elimination is a new system that is satisfiable if the original system is satisfiable. This process can be repeated until all variables are eliminated, and the result will be a set of reduced constraints. The original system will be known to be unsatisfiable whenever there is an unsatisfiable reduced constraint in the resulting system.

4 Block Qualifiers

In contrast to FME, which operates on linear inequality constraints that are considered to all hold simultaneously, BQVE operates on a set of linear inequalities for which some subset is considered to hold at each program location. This extended system is only considered unsatisfiable if the inequalities holding at some program location are inconsistent.

In examining the locations at which linear inequalities hold, two sources of constraints must be considered: constraints derived from variable definitions and constraints derived from conditional branches being or not being taken. A constraint derived from a program condition will be only known to hold in the region dominated by the target of the conditional branch (or the block to which the control flow falls-through to when the condition is false). A constraint that is extracted from a program instruction that defines a variable will be known to hold at every program location for which the variables involved in the definition are in scope and not subsequently modified. Since BQVE operates on programs in SSA form [2], the variables are in scope and unmodified precisely at the program locations that are dominated by the definition giving rise to the constraint.

In either case, BQVE embellishes the linear inequality constraint with an additional *block-validity qualifier* that specifies in what code regions the linear inequality holds, and the elimination process is extended such that, when it combines two source constraints to create a new derived constraint, it qualifies that new constraint to hold only in the blocks where both of the source constraints were qualified to hold. No matter what the origin of a constraint, the code regions where that constraint is valid will be coterminous with the region dominated by a particular program location. This has two important consequences. First, since all constraints that hold at any point in a basic block will also hold at the end of the basic block, the satisfiability of a system of constraints will be unaffected by what point in a basic block its constraints' regions of validity begin, so we will simply say that a constraint holds in some block if it holds at the end of that block. Second, there is a representative block n that can be used to uniquely define the set of blocks $DomReg(n) = \{m | n \,\mathbf{dom}\, m\}$ in which any constraint holds. Thus, BQVE is able to use a dominator region, identified by that region's representative block, as the block-validity qualifier of each initial constraint drawn from the source program.

Dominator regions can also be used as the block-validity qualifiers of constraints derived during the elimination process. When two source constraints are combined, the combination is considered to be valid only in the intersection of the source constraints block-validity regions. Since the block-validity regions are subtrees of the dominator tree, the region resulting from their intersection will either be the empty set or one of the original subtrees. In either case, this result can be calculated by comparing the two representative blocks under the partial order defined by the **dom** relationship:

$$DomReg(m) \cap DomReg(n) = \begin{cases} DomReg(m) & \text{if } n \,\mathbf{dom}\, m \\ DomReg(n) & \text{if } m \,\mathbf{dom}\, n \\ \emptyset & \text{otherwise} \end{cases}$$

The necessary tests for this operation can be performed efficiently by using memoization or level-ancestor queries [1] on a preprocessed dominator tree.

Constraints that are not qualified to hold anywhere (i.e., have a block-validity qualifier of $\emptyset$), provide no information about possible program states during execution and can be dropped whenever they are created.

As an example, consider the program in Figure 1. Because of the array referenced by a_0 is created with the assignment `a0 = new int[y0]`, it is known that $y_0 - a_0.length \leq 0$ and $-y_0 + a_0.length \leq 0$ in the program region dominated by L1 (i.e., in blocks L1, L2, L3, and L4). As a result of the condition, it is known that $x_1 - y_0 + 1 \leq 0$ in the region dominated by L3 (i.e., in block L3). The result of eliminating y_0 and combining the constraint $y_0 - a_0.length \leq 0$ in $DomReg(L1)$ with $x_1 - y_0 + 1 \leq 0$ in $DomReg(L3)$ is $x_1 - a_0.length + 1 \leq 0$ in $DomReg(L3)$. This is precisely what is needed to know that the accesses to $a_0[x_1]$ in L3 does not violate a_0's upper bound.

5 Testing Conjectures

If the variable elimination process is run to completion, the result will be a system that contains unsatisfiable reduced constraints only if the original system is unsatisfiable. What is usually important, however, is not determining whether the system derived from program constraints is satisfiable. (For any reachable code region, the system of constraints whose block-validities include that region will be satisfiable by construction.) What is actually desired is to determine whether some specific set of *conjectures* are entailed by the constraints extracted from the program instructions. In the example program, to eliminate the bounds check, it is necessary to determine that $x_1 - a_0.length + 1 \leq 0$ and $-x_1 \leq 0$ in block $L3$. If both of these inequalities hold whenever $L3$ is executed then the bounds check for the assignment to $a_0[x_1]$ can be eliminated.

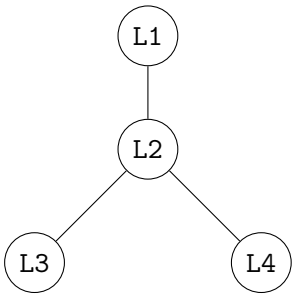
Tests for these properties can be incorporated into the constraint system using an approach modeled after proof by contradiction. In this approach, the opposite of

```
...
int[] a = new int[y];
int x = 0;
while (x < y) {
  a[x] = foo();
  x = x + 1;
}
...
```

(a) source

```
     ...
L1:  ...
     a0 = new int[y0]
     x0 = 0
L2:  x1 = φ(x0, x2)
     if (x1 >= y0) goto L4:
L3:  /* x1 < y0 */
     a0[x1] = foo()
     x2 = x1 + 1
     goto L2:
L4:  /* x1 >= y0 */
     ...
```

(b) SSA Form



(c) dominator tree

Figure 1: An Example Program

each conjectured property is assumed, and added into the system as an *assumed constraint*. (Specifically, for a conjectured linear inequality, $c_1v_1 + \ldots + c_nv_n + c \leq 0$, the assumed constraint will be $-c_1v_1 + \ldots + -c_nv_n + (1-c) \leq 0$.) If after eliminating variables from the new system including this *assumed constraint*, the resulting system has an unsatisfiable reduced constraint, then its opposite, the original conjecture, must hold (or more precisely, the conjectured linear inequality will hold at runtime whenever execution reaches a block in which an unsatisfiable reduced constraint is qualified to be valid).

5.1 Multiple Conjectures and Synthetic Blocks

The procedure described above determines in which blocks a system of block-qualified inequalities is satisfiable and allows one to test a single conjecture per run of the algorithm. Multiple conjectures can be tested simultaneously by appropriately setting the block-validity qualifiers of the assumed constraint when it is added to the system and then examining the block-validities of any resulting unsatisfiable reduced constraints. Specifically, for a query to determine whether some constraint C holds in block L (or, equivalently, $\mathit{DomReg}(L)$), BQVE creates a fresh name for a synthetic block Q that does not correspond to any block in the actual program, but is considered to be part of the dominator tree used in the block-validity tests. In particular, Q is considered to be dominated by all blocks that dominate L (including L), and Q is considered to dominate only itself. $\mathit{DomReg}(Q)$ is then used as the block-validity qualifier of the assumed constraint C^- to test the conjecture that C holds in L. Thus, any constraint derived by combining the assumed constraint C^- with other constraints valid in L will have a block qualifier of $\mathit{DomReg}(Q)$. In contrast, any constraints derived by combining C^- with constraints not valid in L (and not valid in Q) will have a block-qualifier of $\emptyset$ and can thus be safely dropped from the system. If the elimination process results in an unsatisfiable constraint with a qualifier of $\mathit{DomReg}(Q)$, then it is known that the conjecture C, in fact, holds at block L (since the unsatisfiable constraint can only arise if C^- is inconsistent with the constraints known to be valid in L by construction). Multiple conjectures can be tested simultaneously but independently by placing the assumed constraint for each in its own synthesized block.

For example, suppose we wanted to know whether, in the program from Figure 1, the following conjectures hold:

$y_0 - a_0.\mathit{length} - 2 \leq 0$ in block $L1$ (which is the case)
$x_1 - a_0.\mathit{length} + 1 \leq 0$ in block $L3$ (which is the case)
$x_1 - a_0.\mathit{length} + 1 \leq 0$ in block $L2$ (which is not the case)

In fact, the first holds, because of the definition of a_0. The second holds because of the conditional branch and the definition of a_0. The third does not hold, because $L2$

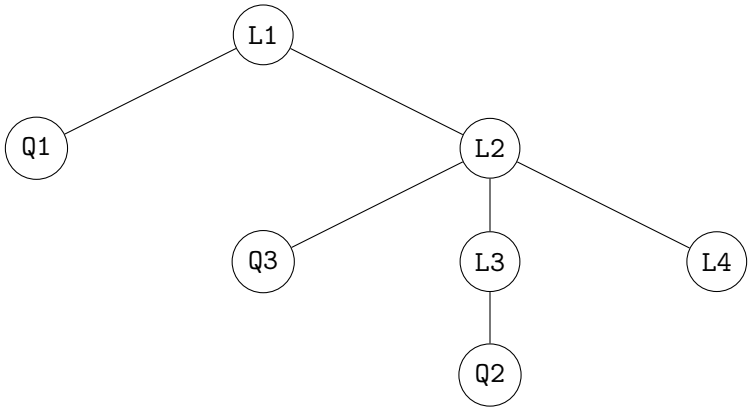


Figure 2: Dominator Tree Containing Synthetic Blocks Q1, Q2, and Q3

is not control-dependent on the conditional branch. If the synthetic blocks $Q1$, $Q2$, $Q3$ are added with dominator relations as shown in Figure 2, then these results can be obtained by adding the following assumed constraints to the system:

$$\begin{aligned} -y_0 + a_0.length + 3 \leq 0 \text{ in } DomReg(Q1) \\ -x_1 + a_0.length \leq 0 \text{ in } DomReg(Q2) \\ -x_1 + a_0.length \leq 0 \text{ in } DomReg(Q3) \end{aligned}$$

During elimination, the first assumed constraint would be combined with the constraint $-a_0.length + y_0 \leq 0$ in $DomReg(L1)$ resulting in $3 \leq 0$ in $DomReg(Q1) \cap DomReg(L1) = DomReg(Q1)$. Since this is an unsatisfiable reduced constraint in $DomReg(Q1)$, it means that the assumed constraint can never hold, and therefore it can be concluded that the first conjecture is true: $y_0 - a_0.length - 2 \leq 0$ always holds in block $L1$. Similarly, the second assumed constraint would be combined with $x_1 - y_0 + 1 \leq 0$ in $DomReg(L3)$ and $y_0 + -a_0.length \leq 0$ in $DomReg(L1)$ to produce $1 \leq 0$ in $DomReg(Q2) \cap DomReg(L3) \cap DomReg(L2) = DomReg(Q2)$, showing that the corresponding conjecture holds. However, when the same two program constraints are combined with the third assumed constraint, the block-validity would be $DomReg(Q3) \cap DomReg(L2) \cap DomReg(L3) = \emptyset$. In fact, there is no combination of constraints involving the third assumed constraint that will produce an unsatisfiable reduced constraint with a non-empty block-validity, and therefore, there will be no unsatisfiable constraints with a block-validity of $DomReg(Q3)$. Since there is no unsatisfiable constraints in $DomReg(Q3)$ it must be conservatively assumed that the third conjecture may not hold.

6 Loops and ϕ-Functions

When an original program variable is modified conditionally or in a loop, a single subsequent use of the variable may refer to different program definitions depending on how program flow reached that variable. This cannot be represented directly in SSA, because each definition is given a unique name. SSA uses ϕ-functions to handle these situations. These ϕ-functions are located where control flow merges, in *join blocks* that have multiple predecessors in the control flow graph. Each ϕ-function has multiple operands, one for each predecessor of the join block containing the ϕ-function. The semantics of the ϕ-functions are such that each time control is transferred from one of the predecessors to a join block, each ϕ-function in that join block will use the value of the ϕ-function's operand corresponding to that predecessor as the ϕ-function's value. As an example, consider the instruction `x1 = φ(x0, x2)` in the program of Figure 1. For the initial iteration of the loop, x_1 will be assigned the value of x_0, but during subsequent iterations of the loop, x_1 will be assigned the value of x_2 from the previous loop iteration. Therefore, the analysis must be designed conservatively, such that it only ascribes to ϕ-result variables, like x_1, those constraints that are known to hold regardless of which block was executed immediately prior to the block containing the ϕ-function.

6.1 Synthetic ϕ-Parameter Variables

In order to distinguish constraints holding on a ϕ-function when the control flow enters the join block containing the ϕ-function from different predecessors, BQVE synthesizes a separate ϕ-parameter variable for each predecessor block. These synthetic ϕ-parameter variables are constrained to be equal to the corresponding ϕ-function operand. So, for the program of Figure 1, the synthetic variables x_1^{L1} and x_1^{L3} would be split from the ϕ-result variable x_1 resulting in the following constraints:

$$x_0 + -x_1^{L1} \leq 0 \text{ in } DomReg(L1)$$
$$-x_0 + x_1^{L1} \leq 0 \text{ in } DomReg(L1)$$
$$-x_1^{L3} + x_2 \leq 0 \text{ in } DomReg(L3)$$
$$x_1^{L3} + -x_2 \leq 0 \text{ in } DomReg(L3)$$

Special rules govern the combining of constraints involving ϕ-result variables and their associated synthetic ϕ-parameter variables. The intuition behind them is that a constraint can be inferred on the ϕ-result variable (e.g., x_1) if there is a stronger constraint (i.e., a constraint that puts at least as tight of a bound) on each of that ϕ-function's synthetic ϕ-parameter variables. This is accomplished by taking each constraint on a synthetic ϕ-parameter v^β that is valid in β and creating a candidate constraint for the related ϕ-result variable v. The candidate can be formed from the

synthetic ϕ-parameter constraint by removing the term involving the synthetic ϕ-parameter v^β and replacing it with a new term that places on v the same coefficient that was on v^β and making the candidate's block-validity the block containing v. This candidate is then compared to the constraints on each of the synthetic ϕ-parameters. If each of these synthetic ϕ-parameters v^γ has at least one constraint that is valid at the corresponding predecessor γ and is stronger than the candidate adjusted to v^γ, then the candidate can be added as a valid constraint on v. Here, adjusting the candidate constraint to the synthetic ϕ-parameter v^γ, means that the coefficient on v becomes the coefficient on v^γ and the block-validity becomes γ.

More concretely, suppose that block α has two predecessors β and γ and contains a ϕ-function defining the variable v_ϕ. In this case, each of the constraints on synthesized ϕ-parameters v^β_ϕ and v^γ_ϕ will be considered as potential candidates and compared with the others. If two of the constraints, place comparable upper (or lower) bounds on the respective synthesized ϕ-parameters that are valid in the corresponding predecessor blocks, then one will be stronger than the other and the other, weaker, one will be an acceptable candidate and lifted to v. That is, if v^β_ϕ has a constraint:

$$c_0 v^\beta_\phi + c_1 v_1 + \ldots + c_n v_n + c \leq 0 \text{ in } DomReg(\beta)$$

and v^γ_ϕ has a constraint:

$$d_0 v^\gamma_\phi + d_1 v_1 + \ldots + d_n v_n + d \leq 0 \text{ in } DomReg(\gamma)$$

where $\forall i : d_0 c_i = c_0 d_i$ and $c_0 d_0 > 0$, one of these will be lifted to:

$$\begin{array}{ll} c_0 v_\phi + c_1 v_1 + \ldots + c_n v_n + c \leq 0 \text{ in } DomReg(\alpha) & \text{if } |d_0|c \leq |c_0|d \\ d_0 v_\phi + d_1 v_1 + \ldots + d_n v_n + d \leq 0 \text{ in } DomReg(\alpha) & \text{if } |d_0|c > |c_0|d \end{array}$$

After the constraints from v^β_ϕ and v^γ_ϕ are lifted to v_ϕ, the new constraint can be combined with the existing constraints on v as when non-ϕ variables are eliminated.

As an example, suppose that the following constraints are known:

$$\begin{aligned} x_1^{L1} - a_0.length - 3 \leq 0 \text{ in } DomReg(L1) \\ x_1^{L1} + 0 \leq 0 \text{ in } DomReg(L1) \\ x_1^{L3} - a_0.length - 1 \leq 0 \text{ in } DomReg(L3) \end{aligned}$$

In this case, the constraints on the synthetic variables would be lifted into a single constraint on x_1: $x_1 - a_0.length - 3 \leq 0$ in $DomReg(L2)$, which is weaker than the first and third constraints.

6.2 Ordering and Dependencies

The special rules for lifting ϕ-parameters, mean that—unlike FME—the elimination of all variables in different orders will not produce equally precise results. The reason for this is that the success of lifting constraints from the synthetic ϕ-parameters to the ϕ-result variable depends on whether constraints are comparable and whether they have the right block-validities at the time they are lifted. Both of these can be affected by the process of eliminating variables and combining constraints.

In most cases, the initial constraints on synthetic ϕ-parameters are not directly comparable, and it is only after combining constraints while eliminating other variables, that new constraints that are comparable are created on the synthetic ϕ-parameters. This argues for lifting synthetic ϕ-parameters after other (non-ϕ) variables have been eliminated. In some cases, however, synthetic ϕ-parameters will be constrained against other ϕ-result variables, such that the lifting of a constraint related to one ϕ-function can enable the lifting of a constraint on another ϕ-function. In these cases, how the processing of ϕ-related constraints is ordered may affect the precision of the analysis.

Another issue affected by ordering is block-validity at the time of lifting. This is because when a constraint is lifted to a ϕ-result variable from its ϕ-parameters, the lifted constraint's block-validity (and thus the constraints it can be productively combined with) is different from the synthetic ϕ-parameter constraints. This may cause precision to be lost as certain non-ϕ variables, which we call ϕ-straddling variables, are eliminated.

In order to address these issues, BQVE does not eliminate each ϕ-result variable as a discrete operation but instead performs tasks related to ϕ-functions during each of its three phases. In the first phase, it clones ϕ-straddling variables and then eliminates regular non-ϕ variables. In the second phase, BQVE combines the constraints on ϕ-result variables, and then lifts constraints on synthetic ϕ-parameters and combines those new constraints with the other ϕ-result variable constraints. In some cases, combining newly lifted constraints with other constraints will produce new constraints on synthetic ϕ-parameters. When this happens, an attempt will be made to lift and combine these until no new constraints are discovered. In the third phase, BQVE eliminates the clones and attempts to lift and combine any new constraints on synthetic ϕ-parameters.

6.3 Cloning of ϕ-Straddling Variables

As noted above, it is often necessary that one or more variables must be eliminated before the respective constraints on synthetic ϕ-parameters become comparable. Thus, unlike with FME, precision depends on the elimination order. BQVE eliminates non-ϕ variables before processing ϕ-result variables and avoids loss of precision during

the elimination of non-ϕ variables by *cloning* certain non-ϕ variables.

If cloning were not used, useful constraints could be (conservatively) lost as non-ϕ variables are eliminated. This can happen, for example, when eliminating a variable that is both bound by a (possibly assumed) constraint whose block-validity qualifier is dominated by a ϕ-function and is also bound by other constraints that involve the synthetic ϕ-parameters of that ϕ-function. In such cases, the block-validity qualifier resulting from combining the dominated constraint with the constraints on the synthetic ϕ-parameters will be $\emptyset$ and thus the constraints will be dropped. This can happen even if, before the elimination, it would have been possible for the synthetic ϕ-parameter constraints to have been lifted to a new constraint on the ϕ-function and given a block-validity qualifier of the region dominated by the block containing the ϕ-function. This would allow the lifted constraints to be productively combined with the ϕ-dominated constraint.

Suppose we had the constraints:

$$x_1^{L1} - a_0.length + 1 \leq 0 \text{ in } DomReg(L1)$$
$$x_1^{L3} - a_0.length + 3 \leq 0 \text{ in } DomReg(L3)$$
$$-x_1 + a_0.length \leq 0 \text{ in } DomReg(Q2)$$

with the dominator tree as shown in Figure 2. In this case, whether an unsatisfiable reduced constraint in $DomReg(Q2)$ is derived, depends on whether $a_0.length$ is eliminated before attempting to lift constraints from x_1^{L1} and x_1^{L3} to x_1. This is because the resulting block qualifier of any constraint derived from the assumed constraint will not include the predecessors ($L1$ and $L2$) of the join block containing the definition of the ϕ-result variable x_1, and so any such derived constraints cannot be lifted to the x_1. This is the case, even though, if the constraints were lifted to x_1 before $a_0.length$ was eliminated, then the $DomReg(Q2)$-qualified assumed constraint would be successfully combined with the lifted constraints relating x_1 to $a_0.length$.

The solution to this problem is to create and protect from elimination clones of variables, such as $a_0.length$, that are involved in constraints with block validities that might 'straddle' ϕ-functions. This can be approximated by considering a variable v to straddle a join block α (and all the ϕ-functions therein) if the definition of v is in a block β that strictly dominates α and v also has a constraint whose region of validity is dominated by α. For any join block α and variable v, where v is considered to straddle α, BQVE creates a clone variable v^α associated with α, and the constraints $v + -v^\alpha \leq 0$ in $DomReg(\beta)$ and $-v + v^\alpha \leq 0$ in $DomReg(\beta)$. The variable v is then allowed to be eliminated in the normal way, but variable v^α will not be eliminated until after the ϕ-functions in α have been eliminated. This allows relationships, between v and the synthetic parameters of some ϕ-function in α, that were entailed by the original constraint system to be represented at the time of that ϕ-function's elimination by the

constraints between the synthetic ϕ-parameters and v^{α}. These constraints can then be lifted to the ϕ-result parameter to produce new constraints with a block-validity qualifier of $DomReg(\alpha)$. Similarly, the relationships on v that were known from the original system to hold in some subset of blocks dominated by $DomReg(\alpha)$ can be preserved as constraints on v^{α}. These can then be combined with constraints lifted to α as the result of ϕ-elimination potentially producing unsatisfiable constraints that would have been (conservatively) lost when v was eliminated.

6.4 Monotonic Cycles

Additional rules are needed for handling induction variables more precisely than would result from the rules for handling ϕ-functions as described above. This is the case, because a monotonically increasing (decreasing) variable can be considered to have a lower (upper) bound equal to its initial value. Monotonically changing variables are detected by identifying *cyclic constraints*, which are inequality constraints with three non-zero terms: a constant term, a term for a synthetic ϕ-parameter, and a term for the corresponding ϕ-result variable, where the coefficients on the synthetic ϕ-parameter and ϕ-result variable are opposite. We will use *cyclic lower bound* to denote a cyclic constraint whose synthetic ϕ-parameter has a negative coefficient, and *cyclic upper bound* to denote a cyclic constraint whose synthetic ϕ-parameter has a positive coefficient. For purposes of comparison of constraints on synthesized ϕ-parameters, a cyclic lower bound constraint with non-negative constant (e.g., $-x_1^{L3} + x_1 + 1 \leq 0$) is considered to be the stronger than all other lower bounds on the synthesized parameter. This case occurs only if the ϕ-function result increases monotonically whenever control flow comes from the corresponding predecessor (i.e., from $L3$ in the example), so it is safe to use whatever lower bound constraints hold on the other incoming edge(s) (e.g., at the entry of the loop). So for our example program, because $-x_1^{L3}+x_1+1 \leq 0$ establishes x_1 as a monotonically increasing induction variable, the constraint on $-x_1^{L1}$ (i.e., $-x_1^{L1} + 0 \leq 0$) can be lifted to x_1 as $-x_1 + 0 \leq 0$

Similarly, cyclic upper bounds with a non-negative constant can be considered stronger than all other upper bounds on the synthesized parameter. For example, if our program was modified so that x_2 was defined by the assignment, `x2 = x1 - 1` (making x_1 a monotonically decreasing induction variable), the upper-bound constraint remaining on x_1^{L3} after elimination of x_2 would have been $x_1^{L3} - x_1 + 1 \leq 0$. This can be considered equivalent to be a strong upper bound on x_1^{L3} allowing any upper bound established for the initial iteration (i.e., an upper bound on x_1^{L1}) to be lifted to x_1.

7 Arithmetic Overflow

Like many languages, Java's integer type cannot represent arbitrarily large (or small) integers. When an integer operation would produce a result that cannot be rep-

resented as a 32-bit binary number, the Java Language Specification [4] specifies overflowing integer operations have 2's complement 'wrap around' semantics. This is relevant, because the analysis will make use of constraints that may not actually hold when overflow occurs. In our running example, the instruction x_2 `=` x_1 `+ 1` gives rise to the constraints $-x_2 + x_1 + 1 \leq 0$ and $x_2 + -x_1 + -1 \leq 0$. The first of these constraints ($-x_2 + x_1 + 1 \leq 0$) will hold if and only if x_1 is not 2,147,483,647 (`MAX`). More generally, the constraint $-a + b + c \leq 0$ can only be drawn from the operation `a = b + c` if $b + c \leq$ `MAX`. Furthermore, the constraint $a - b - c \leq 0$ can only be drawn from the operation `a = b + c` if $b + c \geq$ `MIN`, where `MIN` is -2,147,483,648. Similar rules can be derived for other arithmetic operations.

This can be handled in the analysis by adding additional conjecture queries stating that each arithmetic operation does not overflow. (In general, each arithmetic operation will give rise to two assumed constraints, each in its own synthetic block with one giving the condition for an overflow to occur and the other giving the condition for an underflow to occur.) Any arithmetic-operation-derived constraint whose corresponding overflow conjectures cannot be shown must be discarded along with any results derived from that constraint. This process must be repeated until a fixed point is reached.

It should be noted, that in order to check these overflow conjectures, it is useful to extract some additional constraints from program instructions relating variables to the `MIN` and `MAX` bounds. These constraints state, for example, that each program variable contains a value greater than or equal to `MIN` and less than or equal to `MAX`, and that each valid array length is greater than or equal to 0 and less than or equal to `MAX`.

8 Extracting Proofs

The core algorithm described above can be extended to produce proofs while determining the satisfiability of a system of constraints. As the algorithm combines constraints to obtain derived constraints, it maintains a record of which source constraints were combined to produce each derived constraint. These records can then be combined to produce a proof tree, rooted at the $c \leq 0$ unsatisfiable constraint. For example, consider the Java fragment in Figure 3(a). For this program, testing the conjecture that the index y does not exceed the upper bound of the array involves reducing the system formed from the inequalities entailed by instruction semantics and the assumed constraint that the upper bound is violated ($-y + a.length \leq 0$) and seeing that this reduction is unsatisfiable. While deriving the unsatisfiable constraint $1 \leq 0$ for this system of inequalities, the algorithm would also build a tree of constraints leading to this result similar to the one shown in Figure 3(b). (Since this program is a single block, the block-validity qualifiers have been omitted.) Each in-

```
1  x = 5;
2  a = new int[x];
3  y = 4;
4  a[y];
```

(a) source

$$\dfrac{\dfrac{\dfrac{}{-x+5\le 0}\ ASGN_{LB}(1) \quad \dfrac{}{-a.length+x\le 0}\ NEW(2)}{-a.length+5\le 0}\ ADD \quad \dfrac{\dfrac{}{y-4\le 0}\ ASGN_{UB}(3) \quad \dfrac{}{-y+a.length\le 0}\ ASSUMED}{a.length-4\le 0}\ ADD}{1\le 0}\ ADD$$

(b) constraint unsatisfiability tree

$$\dfrac{\dfrac{\dfrac{}{-x+5\le 0}\ ASGN_{LB}(1) \quad \dfrac{}{-a.length+x\le 0}\ NEW(2)}{-a.length+5\le 0}\ ADD \quad \dfrac{}{y-4\le 0}\ ASGN_{UB}(3)}{y-a.length+1\le 0}\ ADD$$

(c) positive logic proof tree

Figure 3: Program with Proof Tree

equality in the proof tree is justified by adding two inequalities together (ADD) or is derived from program source statements according to a rule specific to those source statements. (In other proof trees, inequalities may also be multiplied by a positive constant.) After the tree demonstrating the unsatisfiability of the system has been produced, the occurrences of the assumed constraint are removed, leaving a positive deductive proof of a constraint as strong as or stronger than the conjectured constraint. Figure 3(c) shows such an deductive proof proving that the array access in the code of Figure 3(a) respects the upper bound of the array. This proof tree can be used as a certificate that the check of the index does not exceed the array's upper bound [7].

9 Acknowledgements

The Block-Qualified Variable Elimination algorithm described in this technical report arose out of joint project with Andreas Gampe, David Niedzielksi, Kleanthis Psarris, and Jonathan Vasek. In particular, it builds on several insights developed during our prior work (of which David was the primary author) on the Constraint Analysis System (CAS) algorithm [6]. I am grateful to them for their insights and feedback with regard to this algorithm. I am especially also appreciative of Jonathan Vasek who implemented and helped to debug the BQVE algorithm.

I would also like to thank Keyvan Nayyeri and Zi Yan for their assistance in formatting the pseudocode found in the appendix of this paper.

This work was supported in part by the National Science Foundation under grant CCF-0846010.

References

[1] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. In *LATIN '02: Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, pages 508–515, London, UK, 2002. Springer-Verlag.

[2] 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 Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[3] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A*, 14(3):288–297, 1973.

[4] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, Boston, MA, third edition, 2005.

[5] Theodore S. Motzkin. *Beiträge zur Theorie der Linearen Ungleichungen*. Inaugural dissertation, University of Basel, 1936. Azriel: Jerusalem.

[6] David Niedzielski, Jeffery von Ronne, Andreas Gampe, and Kleanthis Psarris. A verifiable, control flow aware constraint analyzer for bounds check elimination. In *Statis Analysis: Proceedings of the 16th International Static Analysis Symposium, SAS 2009*, volume 5673 of *Lecture Notes in Computer Science*, pages 137–153, August 2009.

[7] Jeffery von Ronne, Andreas Gampe, David Niedzielski, and Kleanthis Psarris. Safe bounds check annotations. *Concurrency and Computations: Practice and Experience*, 21(1), 2009. DOI: 10.1002/cpe.1341.

A BQVE Algorithm

Procedure 1 SOLVE

Require: *Constraints* – set of block-qualified program and assumed constraints making up the constraint system, where the set of program constraints that have validity in a block are consistent with all possible states the program might have when it exits that block, and each assumed constraint in *Constraints* is placed in its own synthetic block.

Require: *Dominators* – dominator relations between pairs of blocks (program and synthetic) used in constraint block-validity qualifiers

Require: *Vars* – set of program variables that are not defined by ϕ-functions

Require: *PhiVars* – set of variables that are defined by ϕ-function

Require: $StraddlerClones = \emptyset$

Require: $NewPhiParamConstraints = \emptyset$

Ensure: *Constraints* will contain an unsatisfiable reduced constraints with a block validity of $DomReg(\beta)$ only if β is a synthetic block considered to be immediately dominated by α, and there can never be a program state during the execution of α that satisfies the inequality of the assumed constraint associated with β.

```
 1: clonePhiStraddlers()
 2: for all variables v in Vars do
 3:    combineAllConstraints(v)
 4:    Constraints ← Constraints \ (constraints on v)
 5: end for

 6: for all variables p in PhiVars do
 7:    combineAllConstraints(p)
 8: end for
 9: NewPhiParamConstraints ← all constraints on synthetic φ-parameters in Constraints
10: checkPhiParams()

11: BlockList ← blocks from control flow graph in reverse post-order
12: for all β in BlockList do
13:    for all v ∈ StraddlerClones associated with β do
14:       combineAllConstraints(v)
15:       Constraints ← Constraints \ (constraints on v)
16:    end for
17:    checkPhiParams()
18: end for
```

Procedure 2 combineAllConstraints

Require: program variable v

Uses: *Constraints* from SOLVE

```
1: for all constraints κ on v do
2:    combineNewConstraint(κ)
3: end for
```

Procedure 3 COMBINENEWCONSTRAINT

Require: program variable v
Require: a constriant κ on v
Uses: *Constraints* from SOLVE
Uses: *Dominators* from SOLVE

1: **for all** constraints χ on v **do**
2: **if** signs of the coefficients on v in κ and χ are opposite **then**
3: $(\kappa', \chi') \leftarrow$ the linear inequalities of κ and χ scaled so the coefficients on v are opposite
4: $newInequality \leftarrow \kappa' + \chi'$
5: $newBlockValidity \leftarrow$ (the block validity of κ') $\cap$ (the block validity of χ')
6: **if** $newBlockValidity \neq \emptyset$ **then**
7: ADDCONSTRAINT($newInequality$ in $newBlockValidity$)
8: **end if**
9: **end if**
10: **end for**

Procedure 4 ADDCONSTRAINT

Require: constraint κ
Uses: *Constraints* from SOLVE
Uses: *NewPhiParamConstraints* from SOLVE

1: **if** κ is a cyclic constraint on some synthetic ϕ-parameter p^α on a ϕ-function defining p **then**
2: **if** the constant of κ is negative or κ is not valid in α **then**
3: ignore κ and **return**
4: **end if**
5: **if** the coefficient on p^α in κ is positive **then**
6: $\kappa \leftarrow$ canonical strong upper bound on p^α
7: **else**
8: $\kappa \leftarrow$ canonical strong lower bound on p^α
9: **end if**
10: **end if**
11: **if** not $(\exists \chi \in Constraints : \text{STRONGER}(\chi, \kappa))$ **then**
12: $Constraints \leftarrow \left(Constraints \setminus \{\psi \mid \text{STRONGER}(\kappa, \psi)\}\right) \cup \kappa$
13: **if** κ constrains a synthetic ϕ-parameters **then**
14: $NewPhiParamConstraints \leftarrow NewPhiParamConstraints \cup \kappa$
15: **end if**
16: **end if**

Procedure 5 CHECKPHIPARAMS

Uses: *NewPhiParamConstraints* from SOLVE
Uses: *Constraints* from SOLVE

1: $NewPhiParamConstraints \leftarrow NewPhiParamConstraints \cap Constraints$
2: **while** $NewPhiParamConstraints \neq \emptyset$ **do**
3: $\kappa \leftarrow$ remove an element from $NewPhiParamConstraints$
4: **for all** synthetic ϕ-parameters p^α constrained by κ valid in α **do**
5: $p \leftarrow$ the ϕ-result variable associated with p^α
6: $\beta \leftarrow$ the block containing p
7: **for all** predecessors γ of β **do**
8: **for all** χ that constrain p^γ and are valid in γ **do**
9: **if** χ is not a strong bound and STRONGER$(\kappa, \chi[p^\gamma \rightarrow p^\alpha]$ in $DomReg(\alpha))$ **then**
10: TRYPHICONSTRAINT$(p, \chi[p^\gamma \rightarrow p]$ in $DomReg(\beta))$
11: **end if**
12: **end for**
13: **end for**
14: **end for**
15: **end while**

Procedure 6 TRYPHICONSTRAINT

Require: a variable p defined by a ϕ-function
Require: a candidate constraint κ on p
Uses: *Constraints* from SOLVE

1: $\alpha \leftarrow$ the block in which p is defined
2: **if** $\forall \beta \in \text{pred}(\alpha) : \exists \chi$STRONGER$(\chi, \kappa[p \rightarrow p^\beta]$ in $DomReg(\beta))$ **then**
3: ADDCONSTRAINT(κ)
4: COMBINENEWCONSTRAINT(p, κ)
5: **for all** $\gamma \in \text{pred}(\alpha)$ **do**
6: $Weaker_\gamma \leftarrow$ constraints ψ on p^γ for which STRONGER$(\kappa[p \rightarrow p^\gamma]$ in $DomReg(\gamma), \psi)$
7: $Constraints \leftarrow Constraints \setminus Weaker_\gamma$
8: **end for**
9: **end if**

Procedure 7 CLONEPHISTRADDLERS

Uses: *Constraints*, *Dominators*, *Vars*, *PhiVars*, *StraddlerClones* from SOLVE

Ensure: *StraddlerClones* and *Constraints* are initialized with all of clones for ϕ-straddling variables and their accompanying constraints

```
1: for all variables v in Vars do
2:   for all join blocks α do
3:     if v straddles α then
4:       clone v as v^α associated with α
5:       StraddlerClones ← {v^α} ∪ StraddlerClones
6:     end if
7:   end for
8: end for
```

Function 8 STRONGER: `Boolean`

Require: constraint κ

Require: constraint χ

Uses: *Dominators* from SOLVE

```
1: if (the blocks in which κ is valid) ⊇ (the blocks in which χ is valid) then
2:   if κ is the canonical strong upper bound on v, and χ is an upper bound on v then
3:     return true
4:   else if κ is the canonical strong lower bound on v, and χ is a lower bound on v then
5:     return true
6:   else
7:     V ← Vars ∪ PhiVars ∪ StraddlerClones
8:     if ∃a, b > 0 : (∀v ∈ V : a × (coefficient on v in κ) = b × (coefficient on v in χ)) ∧
       (a × (constant of κ) >= b × (constant of χ)) then
9:       return true
10:    end if
11:  end if
12: end if
13: return false
```