

Verifiable Range Analysis Annotations for Array Bounds Check Elimination

Jeffery von Ronne, Kleanthis Psarris, and David Niedzielski

Department of Computer Science
The University of Texas at San Antonio
San Antonio, TX 78249
{vonronne,psarris,dniedzie}@cs.utsa.edu

Abstract. For performance reasons, it is desirable for Java just-in-time (JIT) compilers to statically identify array element accesses that can never cause an out of bounds exception, but the most precise analyses are too expensive to run in JIT compilers. We present verifiable annotations that can be added to Java programs to capture the results of range analyses as claimed linear inequalities and proofs of these claims. These proofs can be efficiently verified so that array bounds checks can be safely eliminated during JIT compilation without performing expensive range analyses at runtime.

1 Introduction

The Java language and the Java Virtual Machine are designed to be type and memory safe. The security and sandboxing features of the Java Virtual Machine are built on top of this foundation of type safety. This protection requires the Java virtual machine to stop the normal continuation of any array element accesses instructions that exceed the bounds of the array being accessed. This can be achieved by performing a runtime “bounds check” operation (i.e., checking that the index is within the array bounds) as part of each access to an array element, but these array bounds checks slow down program execution, and especially for scientific applications the slowdown can be quite substantial [1]. This performance overhead is caused not only by the direct cost of conditional branches implementing the array bounds checks but also by lost opportunities for optimization and parallelization due to Java’s precise exception semantics.

Therefore, for performance reasons, it is desirable for JIT compilers to carry out a static analysis identify array element accesses that can never cause an out of bounds exception and eliminate the array bounds check operations associated with those arrays. Unfortunately, the most precise the analyses (e.g., [1-3]) are too expensive to run in JIT compilers, but faster analyses (e.g., [4],[5]) leave more unnecessary array bounds checks in the program.

In this paper, we present verifiable annotations for SSA-based representations of Java programs, such as SafeTSA [6, 7], that can safely represent the results of range analyses as linear inequalities. These annotations can be produced during

the translation of source Java classes into SafeTSA. During program execution, an enhanced Java Virtual Machine can read these annotations, efficiently verify that these linear inequalities hold, and then perform array bounds check elimination based on those linear inequalities. In this way, the range analysis burden is shifted from the performance-critical just-in-time (JIT) compiler that runs every time the program is executed to the Java source to SafeTSA compiler which only needs to be run once.

2 Bounds Check Elimination

In general, answering the question of whether or not an index will always remain within the bounds of an array is useful for several purposes. For languages that do not define program behavior when the bounds are violated, proving that array bounds are never violated will normally be a precondition for proving the overall correctness of a program, and proving that there is a program input for which the array bounds is violated probably indicates a bug and possibly a security vulnerability that needs to be fixed. Unfortunately, the question of whether the indices of array access in some program are always within the bounds of the arrays is in general undecidable. In some specific instance, however, a static analysis might be able to determine that a particular array access will never violate its bounds, or in another instance, it may be able to determine that there is an execution that does violate the bounds.

If one, however, wants to ensure that all out-of-bounds array accesses are prevented (as required by the Java language), at least some array access will require runtime bounds checks to ensure that the index is within the bounds of the array. If, on the one hand, it can be shown, through static analysis that the index will always be within the array's bounds, then the runtime bounds check is unnecessary. If, on the other hand, the bounds are in fact sometimes violated or it cannot be determined whether they can be violated, then the runtime bounds remains necessary.

Taking the perspective that all of the array accesses “include” a bounds check as part of their semantics (c.f., [8]), it is natural to explicitly represents them in the compiler's intermediate representation. The compiler optimizations that identify and remove those bounds checks that are unnecessary are known as bounds check elimination optimizations.

2.1 Linear Constraints

In order to eliminate the bounds checks associated with an array access, it is necessary to determine that the index is not smaller than the smallest legal index and the index is not larger than the largest legal index. In Java, these constraints means that for an array access, $a[index]$ (where $index$ is some integer expression), to be legal, then $0 \leq index < a.length$. Thus, what is required is to determine what the range of values that $index$ can take.

There have been several algorithms for eliminating bounds checks for Java. They can broadly be categorized as either recognizing specific array forms (e.g., [1,9][5]) and determining the range of the loop induction variable (when it is used as an *index* into an array), or as performing global analysis to limit the range of one program variable relative to another. (e.g., [4,10]). The information used by these algorithms (except, possibly, that of Moreira et al. [1] which also does substantial loop transformation) can be expressed as a limited form of linear inequalities, known as difference constraints.

Generally, if we treat each integer program variable and the length of arrays pointed to by program variables as algebraic variables, then it is possible to examine the semantics of various instructions and the control flow of the program to derive linear inequalities that constrain the possible values the program variables may take at various points in the program. Then when those constraints are combined into a system of inequalities, if the system can only be satisfied when:

$$i \geq 0 \text{ and } i < a.\text{length}$$

where $a.\text{length}$ represents the length of the array a , then an array access instruction that used a as the array and i as the index would not need a runtime bounds check.

For example, the linear constraints on the right can be extracted from the program fragment on the left:

int l = a.length;	Ⓐ $l \leq a.\text{length}$, Ⓑ $l \geq a.\text{length}$
if (x < l - 1) {	Ⓒ $x < l - 1$
y = x + 1;	Ⓓ $y \leq x + 1$, Ⓔ $y \geq x + 1$
a[y];	
}	

Note that, since the variables are all integers, we can change $x < y$ into $x + 1 \leq y$. In addition, the system of linear inequalities is easier to solve if we rearrange them so that they are all asserting that some linear combination of variables is less than or equal to zero:

$$\begin{aligned} \text{Ⓐ } l - a.\text{length} \leq 0, \text{ Ⓑ } -l + a.\text{length} \leq 0, \text{ Ⓒ } x - l + 2 \leq 0, \\ \text{Ⓓ } y - x - 1 \leq 0, \text{ Ⓔ } -y + x + 1 \leq 0 \end{aligned}$$

If we then add constraints Ⓐ, Ⓒ, and Ⓓ, we can see that:

$$y - x - 1 + x - l + 2 + l - a.\text{length} \leq 0 \text{ or } y - a.\text{length} + 1 \leq 0$$

This is equivalent to our condition ($y < A.\text{length}$) for the index y being below the upper bound of the array. It is not possible, however, to show from this system that y is non-negative ($-y \leq 0$), so the lower bounds check associated with the array access cannot be eliminated.

If the program has been translated into three-address code, then the constraints derived from these instructions that can be expressed as linear inequalities will involve at most three variables. Instructions which add or subtract

constants can be expressed as difference constraints (i.e., linear inequalities having coefficients of only 1 and -1 and involving only two variables and a constant).

If the program has been translated into static single assignment form (SSA) [11] rather than just three-address code, then each program variable only takes a single value and linear inequalities derived from instructions can be considered to hold wherever the program variables are in scope. Through the rest of this paper, we will express programs in SSA (each local variable is only assigned by only one instruction and ϕ -functions are used to merge alternative variables from multiple predecessors when the control flow merges).

2.2 Integer Underflow/Overflow

Like most programming languages, Java has a primitive integer type (`int`) that cannot represent any number in the integer domain, but is instead, restricted to integers that can be stored into a 32-bit word using 2's complement representation. When integer arithmetic operations (specifically, addition, subtraction, increment, decrement, and multiplication) result in a value less than -2^{31} (-2147483648 , which we will refer to symbolically as `MIN`) or more than $2^{31} - 1$ (2147483647 , which we will refer to symbolically as `MAX`), the computation “wraps around” so that only the least significant 32-bits are retained, so for example, $1000000 * 1000000$ is -727379968 rather than 1000000000000 .

This difference between the mathematical domain of integers (\mathbb{Z}) and Java's primitive `int` type needs to be taken to account during integer range analysis. If not the soundness of the array bounds check elimination may be compromised. Combining linear inequality constraints relies on properties, such as $(x \leq 0 \wedge y \leq 0) \Rightarrow x + y \leq 0$, which hold for standard arithmetic in \mathbb{Z} , but not for 2's complement arithmetic. Therefore, linear inequality constraints must be expressed in the domain of integer numbers. But linear inequality constraints that would seem to fall out naturally from source code statements may not hold when constraints among 2's complement variables are expressed as inequalities of variables in \mathbb{Z} . For example, consider the claim that the Java statement, `z = x + y`, implies that $-[[z]] + [[x]] + [[y]] \leq 0$, where $[[x]]$ denotes the value in \mathbb{Z} of the Java variable. This claim may not hold if `x + y` overflows or underflows. For example, if `x` $\equiv 100$, `y` $\equiv 2147483647$ (i.e., $2^{31} - 1$), then `z` = -2147483549 (i.e., $-2^{31} + 99$), and it is clear that $-(-2147483549) + 2147483647 + 100 \not\leq 0$

In some cases, this can compromise the soundness of the analysis results and the Java Virtual Machines memory safety. For example, consider the following program fragment:

```
int i = 2147483647;
if (i >= 0) {
    int j = i + 100;
    if (j < a.length)
        x = a[j]
}
```

Due to an integer overflow, the actual value of j being used as an index is -2147483549 , which is clearly not within $[0, \text{a.length})$, so the bounds check associated with the access of $\text{a}[j]$ cannot be legally removed. The ABCD algorithm [4] will, however, determine¹ that $j \geq 0$ and the lower bound is unnecessary by extracting the constraint $i - 0 \geq 0$ from the condition $i \geq 0$, and the constraint $j - i \geq 100$ from the condition $j = i + 100$ and combining them to obtain: $j - 0 \geq 100$ which can be weakened to: $j - 0 \geq 0$. Similarly, the upper bound can also be removed based on the condition: $j < \text{a.length}$. Thus, the ABCD algorithm would remove the bounds check, which would allow the out of bounds memory access to proceed. This is a potentially exploitable flaw that makes virtual machines utilizing ABCD (or similar bounds check elimination algorithms that do not properly handle integer overflow) susceptible to buffer overflow attacks.

3 Annotating Claims and Proofs

Since Java programs are usually optimized during JIT compilation of Java bytecode, for an optimization to be profitable has to “pay for itself”, that is, the cost of performing an optimization must be less than the speedup in a program’s execution resulting from the optimization. In this environment, simple optimizations that cover the most common cases are often preferred to more comprehensive optimizations that produce better code but take much longer to execute. This is the approach taken by, the ABCD algorithm [4], for example.

We take a different approach, instead of trying to use a light-weight bounds check elimination algorithm that covers the most common cases, we are developing an annotation scheme that allows more expensive static analysis to be performed by the annotator once prior to distribution and prior to JIT compilation by the code consumer. The results of the analysis can be shipped to the code consumer as annotations to the bytecode, and the code consumer’s JIT can then optimize the bytecode based on those annotations. This is only safe if the annotations can be verified to be correct, otherwise the code produced could lie causing the code consumer to omit necessary bounds checks leaving the host virtual machine susceptible to buffer overflow attacks.

3.1 Annotated Claims

The core of our annotations are linear inequality constraints. We believe that this will allow a range of optimization algorithms to be employed by various code annotators without imposing a substantial cost in the code consumer. In addition, we add some additional kinds of constraints that allow information derived from the control flow into the system. The following kinds of constraints are allowed in our system:

¹ In practice, constant propagation would probably prevent ABCD from eliminating the bounds check, but if i was assigned 2147483647 based on user input, the result would be as described here.

linear inequality constraints assertions of linear inequalities involving program variables, lengths of arrays pointed to by program variables, and constants

predicate constraints assertions that a particular boolean variable is either true or false

predicated linear inequality constraints assertion that a linear inequality will hold if a particular predicate holds

predicated predicate constraints assertions that some predicate holds if some other predicate holds

Except for a handful of universally valid axiomatic constraints (see Table 1, a constraint cannot be considered valid unless an annotation claims it to be true. These claims are anchored to a particular program point. The claims must match the form of one of those listed in Table 2 or Table 3. These may be categorized as:

- a linear inequality constraint anchored to a particular integer or array operation ((1)-(4),(6)-(13) in Table 2)
- a predicated linear inequalities constraint anchored to an integer comparison operation ((15)-(26) in Table 3)
- a predicated predicate constraint anchored to a boolean “and” or “or” operation ((27)-(30) in Table 3)
- a predicate constraint that is anchored to an out-going control flow graph edge of a conditional branch instruction ((31)-(32) in Table 3)
- a constraint that is anchored to the join node in the control flow graph ((14) in Table 2)
- the claim that a bounds check is unnecessary ((5) in Table 2)

With the exception of claim form, (14), the constraints are dictated by the form and the instruction to which it is attached. The validity of these claim depends only on the semantics of the instruction to which the claim is anchored and the satisfaction of any proof obligations included in the claim form. If the proof obligations are discharged, then the claim is guaranteed to be valid in the region dominated by that claim’s anchor.

The proof obligations can be discharged with annotated proofs consisting of:

- axiomatic constraints
- other claimed constraints which dominate the proof obligation
- the addition combinator (+)
- the modus ponens combinator (MP)

All proof obligations, except for the ones associated with (14), must be proven using only those constraints that have been claimed to be valid at a program point that dominates the claim which the proof obligation is associated with.

3.2 Linear Inequality Claims

Given the earlier discussion of linear inequalities, the claim forms, (1)-(13) in Table 2, should be fairly straight forward. The proof obligations in (8)-(13) serve

to ensure that constraints are derived from addition, subtraction, and multiplication, when integer overflow or underflow cannot disturb them.

Inequalities	
①	$0 \leq 0$
②	$-1 \leq 0$
③	$[[a.\text{length}]] - \text{MAX} \leq 0$
④	$-[[a.\text{length}]] \leq 0$
⑤	$[[x]] - \text{MAX} \leq 0$
⑥	$-[[x]] + \text{MIN} \leq 0$

x is a Java variable of type `int`
 a is a Java variable referencing an array

Table 1. Axiomatic Linear Inequalities

3.3 Boolean Predicates

In Java, `if` statements and `while`, `do`, and `for` loops create regions that are controlled by a boolean expression. In SafeTSA, the boolean controlling a control structure is always specified as a boolean variable, which may be a constant, a parameter, or a variable created by a comparison operator (i.e., `==`, `!=`, `<=`, `>=`, `<`, or `>`) or a boolean logic operator (i.e., `&`, `^`, or `|`).²

If the boolean is a variable created by the integer comparison operations: `==`, `<=`, `>=`, `<`, `>` then a linear inequality constraint can be derived that is true if the boolean variable is true. Similarly, if it is a variable created by the integer comparison operations: `!=`, `<=`, `>=`, `<`, `>` then a linear inequality constraints can be derived which is true if the boolean variable is false. The truth of these inequality constraints is thus predicated on the boolean variable (as shown in (15)-(26) in Table 3).

Predicate constraint (i.e., a boolean variable or its negation) can be derived from conditional branches that depend on a boolean variable. As shown in (31) and (32), if the boolean b controls a conditional branch, and then the predicate b may be asserted on the the control flow graph edge corresponding to the branch being taken, and therefore may be used in the region dominated by that edge. Similarly, the predicate $\neg b$ may be asserted on the control flow graph edge corresponding to the branch not being taken. Therefore, linear inequalities predicated on a boolean variable being true or false, can be used in proof obligations anchored within the region that is control dependent upon that boolean variable being true or false, respectively.

² In SafeTSA, the short-circuit aspect of `&&`, `||`, and `?:` is handled with a synthesized expression-level `if` control structure along with—in the case of `&&` and `||`—a non-short-circuiting instruction implementing `&` or `|`, respectively.

Instruction Form	Nmbr.	Claim	Proof Obligations
$x = a.length$	(1)	$[[x]] - [[a.length]] \leq 0$	—
	(2)	$-[[x]] + [[a.length]] \leq 0$	
$a = \text{new } C[x]$	(3)	$[[x]] - [[a.length]] \leq 0$	—
	(4)	$-[[x]] + [[a.length]] \leq 0$	
$a[i]$	(5)	access within bounds	$[[i]] - [[a.length]] + 1 \leq 0$ $-[[i]] \leq 0$
$x = y$	(6)	$[[x]] - [[y]] \leq 0$	—
	(7)	$-[[x]] + [[y]] \leq 0$	
$x = y + z$	(8)	$[[x]] - [[y]] - [[z]] \leq 0$	$-[[y]] - [[z]] + \text{MIN} \leq 0$
	(9)	$-[[x]] + [[y]] + [[z]] \leq 0$	$[[y]] + [[z]] - \text{MAX} \leq 0$
$x = y - z$	(10)	$[[x]] - [[y]] + [[z]] \leq 0$	$-[[y]] + [[z]] + \text{MIN} \leq 0$
	(11)	$-[[x]] + [[y]] - [[z]] \leq 0$	$[[y]] - [[z]] - \text{MAX} \leq 0$
$x = c * y$	(12)	$[[x]] - c[[y]] \leq 0$	$-c[[y]] + \text{MIN} \leq 0$
	(13)	$-[[x]] + c[[y]] \leq 0$	$c[[y]] - \text{MAX} \leq 0$
$x_1 = \phi(x_0, x_2)$	(14)	$c[[x_1]] + \dots \leq 0$	$c[[x_0]] + \dots \leq 0$ * $c[[x_2]] + \dots \leq 0$

Where x , y , and z are Java variables of type `int`, a is a Java variable containing an array reference, and c is an integer constant

* proof obligation is based on the claimed constraint and the all of ϕ -functions in the join node. See the text for detail.

Table 2. Claimable Linear Inequality Constraints

In addition, predicates associated with a boolean may be inferred from boolean predicates for other variables that are derived from the variable using logical “and” or “or” (using (27)-(30) in Table 3).

3.4 Merging Constraints at Join nodes

The claim form (14), which is associated with join nodes (i.e., nodes in the control flow graph that have more than one predecessor and may contain ϕ -functions), requires some explanation. This claim form is a consequence of the instructions being in SSA form, and exists to allow constraints that are valid in all of the join node’s predecessors to the program region dominated by the join node, even though the join node may not be dominated by any of the join node’s predecessors. This can be used to annotate loop invariants, but can also be used to show that some constraint holds after an if statement because it holds at the

Instruction Form	Claimed Constraint
$\mathbf{b = x == y}$	(15) $b \Rightarrow \quad [[x]] - [[y]] \leq 0$
	(16) $b \Rightarrow \quad -[[x]] + [[y]] \leq 0$
$\mathbf{b = x != y}$	(17) $\neg b \Rightarrow \quad [[x]] - [[y]] \leq 0$
	(18) $\neg b \Rightarrow \quad -[[x]] + [[y]] \leq 0$
$\mathbf{b = x <= y}$	(19) $b \Rightarrow \quad [[x]] - [[y]] \leq 0$
	(20) $\neg b \Rightarrow \quad -[[x]] + [[y]] + 1 \leq 0$
$\mathbf{b = x < y}$	(21) $b \Rightarrow \quad [[x]] - [[y]] + 1 \leq 0$
	(22) $\neg b \Rightarrow \quad -[[x]] + [[y]] \leq 0$
$\mathbf{b = x >= y}$	(23) $b \Rightarrow \quad -[[x]] + [[y]] \leq 0$
	(24) $\neg b \Rightarrow \quad [[x]] - [[y]] + 1 \leq 0$
$\mathbf{b = x > y}$	(25) $b \Rightarrow \quad -[[x]] + [[y]] + 1 \leq 0$
	(26) $\neg b \Rightarrow \quad [[x]] - [[y]] \leq 0$
$\mathbf{b = c \& d}$	(27) $b \Rightarrow c$
	(28) $b \Rightarrow d$
$\mathbf{b = c d}$	(29) $\neg b \Rightarrow \neg c$
	(30) $\neg b \Rightarrow \neg d$
$\mathit{if\ b\ goto\ \dots}$	(31) $b \quad [\text{branch taken}]$
$\mathit{if\ b\ goto\ \dots}$	(32) $\neg b [\text{branch not taken}]$

Table 3. Predicated Constraints

end of the then part of the if statement and at the end of the else part of the if statement.

(14) is unique in a few ways. First, the claim is anchored to a join node as a whole rather than to any individual ϕ -function contained inside of it. Second, the constraint being claimed can be any constraint (which must be described completely in the annotation) as long as the required proof obligations are discharged. Third, one proof obligation is anchored to each of the join node's predecessors and uses the claims which are valid at the branch instruction in that predecessor. Fourth, the proof obligations are obtained by taking the claimed constraint and substituting, for each occurrence of variables which are on the left-hand side of ϕ -functions in the join node, the variable on the right-hand side of that ϕ -function that is associated with the proof obligation's anchor.

3.5 Representing Proofs

Proof obligations can be discharged by providing a sequence of references to universal constraints, claimed constraints, and constraint combinators.

+ **(addition): *inequality + inequality = inequality*** which takes two inequalities and combines them by adding them together to produce a new linear equality. For example,

$$\frac{-i_2 + i_1 + 1 \leq 0}{+ -i_1 \leq 0} \quad \frac{}{-i_2 + 1 \leq 0}$$

MP (Modus Ponens) takes a predicated constraint, and the predicate on which the constraint is predicated, and yields an unpredicated constraint.

$$\frac{b \Rightarrow i_1 - x_0 + 1 \leq 0 \quad \text{MP } b}{i_1 - x_0 + 1 \leq 0}$$

In any particular proof, only the axiomatic constraints and constraints that are from claims that dominates the proof obligation can be used.

3.6 An Example

Consider the Java function:

```
void f(int a[]) {
    int sum = 0;
    int i;

    for (i = 0; i < a.length; i++)
        sum = sum + a[i];
}
```

Since the loop variable i is monotonically increasing, with an initial value of 0 and a maximum value one less than `a.length`, the value i should range from 0 to `a.length` - 1, so the access `a[i]` should never cause bounds check exception.

Figure 1 shows how our annotation scheme can be used to prove this fact. The first column at the top of Figure 1 shows the function body translated into SSA form. The instructions are indented according to the dominator relationship. The second column indicates the claim form being used, the third indicates the constraint being claimed, and the last indicates any proof obligations resulting from that claim. The bottom half of Figure 1 show the proof used to fulfill those proof obligations.

It is important to note, however, that not all of this information needs to be explicitly transmitted. Each annotation needs to indicate the claim form, but once this is indicated, except for the constraint for \textcircled{c} ($-i_1 \leq 0$), the actual constraints can be derived from just the instruction and the claim form. Similarly none of the proof obligations need to be included in the annotation, since they are derived from the instruction and the claim form, and the proofs only need

	Instructions	Form	Claims	Obligations
	Init:			
1	$sum_0 \leftarrow 0$	(6)	(A) $sum_0 - 0 \leq 0$	
2	$i_0 \leftarrow 0$	(7)	(B) $-i_0 - 0 \leq 0$	
	Loop:			
3	$sum_1 \leftarrow \phi(sum_0, sum_1)$ (14)	(C)	$-i_1 \leq 0$	$\boxed{1}$ $-i_0 \leq 0$
4	$i_1 \leftarrow \phi(i_0, i_2)$			$\boxed{2}$ $-i_2 \leq 0$
5	$t_0 \leftarrow a_0.length$	(6)	(D) $t_0 - a_0.length \leq 0$	
6	$t_1 \leftarrow i_1 < t_0$	(21)	(E) $t_1 \Rightarrow i_1 - t_0 + 1 \leq 0$	
7	<i>if</i> t_1 <i>goto</i> Body <i>else goto</i> Done			
	Body:	(31)	(F) t_1	
8	$t_2 \leftarrow \text{load } a_0[i_1]$	(5)	within bounds	$\boxed{3}$ $-i_1 \leq 0$ $\boxed{4}$ $i_1 - a_0.length + 1 \leq 0$
9	$sum_2 \leftarrow sum_1 + t_2$			
10	$i_2 \leftarrow i_1 + 1$ <i>repeat</i> Loop	(9)	(G) $-i_2 + i_1 + 1 \leq 0$	$\boxed{5}$ $i_1 + 1 - \text{MAX} \leq 0$
	Done:			

Proofs	
$\boxed{1}$ $-i_0 \leq 0$ anchored at end of Init \because (A)	$\boxed{4}$ $i_1 - a_0.length + 1 \leq 0$ \because $t_1 \Rightarrow i_1 - t_0 + 1 \leq 0$ (E) MP t_1 (F) $\frac{i_1 - t_0 + 1 \leq 0}{+ t_0 - a_0.length \leq 0}$ (D) $\frac{i_1 - a_0.length + 1 \leq 0$
$\boxed{2}$ $-i_2 \leq 0$ anchored at end of Body \because $-i_2 + i_1 + 1 \leq 0$ (G) $\frac{+ -i_1 \leq 0}{-i_2 + 1 \leq 0}$ (C) $\frac{+ -1 \leq 0}{-i_2 \leq 0}$ (2)	$\boxed{5}$ $i_1 + 1 - \text{MAX} \leq 0$ \because $t_1 \Rightarrow i_1 - t_0 + 1 \leq 0$ (E) MP t_1 (F) $\frac{i_1 - t_0 + 1 \leq 0}{+ t_0 - a_0.length \leq 0}$ (D) $\frac{i_1 - a_0.length + 1 \leq 0}{+ a_0.length - \text{MAX} \leq 0}$ (3) $\frac{i_1 + 1 - \text{MAX} \leq 0$
$\boxed{3}$ $-i_1 \leq 0$ \because (C)	

Fig. 1. Annotations for an Idiomatic For Loop

to include the constraints they are using by reference. For example, to establish \textcircled{G} , it is sufficient to annotate instruction 10 with: (9) , \textcircled{E} , MP , \textcircled{F} , $+$, \textcircled{D} , $+$ $\textcircled{3}$; the code consumer can reproduce the rest.

4 Verifying Annotations

Verification of these linear constraint annotations is straight forward. The system merely requires a pre-order traversal of the programs dominator tree, during which a list of active claims is maintained. When an annotation is encountered, the claim form is checked against the instruction it is anchored to; if it is not appropriate, if the referenced claims are not in the active list, or if the referenced claims do not match the kind of constraint required by the combinator, then the annotations will be rejected. Otherwise the proofs are checked by loading referenced claims from the active list, applying the indicated combinators, and computing a resulting proof. If this discharges the proof obligation for that claim, then the claim is added to the active list until it no longer dominates the current node (claims are added and removed from the active list in LIFO order).

5 Related Work

Array bounds analysis has a long history. In 1977, Suzuki and Ishihata described a theorem prover that was able to derive the necessary loop invariants to prove certain assertions (such as, an index being within array bounds) [12]. The next year, Cousot and Halbwachs described a method for analyzing array bounds using properties of convex polyhedrons to solve linear constraints; one of the intended applications of the analysis is array bounds checks [13]. More recent work on array bounds check analysis includes a symbolic bounds analysis for C programs using linear programming to solve a systems of linear inequality constraints [3] and a bounds checker for Fortran programs [2].

The Java programming language has several features that make the bounds checking problem different from that addressed in bounds checking analyses applied to different languages. First, unlike in C and Fortran, in Java, bounds checking is a required and an integral part of the language semantics; all compliant implementations of Java must provide runtime bounds checking and this bounds checking one aspect of the type safety guarantees on which Java's higher-level security features are built. Second, Java's has precise exception semantics, which means that the compiler not only has to detect that an array access is out of bounds, but it has to allow the exception handler to continue with the program state that existed immediately prior to the attempted array access; this limits the ability of an optimizing compiler to reorder instructions. Third, Java is usually first compiled into bytecode rather than machine code, and then the bytecode is compiled into machine code by a JIT compiler during the program's execution in a Java Virtual Machine; optimizations are usually not performed until the JIT compilation, which has more information (it knows the target architecture, has more compilation units—but not necessarily the whole program—available, and

may have runtime profile information) but is constrained in how much time it can spend on optimization. Fourth, the Java language specifies “wrap around” 2’s complement semantics for integer overflow.

There have been several works addressing the array bounds check problems in Java. Moreira et al. use heavy-weight loop-based transformations and optimizations to optimize bounds checks in scientific applications [1]; their goal was to provide a traditional static compiler for Java programs that provides performance approaching that of traditional optimizing compilers for Fortran, so their approach does not support just-in-time compilation and is not a general solution to the Java bounds check problem.

The ABCD algorithm [4] provides a global bounds check elimination based on extended-SSA form and difference constraints, it is quite efficient but has some limitations since it can only obtain difference constraints that can be overlaid onto the SSA graph. Menon et al. extended the ABCD algorithm to produce optimized programs augmented with verifiable proof variables [14]. The result is quite similar to our claims and proof obligations, but they require the verifier to make judgements about facts (similar to our claims) using integer linear programming instead of providing an explicit proof; we expect that our approach to be more efficient for the code consumer.

Qian et al. [10] use an iterative dataflow analysis based on difference constraints to annotate bytecode with an indication of which bounds checks are unnecessary, but no mechanism is provided to verify that these annotations are correct. Chen and Kandemir suggest annotating the fixed point of a iterative dataflow analysis of integer variable ranges which can then be verified using a single iteration of the same algorithm, but no details are provided about what constraints should be derived from particular instructions [15]. Zhao et al.’s optimization is restricted to limited loop forms (and is, therefore, less comprehensive than our approach) but is quite efficient during JIT compilation [5]. Amme and Gampe’s approach is similarly restricted to simple loop forms but can be applied during the production of an extended form SafeTSA rather than during JIT compilation [9]. SafeTSA itself [7] also provides special types to facilitate the removal of full redundant bounds checks (multiple bounds checks of the same index to the same array), but this is quite limited compared to bounds check elimination based on linear constraints.

We are unaware of any previous work that directly addresses the issue of integer overflow as it exists in Java, and in the bounds check literature as a whole, there are only a few papers that mention the issue. Suzuki and Ishihata intend for their analysis to be also used to prove that integers do not overflow, which makes sense in the context of proving program correctness, but does not work in the Java context where integers are defined to “wrap around”. Xi and Haper’s Dependently Typed Assembly Language mentions that an exception could be raised on integer overflow [16], which is also does not fit Java’s semantics. Menon et al. mention that the integer programming used to verify their proof types operates in “ \mathbb{Z} rather than 32- or 64-bit integers” [14]; they do not, however, provide a solution or elaborate on any consequences of this.

We have taken some inspiration from proof-carrying code [17], but our focus is more limited: linear constraints of integers rather than first order logic.

6 Summary

In this paper, we have described verifiable annotations based on linear inequalities that facilitate bounds check elimination for Java programs, and which we are currently implementing within the context of the SafeTSA system. This approach should allow for a more comprehensive bounds check elimination than would normally be done during just-in-time compilation. This will not unduly increase the time required for compilation, because the annotations allow a just-in-time compiler to benefit from extensive static analysis performed by the annotator.

Another important contribution of this work is that our formulation of bounds check analysis is sound in the presence of integer overflow.

7 Acknowledgement.

This research was supported by the United States Air Force Research Laboratory under grant F30602-02-1-0001.

We would like to thank Humayun Zafar for his contribution to the implementation of this annotation scheme within the SafeTSA system..

References

1. Moreira, J.E., Midkiff, S.P., Gupta, M.: From flop to megaflops: Java for technical computing. *ACM Trans. Program. Lang. Syst.* **22**(2) (2000) 265–295
2. Nguyen, T.V.N., Irigoien, F.: Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.* **27**(3) (2005) 527–570
3. Rugina, R., Rinard, M.C.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.* **27**(2) (2005) 185–235
4. Bodík, R., Gupta, R., Sarkar, V.: Abcd: eliminating array bounds checks on demand. In: *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, New York, NY, USA, ACM Press (2000) 321–333
5. Zhao, J., Rogers, I., Kirkham, C., Watson, I.: Loop parallelisation for the jikes rvm. In: *Proceedings of the Sixth International Conference on Parallel and Distributed Computing (PDCAT'05)*, IEEE Computer Society (2005) 35–39
6. Amme, W., Dalton, N., Franz, M., von Ronne, J.: SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*. Volume 36., ACM Press (June 2001) 137–147
7. von Ronne, J., Amme, W., Franz, M.: An inherently type-safe ssa-based code format. Technical Report CS-TR-2006-004, Computer Science, The University of Texas at San Antonio (2006)

8. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second edn. Addison-Wesley (1999)
9. Amme, W., Gampe, A.: Eliminating bound checks through ranges. Vordiplom Project, Informatik, Friedrich-Schiller-Universität Jena (2005)
10. Qian, F., Hendren, L.J., Verbrugge, C.: A comprehensive approach to array bounds check elimination for java. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, London, UK, Springer-Verlag (2002) 325–342
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **13**(4) (1991) 451–490
12. Suzuki, N., Ishihata, K.: Implementation of an array bound checker. In: POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM Press (1977) 132–143
13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM Press (1978) 84–96
14. Menon, V.S., Glew, N., Murphy, B.R., McCreight, A., Shpeisman, T., Adl-Tabatabai, A.R., Petersen, L.: A verifiable ssa program representation for aggressive compiler optimization. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2006) 397–408
15. Chen, G., Kandemir, M.: Verifiable annotations for embedded java environments. In: CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, New York, NY, USA, ACM Press (2005) 105–114
16. Xi, H., Harper, R.: A dependently typed assembly language. In: ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press (2001) 169–180
17. Necula, G.C.: Proof-carrying code. In: POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1997) 106–119