A Verifiable SSA Program Representation for Aggressive Compiler Optimization

Vijay S. Menon[1] Neal Glew[1] Brian R. Murphy[2] Andrew McCreight[3] * Tatiana Shpeisman[1]
Ali-Reza Adl-Tabatabai[1] Leaf Petersen[1]

[1]Intel Labs
Santa Clara, CA 95054

[2]Intel China Research Center
Beijing, China

[3]Dept. of Computer Science, Yale University
New Haven, CT 06520

{vijay.s.menon, brian.r.murphy, tatiana.shpeisman, ali-reza.adl-tabatabai, leaf.petersen}@intel.com aglew@acm.org
andrew.mccreight@yale.edu

Abstract

We present a verifiable low-level program representation to embed, propagate, and preserve safety information in high performance compilers for safe languages such as Java and C#. Our representation precisely encodes safety information via static single-assignment (SSA) [11, 3] proof variables that are first-class constructs in the program.

We argue that our representation allows a compiler to both (1) express aggressively optimized machine-independent code and (2) leverage existing compiler infrastructure to preserve safety information during optimization. We demonstrate that this approach supports standard compiler optimizations, requires minimal changes to the implementation of those optimizations, and does not artificially impede those optimizations to preserve safety.

We also describe a simple type system that formalizes type safety in an SSA-style control-flow graph program representation. Through the types of proof variables, our system enables compositional verification of memory safety in optimized code.

Finally, we discuss experiences integrating this representation into the machine-independent global optimizer of STARJIT, a high-performance just-in-time compiler that performs aggressive control-flow, data-flow, and algebraic optimizations and is competitive with top production systems.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.4 [*Programming Languages*]: Compilers; D.3.4 [*Programming Languages*]: Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Performance, Design, Languages, Reliability, Theory, Verification

Keywords Typed Intermediate Languages, Proof Variables, Safety Dependences, Check Elimination, SSA Formalization, Type Systems, Typeability Preservation, Intermediate Representations

* Supported in part by NSF grants CCR-0208618 and CCR-0524545.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.
Copyright © 2006 ACM 1-59593-027-2/06/0001. . . $5.00.

1. Introduction

In the past decade, safe languages have become prevalent in the general software community and have gained wide acceptance among software developers. Safe languages such as Java and C# are particularly prominent. These languages provide a C++-like syntax and feature set in conjunction with verifiable safety properties. Foremost among these properties is memory safety, the guarantee that a program will only read or write valid memory locations. Memory safety is crucial to both robustness and security. It prevents common programmer memory errors and security exploits such as buffer overruns through a combination of compile-time and run-time checks.

Both Java and C# were designed to allow programs to be compiled and distributed via bytecode formats. These formats retain the crucial safety properties of the source language and are themselves statically verifiable. Managed runtime environments (MRTEs), such as the Java Virtual Machine (JVM) or the Common Language Infrastructure (CLI), use static verification to ensure that no memory errors have been introduced inadvertently or maliciously before executing bytecode programs.

Bytecodes, however, are still rather high-level compared to native machine code. Runtime checks (e.g., array bounds checks) are built into otherwise potentially unsafe operations (e.g., memory loads) to ease the verification process. To obtain acceptable performance, MRTEs compile programs using a just-in-time (JIT) compiler. A JIT compiler performs several control- and data-flow compiler transformations and produces optimized native machine code. In the process, runtime checks are often eliminated or separated from the potentially unsafe operations that they protect. As far as we are aware, all production Java and CLI JIT compilers remove safety information during the optimization process: optimized low level code or generated machine code is not easily verifiable. From a security perspective, this precludes the use of optimized low level code as a persistent and distributable format. Moreover, from a reliability perspective it requires that the user trust that complex compiler transformations do not introduce memory errors.

In recent years, researchers have developed proof languages (e.g., PCC [20] and TAL [19]) that allow a compiler to embed safety proofs into low-level code, along with verification techniques to validate those proofs. They have demonstrated certifying compilers that can compile Java and safe C-like languages [21, 8, 18, 13] while both performing optimizations and generating safety proofs. Nevertheless, although the proof language and verification process is well-developed, implementing or modifying existing optimizations to correctly generate and/or preserve safety information is still an arduous and poorly understood process.

In this paper, we introduce a new program representation framework for safe, imperative, object-oriented languages to aid in the generation, propagation, and verification of safety information through aggressive compiler optimization. In this representation we encode *safety dependences*, the dependences between potentially unsafe operations and the control points that guarantee their safety, as abstract proof variables. These proof variables are purely static: they have no runtime semantics. Nevertheless, they are first class constructs produced by control points and consumed by potentially unsafe instructions. From the perspective of most compiler transformations, they are the same as any other variable.

We argue that this representation is particularly well-suited to use as an intermediate representation for an aggressively optimizing compiler. We demonstrate that it supports common advanced compiler optimizations without artificially constraining or extensively modifying them. In particular, we demonstrate that by carrying proof values in normal variables a compiler can leverage existing transformations such as SSA construction, copy propagation, and dead code elimination to place, update and eliminate proof variables.

We illustrate our ideas in the context of the machine-independent global optimizer of STARJIT [1], a dynamic optimizing compiler for Java and C#. STARJIT was designed as a high-performance optimizing compiler and is competitive in performance with the best production MRTE systems. We describe a prototype integration of our ideas into STARJIT's internal representation, and we discuss how it is able to preserve safety information through a varied set of aggressive optimizations. The original motivation for the safety dependence representation described in this paper was for optimization rather than safety. However, a prototype implementation of a verifier has also been developed, and this paper is intended to provide both a description of the safety dependence mechanism and a theoretical development of a type system based upon it.

In particular, our paper makes the following contributions:

1. We introduce a safe low-level imperative program representation that combines static single-assignment (SSA) form with explicit safety dependences, and we illustrate how it can be used to represent highly optimized code.
2. We present a simple type system to verify memory safety of programs in this representation. To the best of our knowledge, this type system is the first to formalize type checking in an SSA representation. While SSA is in some sense equivalent to CPS, the details are sufficiently different that our type system is quite unlike the usual lambda-calculus style type systems and required new proof techniques.
3. We demonstrate the utility of this program representation in a high-performance compiler, and we describe how a compiler can leverage its existing framework to preserve safety information. In particular, we demonstrate that only optimizations that directly affect memory safety, such as bounds check elimination and strength reduction of address calculations, require significant modification.

The remainder of the paper is organized as follows. In Section 2, we motivate the explicit representation of safety dependence in an optimizing compiler and describe how to do this via proof variables in a low-level imperative program representation. In Section 3, we describe a formal core language specifically dealing with array-bounds checks and present a type system with which we can verify programs in SSA form. In Section 4, we demonstrate how a compiler would lower a Java program to the core language and illustrate how aggressive compiler optimizations produce efficient and verifiable code. In Section 5, we informally describe extensions to our core language to capture complete Java functionality. In Section 6, we discuss the status of our current implementation, and, finally, in Sections 7 and 8 we discuss related work and conclude.

```
if (a!=null)
  while (!done) {
    b = (B)a;
    ··· = ··· b.x ···
    ···
  }
```

Figure 1. Field load in loop

2. Motivation

We define a *potentially unsafe instruction* as any instruction that, taken out of context, might fault or otherwise cause an illegal memory access at runtime. Some instructions, taken independently, are inherently unsafe. A load instruction may immediately fault if it accesses protected memory or may trigger an eventual crash by reading an incorrectly typed value. A store may corrupt memory with an illegal value (e.g., if an arbitrary integer replaces an object's virtual table).

Consider, for example, the field access in Figure 1. Assuming C++-like semantics, the operation $b.x$ dereferences memory with no guarantee of safety. In general, C++ does not guarantee that b refers to a real object of type B: b may hold an an integer that faults when used as a pointer.

Assuming Java semantics, however, the field access itself checks at runtime that b does not point to a null location. If the check succeeds, the field access executes the load; otherwise, it throws an exception, bypassing the load. By itself, this built-in check does not ensure safety: the load also depends on the preceding cast, which dynamically checks that the runtime type of b is in fact compatible with the type B. If the check succeeds, the cast executes the load; otherwise, it throws an exception, bypassing the load.

Typically, the safety of a potentially unsafe instruction depends on a set of control flow points. We refer to this form of dependence as *safety dependence*. In this example, the safety of the load depends on the cast that establishes its type. We call an instruction *contextually safe* when its corresponding safety dependences guarantee its safety. To verify the output of a compiler optimization, we must prove that each instruction is contextually safe.

2.1 Safety In Java

In Java and the verifiable subset of CLI, a combination of static verification and runtime checks guarantee the contextual safety of individual bytecode instructions. Static type checking establishes that variables have the appropriate primitive or object type. Runtime checks such as type tests (for narrowing operations), null pointer tests, and array bounds tests detect conditions that would cause a fault or illegal access and throw a language-level runtime exception instead.

Figure 2 shows Java-like bytecode instructions (using pseudo-registers in place of stack locations for clarity) for the code of Figure 1. The Java type system guarantees that variable b has type B at compile time, while the `getfield` instruction guarantees non-null access by testing for null at runtime. The check and the static verifier together guarantee that the load operation will not trigger an illegal memory access.

2.2 Safety in a Low-Level Representation

The Java bytecode format was not intended to be an intermediate program representation for an optimizing compiler. There are a number of reasons why such a format is not suitable, but here we

```
        ifnull a goto EXIT
L :
        ifeq done goto EXIT
        b := checkcast(a, B)
        t1 := getfield(b, B::x)
        ...
        goto L
EXIT :
```

Figure 2. Field load with Java-like bytecode

```
        if a = null goto EXIT
L :
        if done = 0 goto EXIT
        checkcast(a, B)
        checknull(a)
        t2 := getfieldaddr(a, B::x)
        t1 := ld(t2)
        ...
        goto L
EXIT :
```

Figure 3. Field load lowered in erasure-style representation

will focus only on those related to safety. First, bytecodes hide redundant check elimination opportunities. For example, in Figure 2, optimizations can eliminate the null check built into the `getfield` instruction because of the `ifnull` instruction. Even though several operations have built-in exception checks, programmers usually write their code to ensure that these checks never fail, so such optimization opportunities are common in Java programs.

Second, extraneous aliasing introduced to encode safety properties hides optimization opportunities. In Figures 1 and 2, variable b represents a copy of a that has the type B. Any use of a that requires this type information must use b instead. While this helps static verification, it hinders optimization. The field access must establish that b is not null, even though the `ifnull` statement establishes that property on a. To eliminate the extra check, a redundancy elimination optimization must reason about aliasing due to cast operations; this is beyond the capabilities of standard algorithms [16, 5].

In the absence of a mechanism for tracking safety dependences, STARJIT would lower a code fragment like this to one like that in Figure 3. Note that the `ld` operation is potentially unsafe and is safety dependent on the null check. In this case, however, the safety dependence between the null check and the load is not explicit. Although the instructions are still (nearly) adjacent in this code, there is no guarantee that future optimizations will leave them so. Figure 4 roughly illustrates the code that STARJIT would produce for our example. Redundant checks are removed by a combination of partial loop peeling (to expose redundant control flow) and common subexpression elimination. The invariant address field calculation is hoisted via code motion. In this case, the dependence of the load on the operations that guarantee its safety (specifically, the `if` and `checkcast` statements) has become obscured. We refer to this as an *erasure-style* low-level representation, as safety information is effectively erased from the program.

An alternative representation embeds safety information directly into the values and their corresponding types. The Java language already does this for type refinement via cast operations. This approach also applies to null checks, as shown in Figure 5. The SafeTSA representation takes this approach, extending it to array bounds checks [25, 2] as well. We refer to this as a *refinement-style* representation. In this representation, value dependences preserve the safety dependence between a check and a load. To preserve

```
        t2 := getfieldaddr(a, B::x)
        if a = null goto EXIT
        if done = 0 goto EXIT
        checkcast(a, B)
L :
        t1 := ld(t2)
        ...
        if done ≠ 0 goto L
EXIT :
```

Figure 4. Field load optimized in erasure-style representation

```
        if a = null goto EXIT
L :
        if done = 0 goto EXIT
        b := checkcast(a, B)
        t3 := checknull(b)
        t2 := getfieldaddr(t3, B::x)
        t1 := ld(t2)
        ...
        goto L
EXIT :
```

Figure 5. Field load lowered in refinement-style representation

safety, optimizations must preserve the value flow between the check and the load. Check elimination operations (such as the `checknull` in Figure 5) may be eliminated by optimization, but the values they produce (e.g., t_2) must be redefined in the process.

From an optimization standpoint, a refinement-style representation is not ideal. The safety dependence between the check and the load is not direct. Instead, it is threaded through the address field calculation, which is really just an addition operation. While the load itself cannot be performed until the null test, the address calculation is always safe. A code motion or instruction scheduling compiler optimization should be free to move it above the check if it is deemed beneficial. In Figure 3, it is clearly legal. In Figure 5, it is no longer possible. The refinement-style representation adds artificial constraints to the program to allow safety to be checked. In this case, the address calculation is artificially dependent on the check operation.

A refinement-style representation also obscures optimization opportunities by introducing multiple names for the same value. Optimizations that depend on syntactic equivalence of expressions (such as the typical implementation of redundancy elimination) become less effective. In Figure 3, a is syntactically compared to `null` twice. In Figure 5, this is no longer true. In general, syntactically equivalent operations in an erasure-style representation may no longer be syntactically equivalent in a refinement-style representation.

2.3 A Proof Passing Representation

Neither the erasure-style nor refinement-style representations precisely represent safety dependences. The erasure-style representation omits them altogether, while the refinement-style representation encodes them indirectly. As a result, the erasure-style representation is easy to optimize but difficult to verify, while the refinement-style is difficult to optimize but easy to verify.

To bridge this gap, we propose the use of a *proof passing* representation that encodes safety dependence directly into the program representation through proof variables. Proof variables act as capabilities for unsafe operations (similar to the capabilities of Walker et al. [26]). The availability of a proof variable represents the availability of a proof that a safety property holds. A potentially unsafe instruction must use an available proof variable to ensure

```
        [s1, s2] if a = null goto EXIT
L :
        if done = 0 goto EXIT
        s3 := checkcast(a, B)
        s4 := checknull(a)
        t2 := getfieldaddr(a, B::x)
        s5 := pfand(s3, s4)
        t1 := ld(t2) [s5]
        ...
        goto L
EXIT :
```

Figure 6. Field load lowered in a proof passing representation

```
        t2 := getfieldaddr(a, B::x)
        [s1, s2] if a = null goto EXIT
L :
        if done = 0 goto EXIT
        s3 := checkcast(a, B)
        s4 := s1
        s5 := pfand(s3, s4)
        t1 := ld(t2) [s5]
        ...
        goto L
EXIT :
```

Figure 7. Field load with CSE and Code Motion

contextual safety. This methodology relates closely to mechanisms proposed for certified code by Crary and Vanderwaart [10] and Shao et al. [23] in the context of the lambda calculus. We discuss the relationship of our approach to this work in Section 7.

Proof variables do not consume any physical resources at runtime: they represent abstract values and only encode safety dependences. Nevertheless, they are first-class constructs in our representation. They are generated by interesting control points and other relevant program points, and consumed by potentially unsafe instructions as operands guaranteeing safety. Most optimizations treat proof variables like other program variables.

Figure 6 demonstrates how we represent a load operation in a proof passing representation. As in Figure 5, we represent safety through value dependences, but instead of interfering with existing values, we insert new proof variables that directly model the safety dependence between the load and both check operations.

Figures 7 to 10 represent the relevant transformations performed by STARJIT to optimize this code. In Figure 7, we illustrate two optimizations. First, STARJIT's common subexpression elimination pass eliminates the redundant `checknull` operation. When STARJIT detects a redundant expression in the right hand side of an instruction, it replaces that expression with the previously defined variable. The `if` statement defines the proof variable s_1 if the test fails. This variable proves the proposition $a \neq \texttt{null}$. At the definition of s_4, the compiler detects that $a \neq \texttt{null}$ is available, and redefines s_4 to be a copy of s_1. STARJIT updates a redundant proof variable the same way as any other redundant variable.

Second, STARJIT hoists the definition of t_2, a loop invariant address calculation, above the loop. Even though the computed address may be invalid at this point, the address calculation is always safe; we require a proof of safety only on a memory operation that dereferences the address.

Figure 8 shows a step of copy propagation, which propagates s_1 into the load instruction and eliminates the use of s_4, allowing dead code elimination to remove the definition of s_4.

Figure 9 illustrates the use of partial loop peeling to expose redundant control flow operations within the loop. This transformation duplicates the test on $done$ and the checkcast operation, and makes the load instruction the new loop header. The proof variable s_3 is now defined twice, where each definition establishes that a has type B on its corresponding path. The compiler leverages SSA form to establish that the proof variable is available within the loop.

```
        t2 := getfieldaddr(a, B::x)
        [s1, s2] if a = null goto EXIT
L :
        if done = 0 goto EXIT
        s3 := checkcast(a, B)
        s5 := pfand(s3, s1)
        t1 := ld(t2) [s5]
        ...
        goto L
EXIT :
```

Figure 8. Field load with Copy Propagation

```
        t2 := getfieldaddr(a, B::x)
        [s1, s2] if a = null goto EXIT
        if done = 0 goto EXIT
        s3^1 := checkcast(a, B)
L :
        s3^2 := φ(s3^1, s3^3)
        s5 := pfand(s3^2, s1)
        t1 := ld(t2) [s5]
        ...
        if done = 0 goto EXIT
        s3^3 := checkcast(a, B)
        goto L
EXIT :
```

Figure 9. Field load with Partial Loop Peeling

```
        t2 := getfieldaddr(a, B::x)
        [s1, s2] if a = null goto EXIT
        if done = 0 goto EXIT
        s3 := checkcast(a, B)
        s5 := pfand(s3, s1)
L :
        t1 := ld(t2) [s5]
        ...
        if done ≠ 0 goto L
EXIT :
```

Figure 10. Field load with 2nd CSE and Branch Reversal

Finally, in Figure 10, another pass of common subexpression elimination eliminates the redundant `checkcast`. Copy propagation propagates the correct proof variable, this time through a redundant phi instruction. Note, that this final code is equivalent to the erasure-style representation in Figure 4 except that proof variables provide a direct representation of safety. In Figure 10, it is readily apparent that the `if` and `checkcast` statements establish the safety of the load instruction.

In the next section we formalize our approach as a small core language, and the following sections show its use and preservation across compiler optimizations and extension to full Java.

3. Core Language

In this section we describe a small language that captures the main ideas of explicit safety dependences through proof variables. As usual with core languages, we wish to capture just the essence of the problem and no more. The issue at hand is safety dependences, and to keep things simple we will consider just one such depen-

$(P, L_1, n_1, b.i) \mapsto (P, L_2, n_2, pc)$ where:

$P(b.i)$	L_2	n_2	pc	Side conditions
$\overline{p}$	$L_1\{\overline{x}_1 := L_1(\overline{x}_2)\}$	n_1	$b.(i+1)$	$\overline{p}[n_1] = \overline{x}_1 := \overline{x}_2$
$x : \tau := i$	$L_1\{x := i\}$	n_1	$b.(i+1)$	
$x_1 : \tau := x_2$	$L_1\{x_1 := L_1(x_2)\}$	n_1	$b.(i+1)$	
$x_1 : \tau := \texttt{newarray}(x_2, x_3)$	$L_1\{x_1 := v_1\}$	n_1	$b.(i+1)$	$L_1(x_2) = n, L_1(x_3) = v_3, v_1 = \langle \underbrace{v_3, \ldots, v_3}_{n} \rangle$
$x_1 : \tau := \texttt{newarray}(x_2, x_3)$	$L_1\{x_1 := v_1\}$	n_1	$b.(i+1)$	$L_1(x_2) = i, i < 0, v_1 = \langle\rangle$
$x_1 : \tau := \texttt{len}(x_2)$	$L_1\{x_1 := n\}$	n_1	$b.(i+1)$	$L_1(x_2) = \langle v_0, \ldots, v_{n-1} \rangle$
$x_1 : \tau := \texttt{base}(x_2)$	$L_1\{x_2 := v@0\}$	n_1	$b.(i+1)$	$L_1(x_2) = v, v = \langle \overline{v'} \rangle$
$x_1 : \tau := x_2$ *bop* x_3	$L_1\{x_1 := i_4\}$	n_1	$b.(i+1)$	$L_1(x_2) = i_2,\ L_1(x_3) = i_3,\ i_4 = i_2$ *bop* i_3
$x_1 : \tau := x_2$ *bop* x_3	$L_1\{x_1 := v@i_4\}$	n_1	$b.(i+1)$	$L_1(x_2) = v@i_2, L_1(x_3) = i_3, i_4 = i_2$ *bop* i_3
$x_1 : \tau := \texttt{ld}(x_2)\ [x_3]$	$L_1\{x_1 := v_i\}$	n_1	$b.(i+1)$	$L_1(x_2) = \langle v_0, \ldots, v_n \rangle @i, 0 \le i \le n$
$x_1 : \tau := \texttt{pffact}(x_2)$	$L_1\{x_1 := \texttt{true}\}$	n_1	$b.(i+1)$	
$x : \tau := \texttt{pfand}(\overline{y})$	$L_1\{x := \texttt{true}\}$	n_1	$b.(i+1)$	
$[x_1 : \tau_1, x_2 : \tau_2]$ **if** x_3 *rop* x_4 **goto** b'	$L_1\{x_1 := \texttt{true}\}$	$\texttt{edge}_P(b, b+1)$	$(b+1).0$	$L_1(x_3) = i_3, L_1(x_4) = i_4, \neg(i_3$ *rop* $i_4)$
$[x_1 : \tau_1, x_2 : \tau_2]$ **if** x_3 *rop* x_4 **goto** b'	$L_1\{x_2 := \texttt{true}\}$	$\texttt{edge}_P(b, b')$	$b'.0$	$L_1(x_3) = i_3, L_1(x_4) = i_4, i_3$ *rop* i_4
goto b'	L_1	$\texttt{edge}_P(b, b')$	$b'.0$	

Figure 11. Operational semantics

dence, namely, bounds checking for arrays. In particular, we consider a compiler with separate address arithmetic, load, and store operations, where the type system must ensure that a load or store operation is applied only to valid pointers. Moreover, since the basic safety criteron for a store is the same as for a load, namely, that the pointer is valid, we consider only loads; adding stores to our core language adds no interesting complications. Not considering stores further allows us to avoid modelling the heap explicitly, but to instead use a substitution semantics which greatly simplifies the presentation.

The syntax of our core language is given as follows:

Prog. States	S	::=	(P, L, n, pc)
Programs	P	::=	$\overline{B}$
Blocks	B	::=	$\overline{p}; \overline{\iota}; c$
Phi Instructions	p	::=	$x : \tau := \phi(\overline{x})$
Instructions	ι	::=	$x : \tau := r$
Right-hand sides	r	::=	$i \mid x \mid \texttt{newarray}(x_1, x_2) \mid$ $\texttt{len}(x) \mid \texttt{base}(x) \mid$ x_1 *bop* $x_2 \mid \texttt{ld}(x_1)\ [x_2] \mid$ $\texttt{pffact}(x) \mid \texttt{pfand}(\overline{x})$
Binary Ops	*bop*	::=	$+ \mid -$
Transfers	c	::=	$\texttt{goto}\ n \mid \texttt{halt} \mid$ $[x_1 : \tau_1, x_2 : \tau_2]\ \texttt{if}\ x_3$ *rop* x_4 $\texttt{goto}\ n$
Relations	*rop*	::=	$< \mid \le \mid = \mid \neq$
Environments	L	::=	$\overline{x} := \overline{v}$
Values	v	::=	$i \mid \langle \overline{v} \rangle \mid \langle \overline{v} \rangle @ i \mid \texttt{true}$
Prog. Counters	pc	::=	$n_1.n_2$

Here i ranges over integer constants, x ranges over variables, n ranges over natural numbers, and ϕ is the phi-operation of SSA. We use the bar notation introduced in Featherweight Java [15]: $\overline{B}$ abbreviates $B_0, \ldots, B_n$, $\overline{x} := \overline{v}$ abbreviates $x_0 := v_0, \ldots, x_n := v_n$, *et cetera*. We also use the bar notation in type rules to abbreviate a sequence of typing judgements in the obvious way. In addition to the grammar above, programs are subject to a number of context-sensitive restrictions. In particular, the n in $[x_1 : \tau_1, x_2 : \tau_2]$ **if** x_3 *rop* x_4 **goto** n and **goto** n must be a block number in the program (i.e., if the program is $B_0, \ldots, B_m$ then $0 \le n \le m$); the transfer in the last block must be a goto or halt; the number of variables in a phi instruction must equal the number of incoming edges (as defined below) to the block in which it appears; the variables assigned in the phi instructions of a block must be distinct.

Informally, the key features of our language are the following. The operation $\texttt{base}(x)$ takes an array and creates a pointer to the element at index zero. The arithmetic operations can be applied to such pointers and an integer to compute a pointer to a different index. The $\texttt{ld}(x_1)\ [x_2]$ operation loads the value pointed to by the pointer in x_1. The variable x_2 is a proof variable and conceptually contains a proof that x_1 is a valid pointer: that is, that it points to an in-bounds index. The typing rules ensure that x_1 is valid by requiring x_2 to contain an appropriate proof. The operations $\texttt{pffact}(x)$ and $\texttt{pfand}(\overline{x})$ construct proofs. For $\texttt{pffact}(x)$ a proof of a formula based on the definition of x is constructed. For example, if x's definition is $x : \texttt{int} := \texttt{len}(y)$ then $\texttt{pffact}(x)$ constructs a proof of $x = \texttt{len}(y)$. A complete definition of the defining facts of instructions appears in Figure 14. For $\texttt{pfand}(x_1, \ldots, x_n)$, x_1 through x_n are also proof variables, and a proof of the conjunction is returned. Values of the form $\langle v_0, \ldots, v_n \rangle @ i$ represent pointers to array elements: in this case a pointer to the element at index i of an array of type $\langle v_0, \ldots, v_n \rangle$. Such a pointer is valid if i is in bounds (that is, if $0 \le i \le n$) and invalid otherwise. The typing rules must ensure that only valid pointers are loaded from, with proof variables used to provide evidence of validity. The final unusual aspect of the language is that branches assign proofs to proof variables that reflect the condition being branched on. For example, in the branch $[x_1 : \tau_1, x_2 : \tau_2]$ **if** $x_3{=}x_4$ **goto** n, a proof of $x_3 \neq x_4$ is assigned to x_1 along the fall-through edge, and a proof of $x_3 = x_4$ is assigned to x_2 along the taken edge. These proofs can then be used to discharge validity requirements for pointers.

To state the operational semantics and type system we need a few definitions. The program counters of a program $\texttt{pcs}(P)$ are $\{b.i \mid P = B_0, \ldots, B_m \wedge b \le m \wedge B_b = \overline{p}; \iota_1; \cdots; \iota_n; c \wedge i \le n{+}1\}$. We write $P(b)$ for B_b when $P = B_0, \ldots, B_n$ and $b \le n$; if $P(b) = \overline{p}; \iota_1; \ldots; \iota_m; c$ then $P(b.n)$ is $\overline{p}$ when $n = 0$, and ι_n when $1 \le n \le m$ and c when $n = m + 1$. The edges of a program P, $\texttt{edges}(P)$, are as follows. The entry edge is $(-1, 0)$. If $P(n)$ ends in $[x_1 : \tau_1, x_2 : \tau_2]$ **if** x_3 *rop* x_4 **goto** n' then there are edges $(n, n{+}1)$, called the fall-through edge, and (n, n'), called the taken edge. If $P(n)$ ends in **goto** n' then there is an edge (n, n'). For a given P and n_2 the edges $(n_1, n_2) \in \texttt{edges}(P)$ are numbered from zero in the order given by n_1; $\texttt{edge}_P(n_1, n_2)$ is this number, also called the incoming edge number of (n_1, n_2) into n_2.

Operational Semantics A program P is started in the state $(P, \emptyset, 0, 0.0)$. The reduction relation that maps one state to the next is given in Figure 11. Note that the third component of a pro-

gram state tracks which incoming edge led to the current program counter—initially this is the entry edge $(-1, 0)$, and is updated by transfers. It is used by phi instructions to select the correct variable. The notation $\overline{p}[i]$ denotes $x_1 := x_{1i}, \ldots, x_n := x_{ni}$ when $\overline{p} = x_1 : \tau_1 := \phi(x_{11}, \ldots, x_{1m}), \ldots, x_n : \tau_n := \phi(x_{n1}, \ldots, x_{nm})$. A program terminates when in a state of the form (P, L, n, pc) where $P(pc) = \texttt{halt}$. A program state is stuck if it is irreducible and not a terminal state. Stuck states all represent type errors that the type system should prevent. Note that the array creation operation must handle negative sizes. Our implementation would throw an exception, but since the core language does not have exceptions, it simply creates a zero length array if a negative size is requested.

In the operational semantics, the proof type has the single inhabitant `true`, upon which no interesting operations are defined. Proofs in this sense are equivalent to unit values for which non-escaping occurrences can be trivially erased when moving to an untyped setting. This "proof erasure" property is precisely analogous to the "coercion erasure" property of the coercion language of Vanderwaart et al. [24]. In practice, uses of proof variables in the STARJIT compiler are restricted such that all proof terms can be elided during code generation and consequently impose no overhead at run time. While we believe that it would be straightforward to formalize the syntactic restrictions that make this possible, we choose for the sake of simplicity to leave this informal here.

Type System The type system has two components: the SSA property and a set of typing judgements. The SSA property ensures both that every variable is assigned to at most once in the program text (the single assignment property) and that all uses of variables are dominated by definitions of those variables. In a conventional type system, these properties are enforced by the typing rules. In particular, the variables that are listed in the context of the typing judgement are the ones that are in scope. For SSA IRs, it is more convenient to check these properties separately.

The type checker must ensure that during execution each use of a variable is preceded by an assignment to that variable. Since the i-th variable of a phi instruction is used only if the i-th incoming edge was used to get to the block, and the proof variables in an if transfer are assigned only on particular out-going edges, we give a rather technical definition of points at which variables are assigned or used. These points are such that a definition point dominating a use point implies that assignment will always precede use. These points are based on an unconventional notion of control-flow graph, to avoid critical edges which might complicate our presentation. For a program P with blocks 0 to m, the control-flow graph consists of the nodes $\{0, \ldots, m\} \cup \mathsf{edges}(P)$ and edges from each original node n to each original edge (n, n') and similarly from (n, n') to n'. The definition/use points, $\mathsf{du}(P)$, are $\mathsf{pcs}(P) \cup \{b.0.i \mid P(b.0) = p_0, \ldots, p_n \wedge 0 \leq i \leq n\} \cup \{e.i \mid e \in \mathsf{edges}(P) \wedge i \in \{0, 1\}\}$.

Figure 13 gives the formal definition of dominance, definition/use points, and the SSA property.

The syntax of types is:

Types	τ	$::= \texttt{int} \mid \texttt{array}(\tau) \mid \texttt{ptr}_?\langle\tau\rangle \mid \mathtt{S}(x) \mid \texttt{pf}_{(F)}$
Facts	F	$::= e_1 \mathit{\ rop\ } e_2 \mid F_1 \wedge F_2$
Fact Exps.	e	$::= i \mid x \mid \texttt{len}(x) \mid e_1 \mathit{\ bop\ } e_2 \mid x@e$
Environments	Γ	$::= \overline{x} : \overline{\tau}$

The type $\texttt{ptr}_?\langle\tau\rangle$ is given to pointers that, if valid, point to values with type τ (the ? indicates that they might not be valid). The singleton type $\mathtt{S}(x)$ is given to things that are equal to x. The type $\texttt{pf}_{(F)}$ is given to proof variables that contain a proof of the fact F. Facts include arithmetic comparisons and conjunction. Fact expressions include integers, variables, array lengths, arithmetic operations, and a subscript expression—the fact expression $x@e$ stands for a pointer that points to the element at index e of array x.

Judgement	Meaning
$\Gamma \vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2 in Γ
$\vdash F_1 \implies F_2$	F_1 implies F_2
$\Gamma \vdash \overline{p}$	$\overline{p}$ is safe in environment Γ
$\Gamma \vdash_P \iota$	ι is safe in environment Γ
$\Gamma \vdash c$	c is safe in environment Γ
$\vdash_P \tau$ at du	τ well-formed type at du in P
$\vdash_P \Gamma$	environment Γ well-formed in P
$\vdash P$	P is safe

Figure 12. Typing judgements

The judgements of the type system are given in figure 12. Most of the typing rules are given in Figure 14. Typing environments Γ state the types that variables are supposed to have. The rules check that when assignments are made to a variable, the type of the assigned value is compatible with the variable's type. For example, the judgement $\Gamma \vdash \texttt{int} \leq \Gamma(x)$ in the rule for $x : \tau := i$ checks that integers are compatible with the type of x. The rules also check that uses of a variable have a type compatible with the operation. For example, the rule for load expects a proof that the pointer, x_2, is valid, so the rule checks that x_3's type $\Gamma(x_3)$ is a subtype of $\texttt{pf}_{(x@0 \leq x_2 \wedge x_2 < x@\texttt{len}(x))}$ for some x. It is this check along with the rules for proof value generation and the SSA property that ensure that x_2 is valid.

Given these remarks, the only other complicated rule is for phi instructions. In a loop a phi instruction might be used to combine two indices, and the compiler might use another phi instruction to combine the proofs that these indices are in bounds. For example, consider this sequence:

$$x_1 : \texttt{int} := \phi(x_2, x_3)$$
$$y_1 : \texttt{pf}_{(0 \leq x_1)} := \phi(y_2, y_3)$$

where $y_2 : \texttt{pf}_{(0 \leq x_2)}$ and $y_3 : \texttt{pf}_{(0 \leq x_3)}$. Here the types for y_1, y_2, and y_3 are different and in some sense incompatible, but are intuitively the correct types. The rule for phi instructions allows this typing. In checking that y_2 has a compatible type, the rule substitutes x_2 for x_1 in y_1's type to get $\texttt{pf}_{(0 \leq x_2)}$, which is the type that y_2 has; similarly for y_3.

For a program P that satisfies the SSA property, every variable mentioned in the program has a unique definition point, and that definition point is decorated with a type. Let $\mathsf{vt}(P)$ denote the environment formed from extracting these variable/type pairs. A program P is *well formed* ($\vdash P$) if:

1. P satisfies the SSA property,
2. $\vdash_P \mathsf{vt}(P)$,
3. $\mathsf{vt}(P) \vdash \overline{p}$ for every $\overline{p}$ in P,
4. $\mathsf{vt}(P) \vdash_P \iota$ for every instruction ι in P, and
5. $\mathsf{vt}(P) \vdash c$ for every transfer c in P.

The type system is safe:

THEOREM 1 (Type Safety).
If $\vdash P$ *and* $(P, \emptyset, 0, 0.0) \mapsto^* S$ *then* S *is not stuck.*

A proof of this theorem appears in the companion technical report [17]. The proof takes the standard high-level form of showing preservation and progress lemmas, as well as some lemmas particular to an SSA language. It is important to note that safety of the type system is contingent on the soundness of the decision procedure for $\vdash F_1 \implies F_2$. In the proof, a judgement corresponding to truth of facts in an environment is given. In this setting, the assumption of logical soundness corresponds to the restriction that in any environment in which F_1 is true, F_2 is also true.

Defs and Uses:
If $P(b.i) = x : \tau := r$ then program counter $b.i$ defines x, furthermore, $b.i$ is a use of the ys where r has the following forms:

$$y \mid \texttt{newarray}(y_1, y_2) \mid \texttt{len}(y) \mid \texttt{base}(y) \mid y_1 \; \mathit{bop} \; y_2 \mid \texttt{ld}(y_1)\;[y_2] \mid \texttt{pffact}(y) \mid \texttt{pfand}(\overline{y})$$

If $P(b.i) = (p_0, \ldots, p_n)$ and $p_j = x_j : \tau_j := \phi(y_{j1}, \ldots, y_{jm})$ then $b.i.j$ defines each x_j and $e_k.1$ uses each y_{jk} where e_k is the k-th incoming edge of b. If $P(b.i) = [x_1 : \tau_1, x_2 : \tau_2]\ \texttt{if}\ y_1\ \mathit{rop}\ y_2\ \texttt{goto}\ n$ then $e_1.0$ defines x_1 and $e_2.0$ defines x_2 where e_1 and e_2 are the fall-through and taken edges respectively, and $b.i$ uses y_1 and y_2. If x has a unique definition/use point in P that defines it, then $\mathsf{def}_P(x)$ is this point.

Dominance:

- In program P, node n dominates node m, written $\mathsf{dom}_P(n, m)$, if every path in the control-flow graph of P from $(-1, 0)$ to m includes n.
- In program P, definition/use point $n_1.i_1$ strictly dominates definition/use point $n_2.i_2$, written $\mathsf{sdom}_P(n_1.i_1, n_2.i_2)$ if $n_1 = n_2$ and $i_1 < i_2$ (here i_1 or i_2 might be a dotted pair $0.j$, so we take this inequality to be lexicographical ordering) or $n_1 \neq n_2$ and $\mathsf{dom}_P(n_1, n_2)$.

Single Assignment:
A program satisfies the *single-assignment property* if every variable is defined by at most one definition/use point in that program.

In Scope:
A program P satisfies the *in-scope property* if for every definition/use point du_1 that uses a variable there is a definition/use point du_2 that defines that variable and $\mathsf{sdom}_P(du_2, du_1)$.

SSA:
A program satisfies the *Single Static Assignment (SSA) property* if it satisfies the single-assignment and in-scope properties. Note that a program that satisfies SSA has a unique definition for each variable mentioned in the program.

Figure 13. SSA definitions

$\boxed{\vdash_P \tau\ \mathsf{at}\ du \quad \vdash_P \Gamma}$

$$\frac{\mathsf{fv}(\tau) \subseteq \mathit{inscope}_P(du)}{\vdash_P \tau\ \mathsf{at}\ du} \qquad \frac{\vdash_P \overline{\tau}\ \mathsf{at}\ \mathsf{def}_P(\overline{x})}{\vdash_P \overline{x} : \overline{\tau}}$$

$\boxed{\Gamma \vdash \tau_1 \leq \tau_2 \quad \vdash F_1 \implies F_2}$

$$\frac{}{\Gamma \vdash \texttt{int} \leq \texttt{int}} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash \texttt{array}(\tau_1) \leq \texttt{array}(\tau_2)} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash \texttt{ptr}_?\langle\tau_1\rangle \leq \texttt{ptr}_?\langle\tau_2\rangle}$$

$$\frac{}{\Gamma \vdash \mathtt{S}(x) \leq \mathtt{S}(x)} \qquad \frac{}{\Gamma \vdash \mathtt{S}(x) \leq \Gamma(x)} \qquad \frac{\vdash F_1 \implies F_2}{\Gamma \vdash \texttt{pf}_{(F_1)} \leq \texttt{pf}_{(F_2)}} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_2 \quad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3}$$

The judgement $\vdash F_1 \implies F_2$ is some appropriate decision procedure for our fact language.

$\boxed{\Gamma \vdash \overline{p} \quad \Gamma \vdash_P \iota \quad \Gamma \vdash c}$

$$\frac{\Gamma \vdash \mathtt{S}(x_{ij}) \leq \Gamma(x_i)\{x_1, \ldots, x_n := x_{1j}, \ldots, x_{nj}\}}{\Gamma \vdash x_1 : \tau_1 := \phi(x_{11}, \ldots, x_{1m}), \ldots, x_n : \tau_n := \phi(x_{n1}, \ldots, x_{nm})}$$

$$\frac{\Gamma \vdash \texttt{int} \leq \Gamma(x)}{\Gamma \vdash_P x : \tau := i} \qquad \frac{\Gamma \vdash \mathtt{S}(x_2) \leq \Gamma(x_1)}{\Gamma \vdash_P x_1 : \tau := x_2} \qquad \frac{\Gamma \vdash \Gamma(x_2) \leq \texttt{int} \quad \Gamma \vdash \texttt{array}(\Gamma(x_3)) \leq \Gamma(x_1)}{\Gamma \vdash_P x_1 : \tau := \texttt{newarray}(x_2, x_3)}$$

$$\frac{\Gamma \vdash \Gamma(x_2) \leq \texttt{array}(\tau_2) \quad \Gamma \vdash \texttt{int} \leq \Gamma(x_1)}{\Gamma \vdash_P x_1 : \tau := \texttt{len}(x_2)} \qquad \frac{\Gamma \vdash \Gamma(x_2) \leq \texttt{array}(\tau_2) \quad \Gamma \vdash \texttt{ptr}_?\langle\tau_2\rangle \leq \Gamma(x_1)}{\Gamma \vdash_P x_1 : \tau := \texttt{base}(x_2)}$$

$$\frac{\Gamma \vdash \Gamma(x_2) \leq \texttt{int} \quad \Gamma \vdash \Gamma(x_3) \leq \texttt{int} \quad \Gamma \vdash \texttt{int} \leq \Gamma(x_1)}{\Gamma \vdash_P x_1 : \tau := x_2\ \mathit{bop}\ x_3} \qquad \frac{\Gamma \vdash \Gamma(x_2) \leq \texttt{ptr}_?\langle\tau_2\rangle \quad \Gamma \vdash \Gamma(x_3) \leq \texttt{int} \quad \Gamma \vdash \texttt{ptr}_?\langle\tau_2\rangle \leq \Gamma(x_1)}{\Gamma \vdash_P x_1 : \tau := x_2\ \mathit{bop}\ x_3}$$

$$\frac{\Gamma \vdash \Gamma(x_2) \leq \texttt{ptr}_?\langle\tau_2\rangle \quad \Gamma \vdash \Gamma(x_3) \leq \texttt{pf}_{(x@0 \leq x_2 \wedge x_2 < x@\texttt{len}(x))} \quad \Gamma \vdash \tau_2 \leq \Gamma(x_1)}{\Gamma \vdash_P x_1 : \tau := \texttt{ld}(x_2)\ [x_3]}$$

$$\frac{\Gamma \vdash \texttt{pf}_{(\mathsf{deffact}_P(x_2))} \leq \Gamma(x_1)}{\Gamma \vdash_P x_1 : \tau := \texttt{pffact}(x_2)} \qquad \frac{\Gamma \vdash \Gamma(y_1) \leq \texttt{pf}_{(F_1)} \quad \cdots \quad \Gamma \vdash \Gamma(y_n) \leq \texttt{pf}_{(F_n)} \quad \Gamma \vdash \texttt{pf}_{(F_1 \wedge \cdots \wedge F_n)} \leq \Gamma(x_1)}{\Gamma \vdash_P x : \tau := \texttt{pfand}(y_1, \ldots, y_n)}$$

$$\frac{\Gamma \vdash \Gamma(x_3) \leq \texttt{int} \quad \Gamma \vdash \Gamma(x_4) \leq \texttt{int} \quad \Gamma \vdash \texttt{pf}_{(\neg(x_3\ \mathit{rop}\ x_4))} \leq \Gamma(x_1) \quad \Gamma \vdash \texttt{pf}_{(x_3\ \mathit{rop}\ x_4)} \leq \Gamma(x_2)}{\Gamma \vdash [x_1 : \tau_1, x_2 : \tau_2]\ \texttt{if}\ x_3\ \mathit{rop}\ x_4\ \texttt{goto}\ n}$$

$$\frac{}{\Gamma \vdash \texttt{goto}\ n} \qquad \frac{}{\Gamma \vdash \texttt{halt}}$$

$\boxed{\mathsf{deffact}_P(x)}$ The fact $\mathsf{deffact}_P(x)$ depends upon the defining instruction of x in P, and is given by these rules:

$$\begin{array}{lcl}
\mathsf{deffact}_P(x : \tau := i) & = & x{=}i \\
\mathsf{deffact}_P(x : \tau := \texttt{len}(x')) & = & x{=}\texttt{len}(x') \\
\mathsf{deffact}_P(x : \tau := \texttt{base}(x')) & = & x = x'@0 \\
\mathsf{deffact}_P(x : \tau := x_1\ \mathit{bop}\ x_2) & = & x{=}x_1\ \mathit{bop}\ x_2
\end{array}$$

Figure 14. Typing rules

The typing rules presented are for the most part syntax-directed, and can be made algorithmic. A consideration is that the rule for load must determine the actual array variable, which is not apparent from the conclusion. In general, the decision prodecure only needs to verify that the rule holds for one of the arrays available at that program point. In practice, the correct array can be inferred by examining the type of the proof variable. We believe that judgements on facts may be efficiently decided by an integer linear programming tool such as the Omega Calculator [22] with two caveats. First, such tools reason over $\mathbb{Z}$ rather than 32- or 64-bit integers. Second, they restrict our fact language for integer relations (and, thus, compiler reasoning) to affine expressions. This is, however, sufficient to capture current STARJIT optimizations.

4. Compiler optimizations

In this section we examine compiler optimizations in the context of the core language. We demonstrate how an optimizing compiler can preserve both proof variables and their type information. We argue that our ideas greatly simplify this process. In previous work, an implementer would need to modify each optimization to update safety information. In our representation, we leverage existing compiler infrastructure to do the bulk of the work. In particular, most control-flow or data-flow optimizations require virtually no changes at all. Others that incorporate algebraic properties only need to be modified to record the compiler's reasoning. In the next section we will discuss how these ideas can be extended from the core language to full Java.

In general, there are two ways in which an optimization can maintain the correctness of the proofs embedded in the program. First, it can apply the transformation to both computation and proof simultaneously. This is sufficient for the majority of optimizations. Second, it can create new proofs for the facts provided by the original computation. As we show below, this is necessary for the few optimizations that infer new properties that affect safety. In the rest of this section we show how these general principles apply to individual compiler optimizations on a simple example. For this example, we show how to generate a low-level intermediate representation that contains safety information and how to preserve this information through several compiler optimizations, such as loop invariant code motion, common subexpression elimination, array bounds check elimination, strength reduction of array element pointer, and linear function test replacement.

The example we will consider, in pseudo code, is:

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

Where we assume that a is a non-null integer array, that a is not modified in the loop, and that the pseudo code array subscripting has an implicit bounds check. Although this example does not reflect the full complexity of Java, it is sufficient to illustrate the main ideas of propagating safety information through the compiler optimizations. Section 5 discusses additional issues in addressing full Java.

The first compilation step for our example lowers the program into a low-level representation suitable for optimization, as shown in Figure 15. In our system, lowering generates instructions that express the computation and any required proofs of the computation's safety. For example, a typical compiler would expand an array element access $a[i]$ into the following sequence: array bounds checks, computation of the array element address, and a potentially unsafe load from that address. In our system, the compiler also generates proof variables that show that the array index i is within the array bounds (q_4 for the lower bound and q_6 for the upper bound) and that the load accesses an element i of the array a (proof variable q_9). The conjunction of these proofs is sufficient to type check the load instruction according to the typing rules in Figure 14. The proof variables are generated by the explicit array bounds checks (which we use as syntactic sugar for the branches that transfer control to a halt instruction if the bounds check fails) and by `pffact` and `pfand` statements that encode arithmetic properties of the address computation as the types of proof variables.

$i_1 : \texttt{int}$	$:= 0$
$uB : \texttt{int}$	$:= \texttt{len}(a)$
$LOOP$:	
$i_2 : \texttt{int}$	$:= \phi(i_1, i_3)$
$[q_1 : \texttt{pf}_{(i_2 < uB)}, q_2 : \ldots] :=$	if $uB \leq i_2$ goto $EXIT$
$aLen : \texttt{int}$	$:= \texttt{len}(a)$
$q_3 : \texttt{pf}_{(aLen = \texttt{len}(a))}$	$:= \texttt{pffact}(aLen)$
$q_4 : \texttt{pf}_{(0 \leq i_2)}$	$:= \texttt{checkLowerBound}(i_2, 0)$
$q_5 : \texttt{pf}_{(i_2 < aLen)}$	$:= \texttt{checkUpperBound}(i_2, aLen)$
$q_6 : \texttt{pf}_{(i_2 < \texttt{len}(a))}$	$:= \texttt{pfand}(q_3, q_5)$
$aBase : \texttt{ptr}_?\langle\texttt{int}\rangle$	$:= \texttt{base}(a)$
$q_7 : \texttt{pf}_{(aBase = a@0)}$	$:= \texttt{pffact}(aBase)$
$addr : \texttt{ptr}_?\langle\texttt{int}\rangle$	$:= aBase + i_2$
$q_8 : \texttt{pf}_{(addr = aBase + i_2)}$	$:= \texttt{pffact}(addr)$
$q_9 : \texttt{pf}_{(addr = a@i_2)}$	$:= \texttt{pfand}(q_7, q_8)$
$q_{10} : \texttt{pf}_{(a@0 \leq addr < a@\texttt{len}(a))}$	$:= \texttt{pfand}(q_4, q_6, q_9)$
$val : \texttt{int}$	$:= \texttt{ld}(addr)\ [q_{10}]$
$\ldots : \ldots$	$:= val$
$i_3 : \texttt{int}$	$:= i_2 + 1$
	goto $LOOP$
$EXIT$:	
	$\ldots$

Figure 15. Low-level representation for array load in loop

$i_1 : \texttt{int}$	$:= 0$
$uB : \texttt{int}$	$:= \texttt{len}(a)$
$q_3 : \texttt{pf}_{(uB = \texttt{len}(a))}$	$:= \texttt{pffact}(uB)$
$aBase : \texttt{ptr}_?\langle\texttt{int}\rangle$	$:= \texttt{base}(a)$
$q_7 : \texttt{pf}_{(aBase = a@0)}$	$:= \texttt{pffact}(aBase)$
$LOOP$:	
$i_2 : \texttt{int}$	$:= \phi(i_1, i_3)$
$[q_1 : \texttt{pf}_{(i_2 < uB)}, q_2 : \ldots] :=$	if $uB \leq i_2$ goto $EXIT$
$q_4 : \texttt{pf}_{(0 \leq i_2)}$	$:= \texttt{checkLowerBound}(i_2, 0)$
$q_5 : \texttt{pf}_{(i_2 < uB)}$	$:= \texttt{checkUpperBound}(i_2, uB)$
$q_6 : \texttt{pf}_{(i_2 < \texttt{len}(a))}$	$:= \texttt{pfand}(q_3, q_5)$
$addr : \texttt{ptr}_?\langle\texttt{int}\rangle$	$:= aBase + i_2$
$q_8 : \texttt{pf}_{(addr = aBase + i_2)}$	$:= \texttt{pffact}(addr)$
$q_9 : \texttt{pf}_{(addr = a@i_2)}$	$:= \texttt{pfand}(q_7, q_8)$
$q_{10} : \texttt{pf}_{(a@0 \leq addr < a@\texttt{len}(a))}$	$:= \texttt{pfand}(q_4, q_6, q_9)$
$val : \texttt{int}$	$:= \texttt{ld}(addr)\ [q_{10}]$
$\ldots : \ldots$	$:= val$
$i_3 : \texttt{int}$	$:= i_2 + 1$
	goto $LOOP$
$EXIT$:	
	$\ldots$

Figure 16. IR after CSE and loop invariant code motion

Next, we take the example in Figure 15 through several common compiler optimizations that are employed by STARJIT to generate efficient code for loops iterating over arrays (Figures 16 - 19). The result is highly-optimized code with an embedded proof of program safety.

We start, in Figure 16, by applying several basic data-flow optimizations such as CSE, dead code elimination, and loop invariant code motion. An interesting property of these optimizations in our system is that they require no modification to preserve the

```
    i1 : int                                  := 0
    q11 : pf(i1=0)                            := pffact(i1)
    uB : int                                  := len(a)
    q3 : pf(uB=len(a))                        := pffact(uB)
    aBase : ptr?⟨int⟩                         := base(a)
    q7 : pf(aBase=a@0)                        := pffact(aBase)
LOOP :
    i2 : int                                  := φ(i1, i3)
    q4 : pf(0≤i2)                             := φ(q11, q13)
    [q1 : pf(i2<uB), q2 : ...] :=             if uB≤i2 goto EXIT
    q6 : pf(i2<len(a))                        := pfand(q3, q1)
    addr : ptr?⟨int⟩                          := aBase+i2
    q8 : pf(addr=aBase+i2)                    := pffact(addr)
    q9 : pf(addr=a@i2)                        := pfand(q7, q8)
    q10 : pf(a@0≤addr<a@len(a))               := pfand(q4, q6, q9)
    val : int                                 := ld(addr) [q10]
    ... : ...                                 := val
    i3 : int                                  := i2+1
    q12 : pf(i3=i2+1)                         := pffact(i3)
    q13 : pf(0≤i3)                            := pfand(q4, q12)
                                              goto LOOP
EXIT :
                                              ...
```

Figure 17. IR after bound check elimination

safety proofs. They treat proof variables identically to other terms, and, thus, are automatically applied to both the computation and the proofs. For example, common subexpression elimination and copy propagation replace *all* occurrences of *aLen* with *uB*, including those that occur in proof types. The type of the proof variable q_3 is updated to match its new definition $\texttt{pffact}(uB)$.

In Figure 17, we illustrate array bounds check elimination. In the literature [4], this optimization is typically formulated to remove redundant bounds checks without leaving any trace of its reasoning in the program. In such an approach, a verifier must effectively repeat the optimization reasoning to prove program safety. In our system, an optimization cannot eliminate an instruction that defines a proof variable without constructing a new definition for that variable or removing all uses of that variable. Intuitively, the compiler must record in a new definition its reasoning about why the eliminated instruction was redundant. Consider the bounds checks in Figure 16. The lower bound check that verifies that $0{\leq}i_2$ is redundant because i_2 is a monotonically increasing variable with the initial value 0. Formally, the facts that $i_1{=}0$, $i_2{=}\phi(i_1, i_3)$ and $i_3{=}i_2{+}1$ imply that $0{\leq}i_2$. This reasoning is recorded in the transformed program through a new definition of the proof variable q_4 and the additional proof variables q_{11} and q_{13}. We use SSA to connect these proofs at the program level. The upper bound check that verifies that $i_2{<}\texttt{len}(a)$ (proof variable q_5) is redundant because the `if` statement guarantees the same condition (proof variable q_1). Because the new proof for the fact q_5 is already present in the program, the compiler simply replaces all uses of of q_5 with q_1.

In Figure 18, we perform operator strength reduction (OSR) [9] to find a pointer that is an affine expression of a monotonically increasing or decreasing loop index variable and to convert it into an independent induction variable. In our example, OSR eliminates *i* from the computation of *addr* by incrementing it directly. Because variable *addr* is used in the $q_8 := \texttt{pffact}(addr)$ statement, the compiler cannot modify the definition of *addr* without also modifying the definition of q_8 (otherwise, the transformed program would not type check). Informally, the compiler must reestablish the proof that the fact trivially provided by the original definition still holds. In our system, OSR is modified to construct a new proof for the fact trivially implied by the original pointer definition by

```
    i1 : int                                  := 0
    q11 : pf(i1=0)                            := pffact(i1)
    uB : int                                  := len(a)
    q3 : pf(uB=len(a))                        := pffact(uB)
    aBase : ptr?⟨int⟩                         := base(a)
    q7 : pf(aBase=a@0)                        := pffact(aBase)
    addr1 : ptr?⟨int⟩                         := aBase+i1
    q14 : pf(addr1=aBase+i1)                  := pffact(addr1)
LOOP :
    i2 : int                                  := φ(i1, i3)
    q4 : pf(0≤i2)                             := φ(q11, q13)
    addr2 : ptr?⟨int⟩                         := φ(addr1, addr3)
    q8 : pf(addr2=aBase+i2)                   := φ(q14, q16)
    [q1 : pf(i2<uB), q2 : ...] :=             if uB≤i2 goto EXIT
    q6 : pf(i2<len(a))                        := pfand(q3, q1)
    q9 : pf(addr2=a@i2)                       := pfand(q7, q8)
    q10 : pf(a@0≤addr<a@len(a))               := pfand(q4, q6, q9)
    val : int                                 := ld(addr2) [q10]
    ... : ...                                 := val
    i3 : int                                  := i2+1
    q12 : pf(i3=i2+1)                         := pffact(i3)
    addr3 : ptr?⟨int⟩                         := addr2+1
    q15 : pf(addr3=addr2+1)                   := pffact(addr3)
    q13 : pf(0≤i3)                            := pfand(q4, q12)
    q16 : pf(addr3=aBase+i3)                  := pfand(q8, q12, q15)
                                              goto LOOP
EXIT :
                                              ...
```

Figure 18. IR after strength reduction of element address

```
    uB : int                                  := len(a)
    q3 : pf(uB=len(a))                        := pffact(uB)
    aBase : ptr?⟨int⟩                         := base(a)
    q7 : pf(aBase=a@0)                        := pffact(aBase)
    addr1 : ptr?⟨int⟩                         := aBase
    q14 : pf(addr1=aBase)                     := pffact(addr1)
    addrUB : ptr?⟨int⟩                        := aBase+uB
    q17 : pf(addrUB=aBase+uB)                 := pffact(addrUB)
LOOP :
    addr2 : ptr?⟨int⟩                         := φ(addr1, addr3)
    q4 : pf(aBase≤addr2)                      := φ(q14, q13)
    [q1 : pf(addr2<addrUB), q2 : ...] :=      if addrUB≤addr2
                                                goto EXIT
    q6 : pf(addr2<aBase+len(a))               := pfand(q3, q1, q17)
    q10 : pf(a@0≤addr2<a@len(a))              := pfand(q4, q6, q7)
    val : int                                 := ld(addr2) [q10]
    ... : ...                                 := val
    addr3 : ptr?⟨int⟩                         := addr2+1
    q15 : pf(addr3=addr2+1)                   := pffact(addr3)
    q13 : pf(aBase≤addr3)                     := pfand(q4, q15)
                                              goto LOOP
EXIT :
                                              ...
```

Figure 19. IR after linear function test replacement

induction on that fact. Again, we leverage SSA to establish the new proof. In this case, $q_8 : \texttt{pf}_{(addr_2=aBase+i_2)}$ is defined by the phi instruction that merges proof variables $q_{14} : \texttt{pf}_{(addr_1=aBase+i_1)}$ and $q_{16} : \texttt{pf}_{(addr_3=aBase+i_3)}$.

Finally, we illustrate linear function test replacement (LFTR) [9] in Figure 19. [1] Classical LFTR replaces the test $uB \leq i_2$ in the branch by a new test $addrUB \leq addr_2$. If our program contained no proof variables, this would allow the otherwise unused base variable i to be removed from the loop. We augment the usual LFTR procedure, which rewrites occurrences of the base induction variable i_2 in loop exit tests (and exits) in terms of the derived induction variable $addr_2$, to also rewrite occurrences of i_2 in the *types* of proof variables. Finally, to eliminate the original induction variable altogether, the compiler must replace the inductive proofs on the original variable (expressed through ϕ instructions) with proofs in terms of the derived induction variable. In this case, the compiler must replace the proof that $0 \leq i_2$ (established by q_{11} and q_{12}) with one that proves $aBase \leq addr_2$ (established by q_{14} and q_{15}). After the replacement, the loop induction variable i and any proof variables that depend upon it are no longer live in the loop, so all definitions of the variable can be removed. The compiler must remove the proof variables whose types reduce to tautologies and apply further CSE to yield Figure 19.

5. Extensions

Our core language can easily be extended to handle other interesting aspects of Java and CLI. In this section we describe several of these extensions.

Firstly, we can handle object-model lowering through the use of our singleton types. Consider an invoke virtual operation. It is typically lowered into three operations: load the virtual dispatch table (vtable), load the method pointer from the vtable, call the method pointer passing the object as an additional argument. In our system, these operations would look like this:

$$
\begin{array}{l}
x : SomeClass := \cdots \\
t_1 : \texttt{vtable}(x) := \texttt{vtable}(x) \\
t_2 : (\texttt{S}(x), \texttt{int}) \rightarrow \texttt{int} := \texttt{method}(foo : (\texttt{int}) \rightarrow \texttt{int}, t_1) \\
t_3 : \texttt{int} := \texttt{call}(t_2)(x, 10)
\end{array}
$$

Here the method foo (taking an integer and returning an integer) is being invoked on variable x. In the lowered code, variable t_1 gets the dependent type $\texttt{vtable}(x)$ meaning that it contains the vtable from the object currently in x. Variable t_2 gets the loaded method pointer. From the type $\texttt{vtable}(x)$, the typing rules can determine a precise function type for this method pointer, namely $(\texttt{S}(x), \texttt{int}) \rightarrow \texttt{int}$, where the first argument must be x. The actual call is the last operation, and here we pass x as an explicit argument. Since x has type $\texttt{S}(x)$, this operation type checks.

By using singleton types based on term variables, we achieve a relatively simple type system and still avoid the well known typing problems with the explicit "this" argument (see [12] and references). The existing solutions to this typing problem have much more complicated type systems, with one exception. Chen and Tarditi [7] have a similarly simple type system for a lowered IR for class-based object-oriented languages. Like our system, theirs also has class names as types, and keeps around information about the class hierarchy, fields, and methods. They also have existentials with subclass bounds (type variables can be bounded above by a class, and range over any subclass of that class). They use these existentials to express the unknown runtime type of any given object, and thus the type of the explicit "this" argument. They also have a class representation function that maps class names to a record type for objects in the class, and they have coercions to convert between the two. These ideas could be adapted to our system instead of our vtable types, and our vtable types could be adapted to their type system. In summary, both systems are simpler than existing, more foundational, object encodings. Theirs has type variables and bounded existentials, ours has singleton types based on term variables.

Java and CLI also allow null as a value in any class type, and at runtime this null value must be checked and an exception thrown before any invocation or field access on an object. We can use our proof variable technique to track and ensure that these null checks are done. We simply add a null constant to the fact expression language. We can add an operation like $p : \texttt{pf}_{(x \neq \texttt{null})} := \texttt{chknull}(x)$ to check that x is not null. If x is null then it throws an exception, if not then it assigns a proof of $x \neq \texttt{null}$ to p. Similarly to array-bounds check elimination, we can eliminate redundant null checks.

To handle exceptions we simply add explicit control flow for them. Each potentially exception throwing operation will end a basic block and there will be edges coming out of the block corresponding to exceptions that go to blocks corresponding to the exception handlers. An important point is that exceptions typically occur before the assignment of the potentially exception throwing operation, so like the conditional branches of our core language, we must treat the definition point as occuring on the fall-through edge rather than at the end of the basic block. So in both $x : \tau := \texttt{chknull}(y)$ and $x : \tau := \texttt{call}(y)(\overline{y})$, the variable x is assigned on the fall-through edge.

We can easily deal with stores to pointers by adding a store operation of the form $\texttt{st}(x, y)\ [p]$ where x holds the pointer, y the value to store, and p a proof that x is valid. The type rule for this operation is:

$$
\frac{\Gamma \vdash \Gamma(x) \leq \texttt{ptr}_?\langle\tau\rangle \quad \Gamma \vdash \Gamma(y) \leq \tau \quad \Gamma \vdash \Gamma(p) \leq \texttt{pf}_{(z@0 \leq x \wedge x < z@\texttt{len}(z))}}{\Gamma \vdash_P \texttt{st}(x, y)\ [p]}
$$

Modifying our formalisation and type soundness proof to accomodate stores would be straightforward.

Java and CLI have mutable covariant arrays, and thus require array-store checks at runtime. In particular, when storing into an array, the runtime must check that the object being stored is compatible with the runtime element type of the array (which could be a subtype of the static element type). In our implementation we use types of the form $\texttt{elem}(x)$ to stand for the runtime element type of array x. The load base operation on x actually returns something of type $\texttt{ptr}_?\langle\texttt{elem}(x)\rangle$. The array-store check produces a proof value that can be used to prove that some other variable has type $\texttt{elem}(x)$ and we have a coercion to use the proof value to change the variable's type. The end of a lowered array store would look something like this:

$$
\begin{array}{l}
x : \texttt{array}(C) := \cdots \\
y : C := \cdots \\
\cdots \\
p_1 : \texttt{pf}_{(x \neq \texttt{null} \wedge x@0 \leq t \wedge t < x@\texttt{len}(x))} := \cdots \\
p_2 : \texttt{pf}_{(y:\texttt{elem}(x))} := \texttt{chkst}(x, y) \\
\texttt{st}(t, \texttt{retype}(y, p_2))\ [p_1]
\end{array}
$$

One technicality is worth noting. In order to avoid circularities between the type system and the fact language, and to avoid making the fact language's decision procedure mutually dependent upon the subtype checker, we restrict the types that can appear in a fact of the form $x : \tau$ to those that do not mention proof types.

Downcasts are similar to store checks, and we can treat them in a similar way. A $\texttt{chkcast}(x : C)$ operation checks that x is in type C and returns a proof of this fact, otherwise it throws an exception. The actual subtype checks performed at runtime in our implementa-

[1] Note that the code resulting from LFTR is not typable in our core language, since we do not allow conditional branches on pointers. Extending the language to handle this is straightforward, but requires a total ordering on pointer values which essentially requires moving to a heap-based semantics. Note though that the fact language does permit *reasoning* about pointer comparison, as used in the previous examples.

tion are generally done by the virtual machine itself, and the virtual machine is not type checked by the type system of our JIT. However, we do partially inline this operation to include some common fast cases, and to expose some parts to redundant elimination and CSE. For example, if a object is null then it is in any reference type and can be stored into any reference array or downcast to any reference type. Another example is comparing the vtable of an object against the vtable of a specific class, if these are equal then that object is in that class. Such comparisons produce facts in our system of the form x=`null` or `vtable`(x)=`vtable`(C). We can simply add axioms to our fact language like $\vdash$ x=`null` $\implies$ $x : C$ or $\vdash$ `vtable`(x)=`vtable`(C) $\implies$ $x : C$.

6. Implementation Status

The current implementation of the STARJIT compiler generates and maintains proof variables throughout its compilation process to enable safe implementation of certain optimizations in the presence of check elimination (to be described in a forthcoming paper). For their initially designed role in optimizations, proof variables did not require proof types: optimizations do not need to know the reason an optimization was safe, but only its safety dependences. As such, the current STARJIT representation is similar to that described in Section 2 with some of the extensions in Section 5.

STARJIT implements all of the optimizations discussed in this paper as well as more described in [1]. We modified each optimization, if necessary, to correctly handle proof variables. Array bounds check elimination and operator strength reduction required the most significant modification, as described in Section 4. For partial inlining of virtual machine type checking functions, as described in Section 5, we updated the definition of proof variables to established that a variable has the checked type. We also modified method inlining to properly establish the type of inlined methods. For each parameter of a method, we added a proof variable that established that it had the correct type. When a method is compiled independently, that proof variable is trivially defined at the method entry (as parameter types to a method are guaranteed by the runtime environment). When the method is inlined, the corresponding proof variables must be defined by the calling method instead. As method call operations require proof variables for each parameter in our system, this information is readily available. Most optimizations, however, did not require significant changes for the reasons outlined in this paper.

An early version of a type verifier which inferred proof types itself was implemented. This implementation was particularly helpful in finding bugs within STARJIT, but was insufficient for complete verification of optimized code. In particular, the inference algorithm was insufficient for some more complicated optimization situations, such as the LFTR example (without proof type information) in Section 4. We are confident that extending the compiler to use precise proof types for proof variables will be straightforward, using the framework developed in this paper.

7. Related Work

As far as we are aware, SafeTSA [25, 2] is the only other example of a type-safe SSA representation in the literature. The motivation of their work is rather different than ours. SafeTSA was designed as an alternative to Java bytecode, whereas our representation is designed to be a low-level intermediate language for a bytecode compiler. SafeTSA can represent certain optimizations, such as CSE and limited check elimination, that Java bytecode does not. However, in our classification in Section 2, SafeTSA is a refinement-style representation and, thus, cannot represent the effect of many of the low-level optimizations we discuss here. For example, it cannot represent the safety of check elimination based upon a previous branch or the construction of an unsafe memory address as illustrated in Figure 7. On the other hand, we do not support their notion of referential security: the property that a program must be safe by construction.

While most of the work on certified code focuses on the final machine code representation, there has been previous work on intermediate representations that allow verification of the memory safety of highly optimized machine level code. One of the major differences between the various approaches lies in the degree to which safety information is made explicit.

On the side of less explicit information are the SpecialJ compiler [8] and DTAL [27]. Both approaches record loop invariants, but not explicit safety dependences. This makes verification harder (all available invariants must be considered by the decision procedure), interferes with more optimizations (such as loop peeling) than our approach, and makes removing dead invariants much more difficult (because invariants never have explicit uses).

At the other end of the spectrum, there are other systems that not only represent dependences explicitly as we do, but also record exactly why the dependences imply safety for each instruction, using proofs, instead of relying on a decision procedure during checking, as in our system. The LTT system of Crary and Vanderwaart [10] and the TSCB system of Shao et al. [23], developed independently, both take this approach, albeit in the setting of a functional or mostly-functional language. Both systems are designed around the idea of incorporating a logic into a type theory, in order to combine the benefits of proof-carrying code [20] with the convenience of a type system. LTT and TSCB adopt the linear logical framework LLF and the Calculus of Inductive Constructions, respectively, as their proof languages. Incorporating a proof system also gives them more flexibility, as they can express a variety of properties within a single framework.

The lack of explicit proofs in the representation forces us to use a decision procedure during typechecking. This limits us to decidable properties, and may be less suited for certified code applications where the added complexity of a decision procedure in the verifier may be undesirable.

On the other hand, a system such as ours is much more suited to use in the internals of an optimizing compiler. For the limited use that we need proofs for—to verify the correctness of checks which are eliminated by a real optimizing compiler—we can get away with a vastly simpler system, one that imposes much less of a burden on the compiler than more syntactically heavy systems. Moreover, for applications of certified code, we believe that it should be possible to take optimized intermediate code in the style presented here and translate it, as part of code generation, to a more explicit form in the style of LTT or TSCB, thereby reaping the benefits of both approaches, perhaps by following the Special J model of using a proof generating theorem prover. However, this remains future work.

Finally, our proof variables are also similar to the Jalapeño Java system's condition registers as described in [6, 14]. Both are mechanisms to represent control-flow information as abstract value dependences. Their usage, however, is more limited. Condition registers are not used to express general safety information or to support verification of code. Instead, they are used by the compiler to model control flow between a check operation and all (rather than just potentially unsafe) instructions that follow it. Jalapeño uses condition registers to collapse control flow due to exceptions into a single extended block and, in that block, to prevent instruction reordering that would violate control flow dependences.

8. Conclusions

This paper has shown a typed low-level program representation that preserves memory safety dependences in highly-optimizing

type-preserving compilers. Our representation encodes safety dependences as first-class term-level proof variables that capture the essential memory-safety dependences in the program without artificially constraining optimizations—previous approaches that piggyback safety dependence on top of value dependence inhibit optimization opportunities. Our representation encodes proofs of memory safety as dependent types associated with proof variables. Experience implementing this representation in the STARJIT compiler has demonstrated that a highly-optimizing Java JIT compiler can easily generate and maintain this representation in the presence of aggressive SSA-based optimizations such as bounds check elimination, value numbering, strength reduction, linear function test replacement, and others. Using explicit proof values and proof types, modern optimizing compilers for type-safe languages can now generate provably safe yet low-level intermediate representations without constraining optimizations.

References

[1] ADL-TABATABAI, A.-R., BHARADWAJ, J., CHEN, D.-Y., GHULOUM, A., MENON, V. S., MURPHY, B. R., SERRANO, M., AND SHPEISMAN, T. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal 7*, 1 (February 2003).

[2] AMME, W., DALTON, N., VON RONNE, J., AND FRANZ, M. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (Snowbird, UT, USA, 2001), pp. 137–147.

[3] BILARDI, G., AND PINGALI, K. Algorithms for computing the static single assignment form. *J. ACM 50*, 3 (2003), 375–425.

[4] BODÍK, R., GUPTA, R., AND SARKAR, V. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (Vancouver, British Columbia, Canada, 2000), pp. 321–333.

[5] BRIGGS, P., COOPER, K. D., AND SIMPSON, L. T. Value numbering. *Software—Practice and Experience 27*, 6 (June 1996), 701–724.

[6] CHAMBERS, C., PECHTCHANSKI, I., SARKAR, V., SERRANO, M. J., AND SRINIVASAN, H. Dependence analysis for Java. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing* (1999), vol. 1863 of *Lecture Notes in Computer Science*, pp. 35–52.

[7] CHEN, J., AND TARDITI, D. A simple typed intermediate language for object-oriented languages. In *Proceedings of the 32nd Annual ACM Symposium on Principles of Programming Languages* (Long Beach, CA, USA, Jan. 2005), ACM Press, pp. 38–49.

[8] COLBY, C., LEE, P., NECULA, G. C., BLAU, F., PLESKO, M., AND CLINE, K. A certifying compiler for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), ACM Press, pp. 95–107.

[9] COOPER, K. D., SIMPSON, L. T., AND VICK, C. A. Operator strength reduction. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23*, 5 (September 2001), 603–625.

[10] CRARY, K., AND VANDERWAART, J. An expressive, scalable type theory for certified code. In *ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, 2002), pp. 191–205.

[11] CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, K. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages* (Austin, TX, Jan. 1989).

[12] GLEW, N. An efficient class and object encoding. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages* (Minneapolis, MN, USA, Oct. 2000), ACM Press, pp. 311–324.

[13] GROSSMAN, D., AND MORRISETT, J. G. Scalable certification for typed assembly language. In *TIC '00: Selected papers from the Third International Workshop on Types in Compilation* (London, UK, 2001), Springer-Verlag, pp. 117–146.

[14] GUPTA, M., CHOI, J.-D., AND HIND, M. Optimizing Java programs in the presence of exceptions. In *Proceedings of the 14th European Conference on Object-Oriented Programming - ECOOP '00 (Lecture Notes in Computer Science, Vol. 1850)* (June 2000), Springer-Verlag, pp. 422–446.

[15] IGARASHI, A., PIERCE, B., AND WADLER, P. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23*, 3 (May 2001), 396–560. First appeared in OOPSLA, 1999.

[16] KNOOP, J., RÜTHING, O., AND STEFFEN, B. Lazy code motion. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation* (San Francisco, CA, June 1992).

[17] MENON, V., GLEW, N., MURPHY, B., MCCREIGHT, A., SHPEISMAN, T., ADL-TABATABAI, A.-R., AND PETERSEN, L. A verifiable SSA program representation for aggressive compiler optimization. Tech. Rep. YALEU/DCS/TR-1338, Department of Computer Science, Yale University, 2005.

[18] MORRISETT, G., CRARY, K., GLEW, N., GROSSMAN, D., SAMUELS, R., SMITH, F., WALKER, D., WEIRICH, S., AND ZDANCEWIC, S. TALx86: A realistic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software* (Atlanta, Georgia, 1999), pp. 25–35. Published as INRIA Technical Report 0288, March, 1999.

[19] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS) 21*, 3 (May 1999), 528—569.

[20] NECULA, G. Proof-carrying code. In *POPL1997* (New York, New York, January 1997), ACM Press, pp. 106–119.

[21] NECULA, G. C., AND LEE, P. The design and implementation of a certifying compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation* (New York, NY, USA, 1998), ACM Press, pp. 333–344.

[22] PUGH, W. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91* (Albuquerque, NM, Nov. 1991).

[23] SHAO, Z., SAHA, B., TRIFONOV, V., AND PAPASPYROU, N. A type system for certified binaries. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages* (January 2002), ACM Press, pp. 216–232.

[24] VANDERWAART, J. C., DREYER, D. R., PETERSEN, L., CRARY, K., AND HARPER, R. Typed compilation of recursive datatypes. In *Proceedings of the TLDI 2003: ACM SIGPLAN International Workshop on Types in Language Design and Implementation* (New Orleans, LA, January 2003), pp. 98–108.

[25] VON RONNE, J., FRANZ, M., DALTON, N., AND AMME, W. Compile time elimination of null- and bounds-checks. In *3rd Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)* (December 2000).

[26] WALKER, D., CRARY, K., AND MORISETT, G. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems (TOPLAS) 22*, 4 (July 2000), 701–771.

[27] XI, H., AND HARPER, R. Dependently typed assembly language. In *International Conference on Functional Programming* (September 2001), pp. 169–180.