University of Texas at San Antonio, Department of Computer Science, Technical Report 2006-004

SafeTSA:
An Inherently Type-Safe SSA-based Code Format

JEFFERY VON RONNE
The University of Texas at San Antonio
and
WOLFRAM AMME
Friedrich-Schiller-Universität Jena
and
MICHAEL FRANZ
University of California, Irvine

Conventional type safe virtual machines, such as the Java Virtual Machine, utilize a stack-oriented bytecode language and require verification prior to execution. We present SafeTSA, a compiler-friendly alternative to stack-oriented bytecode based on static single assignment form. SafeTSA's special features (type separation, dominator-based scoping, and high-level control structures) facilitate producer-side, ahead-of-time optimization and an inherently-safe, space-efficient encoding.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Optimization*; *Run-time enviornments*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Portability*; D.3.2 [**Programming Languages**]: Language Classifications—*Java*; *Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*; E.4 [**Coding and Information Theory**]: —*Data compaction and compression*

General Terms: Design, Languages, Experimentation, Performance, Security

Additional Key Words and Phrases: Program representations, mobile code, SafeTSA, static single assignment form, type safety

1. INTRODUCTION

The Java Virtual Machine provides a type-safe, machine-independent platform for mobile code. The security features of the Java Virtual Machine allow untrusted code, which may be downloaded from an untrusted internet site, to be executed on the users local machine without endangering the confidentiality or integrity of the other programs and data on that machine. This is accomplished by transmitting the programs in Java classfiles containing Java bytecode. The bytecode is usually produced by compiling (using javac or a similar Java compiler) a Java source program into Java classfiles at a "code producer." These Java classfiles can then be transmitted across the internet or on some media to the "code consumer," where the users execution environment, the Java Virtual Machine, will handle executing the bytecode program (often by translating it into native code) and enforcing the system's safety and security policies. Enforcement of type and memory safety is the foundation for Java's higher-level security policies. There are three important aspects to Java type and memory safety: the load-time "verification" (type check-

ing) of the bytecode in classfiles, link-time checks of class hierarchy consistency, and runtime null- and bounds-checks performed by the execution environment. If any of these are missing, Java's type safety along with the integrity of the entire Java Virtual Machine is undermined.

In general, the code producer only processes the program once and does not need to be trusted. On the other hand, many programs (or compilation units thereof) will be executed many times by many different code consumers. A consequence of this is that any work that can be safely shifted from the code consumer to the code producer, should be done by the code producer.

These considerations informed the development of SafeTSA, an alternative code format for the Java platform. SafeTSA's unique design, based on static single assignment form provides several benefits. It safely shifts optimization effort to the code producer. It retains high-level control structure information, which would otherwise need to be reinferred by the code consumer. It has a compact "inherently safe" binary encoding, which is about as small as compressed Java classfiles, and in which there is no way to represent programs that violate important aspects of type and memory safety.

In the rest of this article, we will

—discuss static single assignment form and its advantages compared to stack-oriented bytecodes,

—describe the overall structure of the SafeTSA file format,

—present the SafeTSA instruction set and explain some of its unique, optimization-friendly features,

—describe how SafeTSA retains the source program's high level control structures,

—explain how SafeTSA's inherently safe encoding works, and

—present some experimental results concerning SafeTSA's file size and decoding performance on the Java Grande Benchmarks.

2. AN SSA BASED REPRESENTATION

The SafeTSA representation was designed as an alternative to Java class files containing the Java Virtual Machine's bytecode. The key innovations of SafeTSA are the use of the static single assignment form in a code transportation format and a language design that enforces safety properties by construction allowing an inherently safe encoding to replace the verification process used by conventional mobile code formats.

Most virtual machine "bytecodes," such as the Java bytecode are, however, designed around a different principle. They use an implicit operand stack to communicate the result of one instruction to its use in the next; this stack-oriented virtual machine design easy to compile to, easy to interpret, and usually space efficient, but it requires a verification phase based on iterative dataflow analysis, and is difficult to optimize because code motion or removal tends to disrupt the stack.

Static single assignment form (SSA) [Cytron et al. 1991], however, is an intermediate representation designed for optimization and is now found in many state-of-the-art optimizing compilers (e.g., gcc [GCC 2005]). The same properties that

```
class A {
        double field1;
        double field2;

        static int foo (A a, int i) {
                double x = a.field1;
                if (i > 0)
                        x = 2.0 * x;
                else
                        x = a.field2 * x;
                return x;
        }
}
```

Fig. 1. source for class A

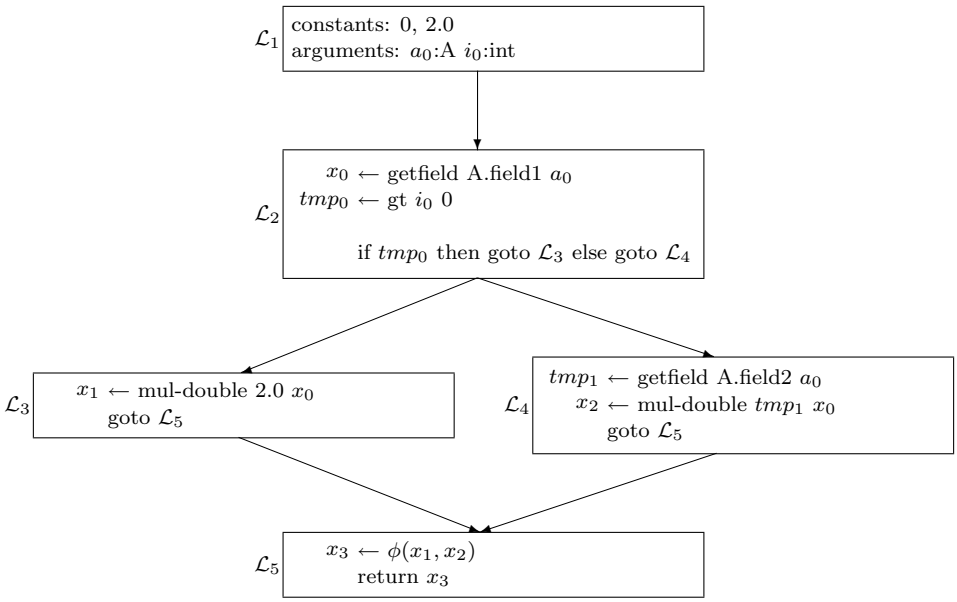

Fig. 2. A.foo in SSA form

have made SSA standard for optimizing compilers, could also be leveraged to produce an type-safe machine-independent on-the-wire format that is easy to verify and optimization friendly. This observation lead to the development of SafeTSA, combines SSA with several other unique language features (including an inherently safe binary encoding that leverages the properties of SSA and type coercion to guarantee that SafeTSA programs are type-correct and otherwise valid, a type-system feature that facilitates ahead-of-time optimization with safe null- and bounds-check eliminations, and the use of high-level control structures that constrain the control flow graph and convey high-level information to subsequent optimization phases).

The distinguishing property of SSA is that the result of each instruction is given a unique name (i.e., is only assigned to at one static location). Figure 1 shows a Java program that we will use as a running example throughout this article, and Figure 2 shows a translation of this program into SSA. Since each instruction's result must be given a unique name, there will be several SSA variable names for each source variable name. By convention each of these SSA variable are named by using the corresponding original program variable plus a unique subscript, and thus, each name/subscript combination appears on the left-hand side of only a single instruction. Likewise, all instruction input operand use the full SSA variable name, specifying both the name and the subscript. With this naming scheme, there is a one-to-one mapping from SSA variable names to instructions, and this property makes it is easy to identify the definition of a variable used by a particular instruction. A difficulty arises, however, when one encounters a place in the original program that may use more than one definition of a variable depending on how the program execution reached that variable. For example, the return statement in Figure 1 may sometimes use the value of x assigned on the then branch and sometimes use the value of x assigned on the else branch depending on which branch was taken. In those cases, SSA uses special ϕ-functions that model the selection of one of its input parameters based on how the control flow reached the ϕ-function.

SSA is often used as an intermediate representation for optimizing compilers, because the single assignment property facilitates more efficient algorithms many scalar analyses and optimizations, such as constant propagation [Wegman and Zadeck 1991] and redundancy elimination [Rosen et al. 1988] In effect the ϕ-functions flatten out the programs dataflow and allows context-insensitive algorithms to perform optimizations that would otherwise require iterative context-sensitive analyses.

3. OVERVIEW OF THE SAFETSA FILE FORMAT

SafeTSA is a complete file format and can be used as a drop-in replacement for Java class files. Like Java classfiles, it operates at the class granularity, and each SafeTSA file consists of:

—a list of constants,

—a list of imported types, their subclass, and interface relationships, their methods, and their fields,

—a list of the class being encoded's name, superclass and super interfaces, method signatures, and fields,

—field initializers, and

—constructor and method bodies

Each of these major components is made up of a sequence of symbols, each of which[1] is encoded using a binary prefix code. By constraining the items representable in the binary prefix codes, it is possible to guarantee the type safety and other important

[1]except for string constants, which are represented in UTF-8

properties of any encoded SafeTSA program, such that it is impossible to represent programs violating these properties.

A SafeTSA file begins with list of constants which may be strings or any of the primitive types. These constants are immediately followed by a list of classes, arrays, interfaces, methods, and fields that are referenced in the definitions that follows. The rest of the SafeTSA file will use these items in a manner consistent with these declarations, but it is up to the executing virtual machine to verify that these declarations correspond with the classes and class hierarchy loaded in virtual machine at runtime. Each field declaration specifies the field's name, type, and owning class and whether the field is static, whereas each method declaration indicates the method's owning class, its name, its access modifiers, its parameter types, and its return type.

This is followed by the definition of a class, which includes all of the methods being defined/overridden by that class. Each SafeTSA method definition consists of its prototype, a listing of formal parameters, high-level control structures (e.g., for loops, switch statements, and try/catch statements) and SSA instructions (each of which uses zero or more SSA Variables as its input and may define one SSA Variable as its output), and SSA phi-functions.

Each SafeTSA method is organized into a tree structure, called the control structure tree (CST), whose relationships express the nesting and ordering of the original source program's high-level control structures (e.g., if and while statements). Each node type may be labelled with different data and puts different constraints on its children.

Two of the most fundamental CST Node types are the Blockgroup and the Joinnode nodes. Each of the CST's Blockgroup nodes contains a single-entry superblock of SSA-instructions which may have many exception-caused exits but only one non-exception exit. Each CST Joinnode contains the SSA phi-functions belonging to a location in the program's control-flow graph where two or more execution paths converge; this location is determined by the Joinnode's context (its position in relationship to its parent and sibling nodes) within the control structure tree.

3.1 Implicit Naming

SafeTSA is designed so that its local variables are not given explicit names. Because SafeTSA is based on Static Single Assignment Form (SSA), each variable is assigned to at only one location. Thus it is sufficient, as shown in Figure 3 to just simply enumerate each instruction (or catch-clause, method-argument, or constant declaration) that defines an SSA variable and use that enumeration as the left-hand-side of that assignment. This program is semantically the same as the one shown in 2 even though it no longer contains information about which original program variables were being defined by each instruction; SSA allows each of the new names to be treated as an independent variable.

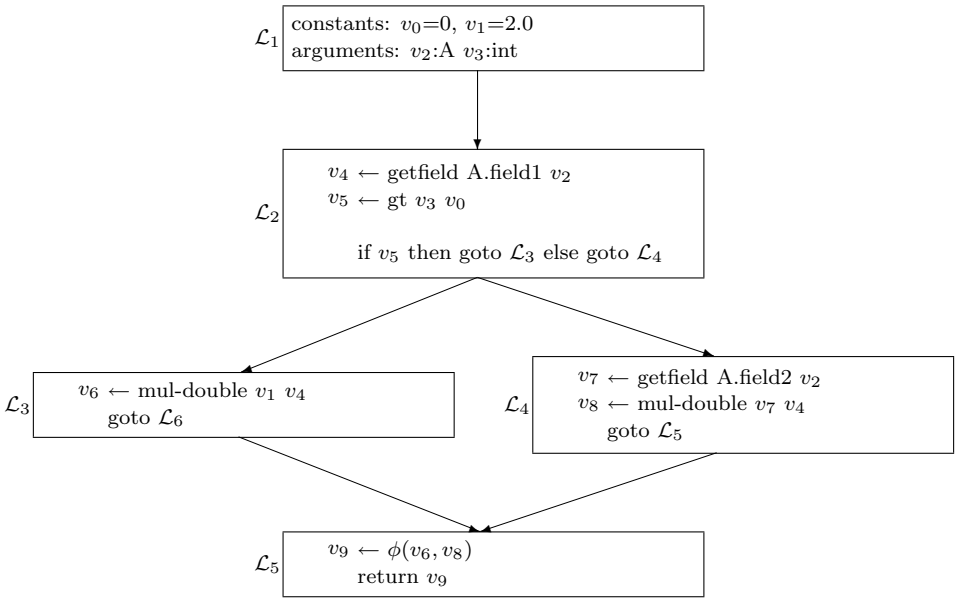

Fig. 3. A.foo in SSA with Implicit Naming

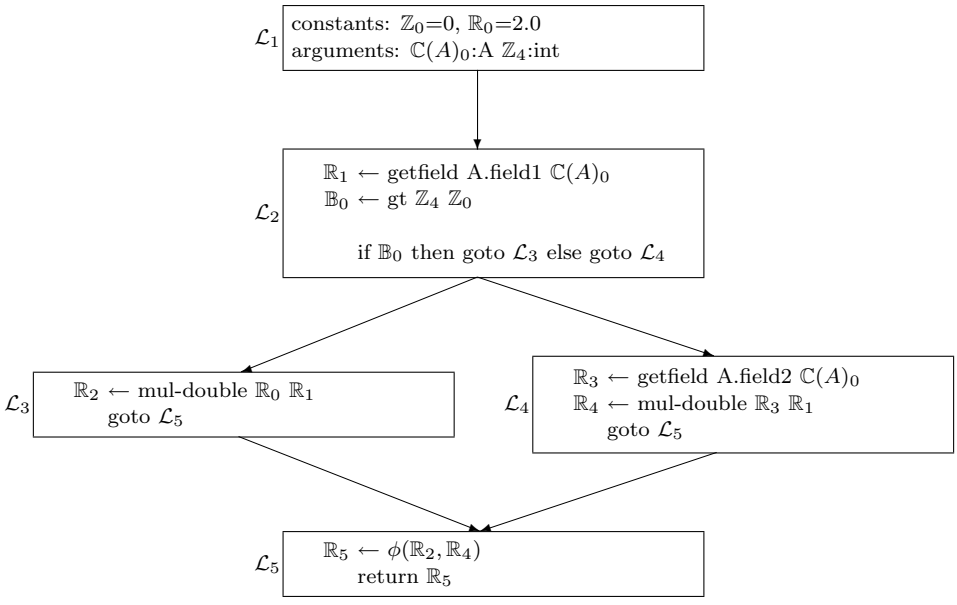

Fig. 4. A.foo in Type-Separated SSA form

3.2 Type Separation

SafeTSA makes a distinction between the instruction parameters[2] (which are static symbols) and operands (which refer to typed variables). Each of SafeTSA's instruction imposes specific typing constraints on its operands and its results. These typing constraints are a function of on the instruction operation and the the symbolic parameterization of the instruction (e.g., a `getField` instruction will be parameterized by the field name and that field's declaration will determine the type of the instructions result value). The typing constraints are independent of any SSA variable operands, and always impose an "exact" type on each operand and the instruction's result. There is no subtyping at the syntactic, SSA variable level, so unlike the Java source language, in SafeTSA *there is no implicit coercion between types.* If a `setFieldStatic` is passed an integer field, then the value operand can only be an integer (not a short or a long). If a `getFieldInstance` is parameterized by the *field*, `AbstractList.modCount`, then the object operand needs to have the type "reference to an `AbstractList`" and not "reference to a `LinkedList`." If a program access the `modCount` field of a LinkedList, then the compiler either needs to add a field declaration for `LinkedList.modCount` (distinct from `AbstractList.modCount`) or needs to insert an explicit cast instruction to create a new SSA variable whose type is "reference to an AbstractList" from the old variable whose type is "reference to an LinkedList."[3] Thus, in SafeTSA, the namespaces for variables of different types are completely separate.

Figure 4 show the program from 1 rendered in a type-separated SSA form. Each variable is referred to by its type (here we use: $\mathbb{Z}$ for integer, $\mathbb{R}$ for doubles, $\mathbb{B}$ for boolean, $\mathbb{C}(A)$ for the type of references to Objects of class A) plus a subscript. For each type, the subscripts are assigned sequentially, so that each type, effectively, has its own name space. This type separation ensures that, whenever any instruction requires an operand, that operand's type is already statically determined and only values of the correct type can be used. This simplifies type checking and makes it possible to create binary encodings that enforce the type discipline.

This type separation is, however, only at the syntactic level. There is still subclassing of objects on the heap, so that one can still have a variable whose static type is "reference to an `AbstractList`" but which—at runtime—refers to an object that is an instance of the `LinkedList` class. When necessary the compiler inserts explicit coercion instructions to translate from one static type to another (c.f., [Breazu-Tannen et al. 1991]).

3.3 Dominator Scoping

While the implicit numbering ensures that each local variable is only assigned to at a single location, and type separation ensures that the types of variable definitions match the types of variable uses. So far, there is nothing that requires a variable to be available when (i.e., always defined before) it is used. This property will be

[2]These instruction parameters can be thought of as being like C++ template arguments, and are unrelated to both ϕ-function parameters and the formal/actual parameters of methods.

[3]Normally, the JIT compiler will be able to recognize that the contents of these two variables are the same, and they can, therefore, be coalesced into the a single register/memory location. Thus there is not a runtime penalty for instantiating these extra variables.

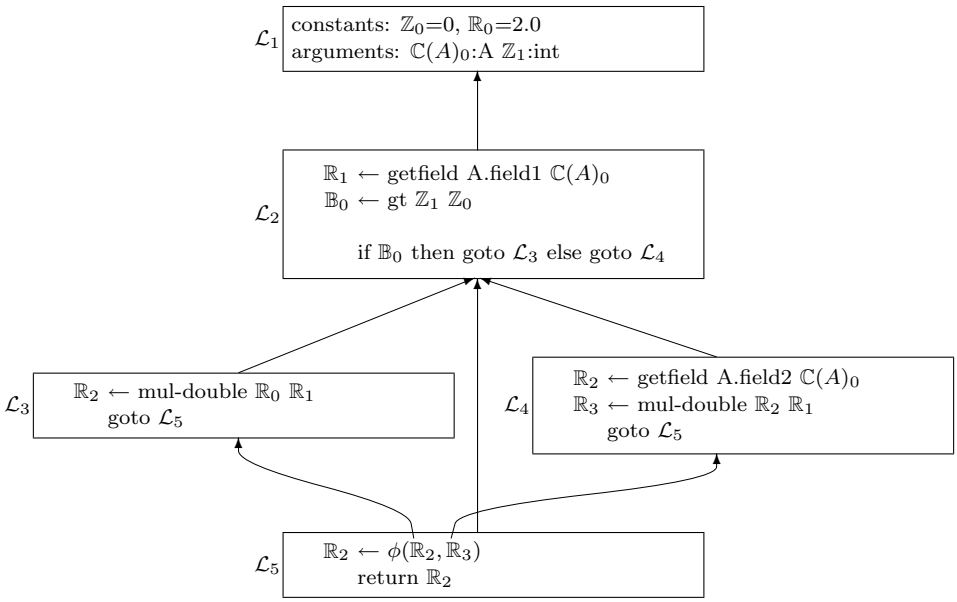

Fig. 5. A.foo in SSA form with Dominator Scoping

satisfied if every use of a variable is dominated by its definition. (A variable definition *dominates* a variable use in a function's intra-procedural control flow graph (CFG) if and only if every path through the CFG from the start of the method to the variable use includes the variable definition.) The domination of every use of a variable by that variable's definition is also a necessary condition for a program to be in correct SSA form. This is because, in SSA form, there is only a single definition site for each variable. If this definition does not dominate the use, then there must be a path from the beginning of the method to that use on which the defining site is not executed. The variable would then be undefined if execution were to proceed along that path; thus, the program is invalid. Fortunately, determining the dominator relationship between nodes in a control flow graph is a well known problem for which efficient solutions are readily available (e.g., [Lengauer and Tarjan 1979]).

Figure 5 shows our example program in a dominator-scoped type-separated SSA form. As with the program in Figure 4 the variables are named by a type plus a subscript, but in Figure 5 names are restricted in scope according to the "is dominated by" relationship (as denoted by the upward pointing arrows in this figure), such that each instructions can only see the variable names defined above it in the dominator tree. In this way, each instruction can only refer to variables, which have been already defined. The variables of each type are numbered so that each variable has the lowest subscript which is not the same as any SSA variables that dominate it. This ensures that at any instruction, the dominating variables of type τ will be named $\tau_0 \ldots \tau_n$. Distinct SSA variables in non-overlapping scopes may have the same name, but there will always be exactly one in-scope definition of each of the variables $\tau_0 \ldots \tau_n$.

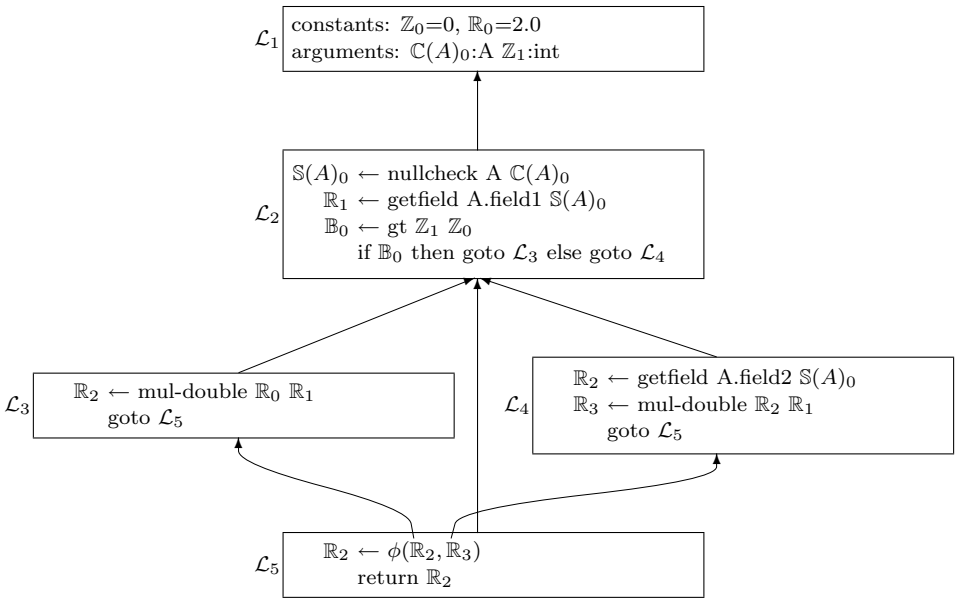

Fig. 6. A.foo in SSA form with Type-Enforced Null-Checking

For a concrete example, consider again Figure 5, and note that there are three different definitions of $\mathbb{R}_2$ in blocks $\mathcal{L}_3$, $\mathcal{L}_4$, and $\mathcal{L}_5$, but these are all in different scopes, since none of those blocks dominates any of the others of those blocks, there is no ambiguity. The return statement at the end of $\mathcal{L}_5$ refers to the $\mathbb{R}_2$ defined in $\mathcal{L}_5$ because the ones in $\mathcal{L}_3$ and $\mathcal{L}_4$ do not dominate $\mathcal{L}_5$ and are therefore out of scope. Likewise, the $\mathbb{R}_2$ referred to in the multiply instruction of $\mathcal{L}_4$ refers to the $\mathbb{R}_2$ in $\mathcal{L}_4$, because the others do not dominate that instruction. One apparent anomaly is the ϕ-function at that beginning of $\mathcal{L}_5$: its two parameters $\mathbb{R}_2$ and $\mathbb{R}_3$ refers to ones in defined in blocks $\mathcal{L}_3$ and $\mathcal{L}_4$ respectively, neither of which dominates $\mathcal{L}_5$. This anomaly is resolved, by considering ϕ-function parameters to be interpreted as if they were at the end of the block from which the control flow came. In this case, the first parameter is used when the control flow comes from $\mathcal{L}_3$, so the $\mathbb{R}_2$ is the one in scope at the end of $\mathcal{L}_3$; similarly, the second parameter is used when the control flow comes from $\mathcal{L}_4$, so the $\mathbb{R}_3$ in $\mathcal{L}_4$ is the one that is intended.

3.4 "Safe" Null-Checked Reference Types

Type-separation and dominator scoping are sufficient to guarantee that all programs are correctly typed, but Java's type safety also depends on the correct application of null- and bounds-checks. The Java programming language and Java bytecode requires that every dereference (e.g., field accesses, method calls) of a reference variable includes an implicit null-check. If this null-check discovers that the reference variable is null, then an exception is thrown. In most programs, many of these null-checks are—in fact—unnecessary because there is no feasible path of execution through the program that will cause the exception to be thrown. If this fact is to be taken advantage of, however, it must be rediscovered by the JIT compiler, because

there is no mechanism for a programmer or a compiler to safely communicate this information through Java bytecode.

SafeTSA, on the other hand, does have a mechanism by which redundant (and therefore unnecessary) null-checks can be eliminated. Instead of having a single reference type for each Java class, SafeTSA has two: a null-checked "safe" reference type and a regular "unsafe" reference type. All dereferencing instructions require that their operand is of a "safe" reference type (which is known to not be null). Thus, these instructions do not need to include an implicit null-check. All values loaded from fields or passed as method parameters belong to "unsafe" reference types (which may be null), and an explicit `nullCheck` instruction is provided to create a new "safe" reference variable from these "unsafe" reference variable. This instruction is parameterized by the static class of the reference variable.[4] When a `nullCheck` instruction is executed, it will throw an exception if the input operand is null. Otherwise it will create a new "safe" reference variable pointing to the same object as the input operand. A `staticCast` can be used to create a "unsafe" reference variable from a "safe" one.

For a concrete example, refer to Figure 6, which shows our example program with an explicit nullcheck instruction. The type of "unsafe" references to class A denoted by $\mathbb{C}(A)$ and the type of "safe" references to class A denoted by $\mathbb{S}(A)$. Note that because the `getfield` instruction only accepts values in the $\mathbb{S}(A)$ namespace, it ensures that it only dereferences values that have already been null-checked. But it no longer requires that the nullcheck occur immediately the dereference. For example, in Figure 6, the `getfield` in $\mathcal{L}_4$ can rely on the `nullcheck` in $\mathcal{L}_1$. The dominator scoping rules, ensure that the dereference is either dominated by a `nullcheck` or a ϕ-function that joins null-checked values on all incoming execution paths.

Thus, SafeTSA safely decouples the null-check operation from the dereferencing instructions. This in turn, enables optimizations (such as common subexpression elimination) to be performed that can identify and remove redundant `nullCheck`'s. (If these optimizations reschedule instruction, however, correctness requires that the rescheduled program only throws exceptions when they would have been thrown by the original program and that Java's precise exception semantics are maintained.) It should also be noted that this mechanism only facilitates the elimination of redundant null-checks but may not allow the elimination of all provably unnecessary null-checks.

3.5 Arrays and "Safe" Bounds-Checked Element Reference Types

SafeTSA also provides a mechanism for decoupling array bounds-checks from individual get and set element instructions. An approach using "safe bounds-checked indices" analogous to "safe null-checked references" is described in [Amme et al. 2001], but because different arrays can have different lengths, this approach requires a new "safe bounds-checked index" type for each array reference, which is a restricted kind of dependent type system (c.f., [Xi and Pfenning 1998]). The current

[4]The type of a reference is determined by whether it is "safe" or "unsafe" and also by a static class. It is required that a reference variable always refer to null or an object that is an instance of that static class or one of its subclasses.

$$\frac{x : \mathsf{unsafe\text{-}ref}(\kappa)}{\texttt{nullCheck}\ \kappa\ x : \mathsf{safe\text{-}ref}(\kappa)}$$

$$\frac{\kappa : \mathsf{array}(\tau) \qquad o : \mathsf{safe\text{-}ref}(\kappa) \qquad x : \mathsf{int}}{\texttt{makeElementRef}\ \kappa\ o\ x : \mathsf{elt\text{-}ref}(\tau)}$$

$$\frac{x : \mathsf{elt\text{-}ref}(\tau)}{\texttt{getElementt}\ \tau\ x : \tau}$$

Fig. 7. Type Rules for the Array Access Instructions

version of SafeTSA avoids this complication, by dividing up the array instructions differently. Instead of instantiating a "safe index type" for each array reference, the new SafeTSA array model instantiates one safe "safe element reference" types for each element type.

A "safe element reference" type represents points—not to Java objects—but to memory cells corresponding to particular elements of arrays. Only one of these types (e.g., "safe element reference to an integer cell") is required for each array/element type.

Under this new model, a Java array element access consists of three instructions. For example, the Java statement:

```
x = a[5];
```

might be translated into the following sequence of instructions:

$t_1 \leftarrow$ `nullCheck` ([int) a_0
$t_2 \leftarrow$ `makeElementRef` ([int) t_1 5
$x_0 \leftarrow$ `getElement` (int) t_2

The type rules governing these in instructions are shown in Figure 7. The first instruction, `nullCheck`, checks that the reference a_0 is not null and produces the result t_1 with the type "safe reference to an array with elements of type int." The second instruction, `makeElementRef`, checks that the index, 5, is within the bounds of the array accessible through t_1 (throwing an `IndexOutOfBounds` exception if it is not in bounds) and results in the address of the 6th element of that array; this result is given the type "safe reference to an element of type int." The third instruction is the actual `getElement` which fetches the integer at the address specified by t_2. The type system ensures that the instructions are used in the correct order while still allowing fully-redundant bounds-checks to be eliminated.

Under the old model [Amme et al. 2001], the address computation was performed in the getElement instruction, which took two operands (i.e., a safe reference to an array and the safe index of the array), and the dependent type system was necessary to ensure that they matched. In this new model, the address computation and array bounds check both occur in the same instruction (i.e., `makeElementRef`), and only a single operand t_2 is passed into the `getElement` instruction.

4. INSTRUCTION SET

SafeTSA's instruction set includes instructions for performing arithmetic and logical operations on the primitive types and interacting with Java objects on the Java

Virtual Machine's type safe heap (including field access, method call, object instantiation, and array manipulation). In SafeTSA, we distinguish between instruction parameterization based on types or symbolic link information, and operands which refer to SSA Variables. The parameters are static and specialize the instructions at load-time. The operands, on the other hand, refer to constants or SSA variables of a particular type. The number and kind of parameters is fixed for each instruction, but the number and type of operands is a function of the operation and the static parameters passed to an instruction.

4.1 The apply Instruction and Primitive Functions

Most of the arithmetic and logical operations of a SafeTSA program are performed using the `apply` instruction[5] which applies a primitive function (e.g., integer addition) to one or two operand variable and creates a new output variable:

ssa-var-def ← `apply` (*function*) *ssa-var-use*$_{\text{operand}}$
ssa-var-def ← `apply` (*function*) *ssa-var-use*$_{\text{operand}}$ *ssa-var-use*$_{\text{operand}}$

(The complete list of primitive functions, along with their types can be found in Appendix B.)

Except for a few primitive functions that can result in a divide-by-zero exception, none of the primitive functions have side-effects, and all of the `apply` instruction's scheduling constraints can be inferred directly from its data dependencies expressed by the SSA variables named by its input operands and result variable. The types of the input operands and result variable are specified by the type of the selected *function*.

4.2 Type Coercion Instructions

SafeTSA provides two instructions for creating a new SSA variable that points to the same dynamic object as another variable but has a different static type:

(1) The `dynamicCast` instruction is used to create a new variable whose syntactic type is a subtype of the old variable's syntactic type.
(2) The `staticCast` instruction is used to create a new variable whose syntactic type is a supertype of the old variable's syntactic type.

The difference is that `staticCast` instruction exists only to facilitate type separation, whereas, the `dynamicCast` implies a runtime check and throws a `ClassCast` exception if that runtime check fails.

The `nullCheck` instruction, mentioned earlier, can be seen as a variant of the dynamicCast instruction that coerces a possibly-null reference to an object of a particular class into a safe non-null reference to an object of that same class if the reference is non-null, but throws a `NullPointerException` when the reference is null.

The syntactic form of these instructions is:

ssa-var-def ← `dynamicCast` (*type*$_{\text{from}}$ *type*$_{\text{to}}$) *ssa-var-use*

[5]In some of the prior work, this was called the "primitive" instruction (e.g., [Amme et al. 2001]).

ssa-var-def ← `staticCast` (*type*$_{\text{from}}$ *type*$_{\text{to}}$) *ssa-var-use*
ssa-var-def ← `nullCheck` (*java-class*) *ssa-var-use*$_{\text{unsafe-ref}}$

4.3 Array Access Instructions

SafeTSA provides four instructions for working with arrays:

ssa-var-def ← `newArray` (*type*$_{\text{element}}$) *ssa-var-use*$_{\text{size}}$
ssa-var-def ← `makeElementRef` (*java-array-type*) *ssa-var-use*$_{\text{object}}$ *ssa-var-use*$_{\text{index}}$
ssa-var-def ← `getElement` (*type*$_{\text{element}}$) *ssa-var-use*$_{\text{element-ref}}$
`setElement` (*type*$_{\text{element}}$) *ssa-var-use*$_{\text{element-ref}}$ *ssa-var-use*$_{\text{new-value}}$

The `newArray` instruction is used to allocate memory for an array of a particular type and size. As we saw earlier, the `makeElementRef` is used to create a "safe element reference" by taking an array object and an integer index, and bounds-checking that index against the array object. If it is out of bounds, an exception is thrown, other wise `makeElementRef` creates a "safe" element reference variable. The `getElement` and `setElement` instructions do not need to perform any bounds-checking or address computation, they merely dereference a "safe element reference" created by the `makeElementRef` instruction.

In this way, redundant bounds-checks and address computations can be eliminated by optimizing away redundant `makeElementRef` instructions.

4.4 Field Access

Like Java bytecode, SafeTSA's instruction set includes instructions for manipulating:

ssa-var-def ← `getFieldInstance` (*field*) *ssa-var-use*$_{\text{object}}$
`setFieldInstance` (*field*) *ssa-var-use*$_{\text{object}}$ *ssa-var-use*$_{\text{new-value}}$
ssa-var-def ← `getFieldStatic` (*field*)
`setFieldStatic` (*field*) *ssa-var-use*$_{\text{new-value}}$

The `getFieldStatic` loads the contents of a particular static (i.e., per class) field into its result variable. This instruction's *field* parameter is an index into a set of fields declared at the beginning of the SafeTSA file.

Similarly, the `getFieldInstance` instruction is parameterized by an index into the field declaration table, but instance *field* declarations (i.e., for per object fields) need to be combined with an object reference (which is passed as a second parameter) in order to be resolved to a particular field instance. The field declaration's "owning class" determines the type of this object reference, and the type of the field determines the type of the `getFieldInstance`'s result variable.

The `setFieldStatic` and `setFieldInstance` instructions are similar to the corresponding "getField" instructions, except that instead of creating a result variable, they take an additional operand (whose type is determined by the field type) and write the value of that operand to the specified field.

It should be noted that these instructions semantics involve interactions with the virtual machine heap memory. Because of this, their inter-dependencies are not

completely captured by their input operands and output result variables and must be taken into consideration during optimization and scheduling. One mechanism for doing this is to thread these instructions together using explicit "heap memory variables." Each instruction that can potentially access the heap memory in any way (e.g., `getFieldStatic`, `setFieldStatic`) would have an extra input operand representing the state of the memory heap before the instruction executes, and those that modify any part of heap memory (e.g., `setFieldStatic`) would also have an addition output variable representing the new memory heap. These heap memory variables would make explicit, in the SSA-based IR, a conservative approximation of the inter-instruction data dependencies going through heap memory, and thus, allow optimizations and instruction scheduling to assume that all scheduling constraints are expressed by the SSA variable naming. This is not done in SafeTSA itself, but can be done in the optimization phase intermediate representations used to generate SafeTSA from Java and/or machine-code from SafeTSA.

4.5 Method Calls

There are three kinds of instructions that result in method invocations:

$$\begin{array}{rl} \textit{ssa-var-def} \leftarrow & \texttt{dispatch}\ (\textit{method})\ \textit{ssa-var-use}_{\text{object}}\ \{\textit{ssa-var-use}_{\text{argument}}\}^* \\ & \texttt{dispatch}\ (\textit{method})\ \textit{ssa-var-use}_{\text{object}}\ \{\textit{ssa-var-use}_{\text{argument}}\}^* \end{array}$$

$$\begin{array}{rl} \textit{ssa-var-def} \leftarrow & \texttt{callInstance}\ (\textit{method})\ \textit{ssa-var-use}_{\text{object}}\ \{\textit{ssa-var-use}_{\text{argument}}\}^* \\ & \texttt{callInstance}\ (\textit{method})\ \textit{ssa-var-use}_{\text{object}}\ \{\textit{ssa-var-use}_{\text{argument}}\}^* \end{array}$$

$$\begin{array}{rl} \textit{ssa-var-def} \leftarrow & \texttt{callStatic}\ (\textit{method})\ \{\textit{ssa-var-use}_{\text{argument}}\}^* \\ & \texttt{callStatic}\ (\textit{method})\ \{\textit{ssa-var-use}_{\text{argument}}\}^* \end{array}$$

The `callStatic` instruction is parameterized by an index into the list of static method's declared at the beginning of the SafeTSA file. The `callStatic` instruction also requires an operand corresponding to (and having the type indicated by) each of the specified method's formal parameters, and if the method has a non-void return type, the `callStatic` instruction will have a result variable of that type.

The `callInstance` instruction is similar to the `callStatic` instruction, except that it also takes the receiver object as an addition operand. This receiver object is passed to the callee, as an implicit "this" (a.k.a. "self") parameter. Like `callStatic`, the method invoked by `callInstance` is determined statically (based on the method declaration's name, argument types, and owning class) and is unaffected by the receiver object's dynamic class. The *callInstance* instruction does, however, differ from just making the receiver object an extra parameter of method called with `callStatic` in that:

(1) `callInstance` requires that the method not be static, whereas `callStatic` requires that the method is static.
(2) The implicit receiver object is not listed among the method's formal parameters, and its type is determined by the owning class of the method.
(3) The static type of the implicit "this" parameter received is determined by the callee is a safe reference to the class where the callee is defined.

(4) The types given the "this" parameter by the callee and by the caller can differ. If the owning class of the method declaration inherits the method definition from a superclass, the callee types the "this" as a supertype of the type used in the caller. This is safe, but never happens with regular function parameters.

The `dispatch` instruction is used to implement the standard Java invocation of a non-static method. It differs from the `callInstance` in that the implementation to be invoked is determined by the receiving object's dispatch table (and thus, reflects the overriding implementations of the method found in the receiver object's dynamic class or that class's superclasses.) Like the callee of `callInstance` invocations, the callee of `dispatch` invocations, receives an implicit "this" parameter whose type is a reference to an object of the class in which the callee is defined. This is safe, because a class's implementation of a method may only be in that class or in any superclass of that class, so the static type of the "this" parameter will always be the dynamic class of the receiver object or a superclass of that receiver object. This may, however, be a subclass of the static type of the receiver object in the caller. In effect, the dynamic dispatch sometimes "includes," for free, a safe dynamic cast from a supertype to a subtype.

The distinct behavior of `callInstance` (as compared to the standard `dispatch` instruction) is needed to implement Java's "super" method invocations, which allow the implementations of methods to explicitly directly call their respective superclass's implementations of methods, avoiding the standard dynamic dispatch. This same distinction is found in Java bytecode, which has an `invokespecial` (which is equivalent to SafeTSA's `callInstance`) as well as `invokevirtual` and `invokeinterface` (which both would be rendered in SafeTSA with `dispatch`).

In addition, there is a `newObject` instruction that allocates memory for an object (of the class that owns the specified constructor) and executing a constructor specified with an index into the set of declared constructors:[6]

ssa-var-def ← `newObject` (*constructor*) $\{$*ssa-var-use*$_{\text{argument}}\}^*$

5. CONTROL STRUCTURE TREE

The SafeTSA instructions are embedded into the leaves of a tree that represents each SafeTSA method. This Control Structure Tree (CST) retains the high-level control structures of Java source programs and reflects the nesting of control structures within the program. The SafeTSA language is designed so that a programs control flow graph and dominator tree can be derived from the program by wiring together blocks of instructions according to the rules of the enclosing control structures.

[6]In addition, to the call to a constructor by the method requesting the creation of an object, each constructor also contains calls to their superclass constructor. This is done by overload `staticCall` to also take *constructor* declarations, but for safety, it needs to be verified that *constructor* `staticCall`'s occurs at the beginning of every constructor, and nowhere else, and that the constructor is always one of the constructors of the superclass. It would probably be more consistent with SafeTSA's philosophy if these "superclass constructor calls" were hard-coded into the constructor definition control structure tree grammar.

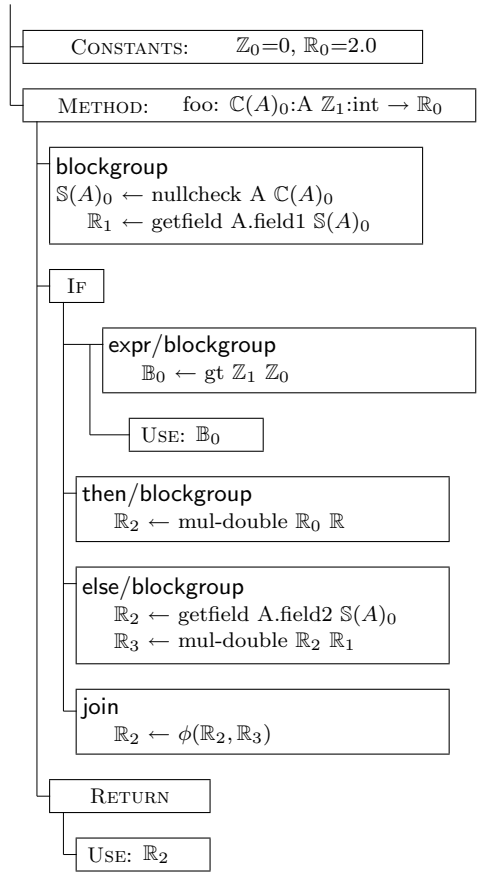

Fig. 8. A.foo's Control Structure Tree and Instructions

As a concrete example, consider Figure 8, which shows how SafeTSA embeds the instructions from Figure 6 into a control structure tree that retains the information about the if and return statements in the original program (Figure 1). In the following sections, we will discuss the different types of CST nodes, their contents, and constraints on their use.

5.1 CST Nodes for Expressions

This CST of a program will normally contains many of the same nodes (particularly, those corresponding to statements) and substantially the same nesting structure as would an abstract syntax tree of that Java program. The SafeTSA program differs from the source program in that the source language's nested expressions are flattened into "block groups" containing individual instructions that are executed sequentially:

blockgroup → `BLOCKGROUP` {*instruction*}*
expression → *expr-if* | blockgroup

There are, however, no instructions for intraprocedural control flow,[7] nearly all control flow is implicit in the statement-level CST nodes:

statement → *if* | *switch* | *for* | *do* | *while* | *try-catch*
| *break* | *continue* | *void-return* | *return-value* | *throw*
| *expression*

The statement-level CST nodes, however, are not convenient for representing the control flow implicit in Java's short-circuit operators: &&, ||, and ?:. For this reason, the CST grammar includes an *expr-if* production that allows the short-circuiting behavior of the operators to be expressed as a special kind of if-statement-like *expression*.

expr-if → `EXPRIF`
expr-with-value$_{\text{condition}}$
expression$_{\text{then}}$
expression$_{\text{else}}$
join
expression$_{\text{follow}}$
expr-with-value → `EXPRWVAL` *expression* *ssa-var-use*$_{\text{value}}$

The *expr-if* production governs a CST node, EXPRIF, and five children of that node:

(1) The *condition* child is a *expr-with-value* subtree which contains a sub-expression (normally, a *block group* of instructions, but possibly also a nested *expr-if* subtree) and a *condition value*. The *condition* sub-expression contains anything that needs to be computed before deciding if the *condition* is true or false. The *condition value* is the name of the boolean ssa-variable that will be used at runtime to decide whether to execute the *then* or the *else* sub-expression. Often the boolean ssa-variable is the result of the last instruction in the *condition* sub-expression, but this is not required.
(2) The *then* child is an *expression* subtree whose contents will only be executed if the *condition*'s *ssa-variable* is true.
(3) The *else* child is an *expression* subtree whose contents will only be executed if the *condition*'s *ssa-variable* is false.
(4) The *join* child contains any SSA ϕ-functions needed to merge SSA variables produced in the *then* or *else* expressions and which need to be referred to outside of those expressions.
(5) The *follow-expression* child contains a sub-expression that will be executed after either the *then* or the *else* subtree completes and the ϕ-function assignments are available. (This is necessary because the original program may have a short-circuit operator embedded in a larger expression.)

For example, the Java expression, (x == y ? 0 : f()), could be translated into SafeTSA using an *expr-if* expression. The *condition* of this *if-expr* would be a *expression-with-value* whose *sub-expression* would be a blockgroup containing an

[7]Some instructions, however, can raise exceptions, which will transfer control to the end of the innermost dynamically surrounding try statement, which could be within the same procedure.

apply `ieq` comparison instruction and whose *value* would be the *ssa-variable* representing the result of the comparison instruction. The call to f() could be placed in the *else* subtree, and a ϕ-function in the *join* would create a variable representing the result of ?: (either 0 or the value returned by f()). In this case, the *then* subtree and *follow-expression* subtrees could remain empty (i.e., *then* would be an empty statement list, and *follow-expression* would be an empty blockgroup).

If the a ?: is a subexpression of a larger expression, the parts of the expression that should be executed before the ?: can be placed before the comparison instruction in the *expression* part of the *condition*, and the parts of the expression that should be executed afterwards can be placed in the *follow-expression*

The other Java short-circuit operators (&& and ||) can also be rendered using the *expr-if* construct by considering a&&b shorthand for a?b:false and a||b as shorthand for a?true:b. (In cases where it is possible to determine at compile time that b does not have any side effects, then it is possible to avoid the *expr-if* construct altogether and simply use the *boolean-and* and *boolean-or* primitive functions.)

Even though expressions contain arbitrarily nested *expr-if*'s, they never contain loops or other abruptly-terminating elements. Thus, an *expression* always corresponds to a directed, acyclic, single-entry region of the program's CFG that has a single non-exception exit. As a result the *join* child always has exactly two predecessors: *then* and *else*.

5.2 If Statement CST Nodes

Overall, however, the statement-level IF CST node is quite similar to the expression-level IFEXPR CST node:

if → `IF`
expr-with-value$_{\text{condition}}$
statements$_{\text{then}}$
statements$_{\text{else}}$
join

Syntactically, there are two obvious differences. The first is that the *if* production only has four children. The statement-level IF node does not need anything corresponding to *expr-if*'s fifth child *follow-expression*, because control structures allowing nested statements always use the *statements* production, which can include as many individual *statement* subtrees as is necessary. (In contrast, when an *expression* is nested, the surrounding control structure always requires exactly one *expression*.)

The second readily apparent difference between the *if* and *if-expr* productions is that IF's *then* and *else* children are *statements* while IFEXPR's *then* and *else* each contain an *expression*. This means that an if statement's *then* and *else* clauses may contain loops or statements (e.g., return) that do not complete normally. As a result the CFG region represented by an if statement is not constrained to representing a DAG and may have multiple exit points.

5.3 Normal Completion and Degenerate Join Nodes

The Java Language Specification (JLS) [Joy et al. 2000] classifies statements that end with execution being transferred somewhere other than the next statement as completing abruptly and the others as *completing normally*. Some types of statements always complete abruptly including break, continue, return, and throw statement. Other kinds of statements will complete normally unless a sub-statement completes abruptly. This is important because the JLS requires compilers to perform a particular conservative analysis that identifies particular statements in a program that always complete abruptly. Based on this analysis, the JLS specifies that it is a compile-time error to place an "unreachable" statement immediately after one of these always-abruptly-completing statements. In addition, the JLS specifies that it is an error if a non-void method does not end in one of these always-abruptly-completing statements. (Since a *return* always completes abruptly, this requirement is satisfied if the the method ends in a return statement.)

SafeTSA also enforces this distinction, and so statement CST nodes are statically classified as either, *potentially completing normally* (PCN) or *never completing normally* (NCN). (Like the JLS analysis, this PCN/NCN classification is conservative, and all subtrees classified as NCN will—in fact—never complete normally, but those classified as PCN do not necessarily have feasible execution paths that result in a normal completion.)

The PCN/NCN distinction is important for two reasons:

(1) SafeTSA follows Java and forbids a certain class of unreachable statements. To this end, in SafeTSA, only the last *statement* of a *statements* production is allowed to be NCN, because otherwise any *statement* following a NCN *statement* would be unreachable. If this last statement is NCN then the entire *statements* subtree is considered NCN, but if it is PCN then the entire *statements* subtree is PCN. In addition, like Java, SafeTSA requires that the last statement of non-void methods be NCN, because otherwise execution could potentially fall off the end of a method and begin executing arbitrary code. (Last statements of "void methods" are allowed to be either PCN or NCN, since it is safe to pretend the method is terminated with an empty "return;" statement.)

(2) Many SafeTSA control structures have *join nodes* which contain ϕ-functions. Each ϕ-function associates an input-operand with each control-flow in-edge of the *join node*, and by not considering NCN *statements* as predecessors of *join nodes*, many unnecessary and meaningless ϕ-function operands can be avoided. This procedure also reflects Java's definite assignment rules; assuming a straight forward translation from Java to SafeTSA, any of the original Java program's variables will be definitely assigned at a join node in its control flow graph if and only if the SafeTSA program has one SSA variable corresponding to that source program variable at each of the join node's PCN predecessor. These SSA variables will be the parameters of the ϕ-function corresponding to that source program variable.

Because of the infeasibility of some CFG paths, some of the syntactic *join nodes* included in our CST are not in fact "real" *join nodes* in the CFG. If a syntactic *join node* has less than two PCN predecessors in the CFG, then it does not actually

join anything and is considered a *degenerate join node*, and should not contain ϕ-functions. If a *degenerate join node* has no PCN predecessors then the *join node* itself is considered NCN; otherwise all *join nodes* are PCN.

The analysis of whether a CST tree is PCN or NCN is recursive and depends on the classification of it subtrees (and—for join nodes—their CFG predecessors): all expression CST subtrees are considered PCN, but many statement CST subtrees' PCN/NCN classification is a function of the PCN/NCN classification of their children. An *if* subtree will be NCN only if and only if both its *then* child and *else* child are NCN, and the *if* statement's *join node* will be degenerate whenever either the *then* child is NCN or the *else* child is NCN.

5.4 Switch and Break Statements

A Java switch statement consists of the switch keyword, an expression of the byte, char, short, or int type, and it has a block containing 0 or more case causes. Each case clause contains a constant value trigger, and a set of associated statements. The type of the value constant value must be assignable to the type of the switch statement's expression. In addition to the constant value case clauses, there may be a single "default" clause, which does not have an associated constant value, but does also have a set of associated statements. The switch statement is executed by first evaluating the expression, if the value of that expression matches the constant value of one of the case clauses, execution will continue with the statements of that case clause. If not, and there is a default clause, control will be transferred to that default clause. Otherwise control will be transferred directly to the end of the switch statement block. If the statements associated with one the case clauses or the default clause completes normally, control will "fall through" to the next clause and its statements will be executed; this process will continue until a clause's statements fail to complete normally or the end of the switch statement block is reached. The most common reason for a case clause to not complete normally, is because of an associated break statement; when such a break statement is encountered, control is immediately transferred to the end of the switch statement.

switch → `SWITCH` *label-def*
type *expr-with-value*$_{\text{cond}}$
{`CASE` *constant* *join*$_{\text{fallthrough}}$ *statements*}*
`DEFAULT-CASE` *join*$_{\text{fallthrough}}$ *statements*
{`CASE` *constant* *join*$_{\text{fallthrough}}$ *statements*}*
join$_{\text{break-phis}}$

break → `BREAK` *label-use*

SafeTSA's *switch* production differs from Java's switch statement in a couple important ways:

(1) In Java, switch statement and break statement labels are optional, in SafeTSA they are required even if the statement has no breaks which use them or a break is to be associated with the innermost "breakable control structure." (In the encoded SafeTSA, however, the explicit labels are replaced with an implicit numbering that based on the current nesting of breakable control structure.)
(2) The *type* (either byte, char, short, or int) is selected explicitly before the expression or any of the constant values are given.

(3) There is always exactly one default case clause. In other words, SafeTSA always requires an explicit default case, but its order with respect to the value-qualified case clauses is not constrained. If needed Java's implicit default behavior can be simulated by inserting an explicit default case containing only a break statement as the first clause in the switch subtree.

SafeTSA's *switch* CST subtrees, also have to take into account the potential *join nodes* associated with the SWITCH node. Each clause has a syntactic *join node* in the grammar, but the first case never has a non-degenerate join, since the only way to reach it is when the switch statement's expression value matches that case's constant value. Subsequent case clauses have non-degenerate joins if and only if the *statements* of the immediately preceding clause are PCN, and thus, execution can "fall through" to that case from the previous one. In addition to the "per-clause" *join nodes*, the SWITCH has a syntactic "*break-phis*" *join node*. A *switch*'s *break-phis*' CFG predecessors are all of the break statements associated with that *switch* plus the normal completion of last clause in the *switch* subtree if that clause's statements are PCN. If neither exist, then the *break-phis* node has no predecessors, and the *switch* subtree is NCN.

5.5 While Loops and Back-Edges

SafeTSA programs like Java source programs always have reducible control flow graphs. This means that the edges in the program's CFG can be partitioned into "forward" edges and "backwards" edges, such that if the "backwards" edges are removed from the CFG, what remains is a directed acyclic graph, and the source and destination of the back-edges are such that the destination always dominates source on both the DAG of forward edges and also on the complete CFG. All of the CFG edges associated with expressions, if statements, and switch statements are always forward edges, but Java has three loop constructs that create back-edges in the program's CFG: do loops, while loops and for loops.

while → `WHILE` *label-def*
join-types$_{\text{loop-phis}}$
expr-with-value$_{\text{condition}}$
statements$_{\text{body}}$
join-values$_{\text{loop-phis}}$
join$_{\text{exit-phis}}$

Of the Java loop constructs, while loops are probably the simplest.[8] In a manner similar to IF nodes (which have a *condition* expression whose value determines whether the *then* or *else* statements will be executed) each SafeTSA WHILE node has a *condition* expression whose value that determines whether the *body* statements are to be executed. The difference is, that whereas the IF node's *condition* is only evaluated once each time the IF is executed, the WHILE's *condition* will be re-evaluated each iteration through the loop in order to determine if the *body* should be executed an additional time.

[8]Especially if we ignore continue statements, the discussion of which, we will defer to the next section.

Because of this, *condition* is preceded by a *join node* whose CFG predecessors include the predecessor of the entire *while* statement and the normal completion of the *body*. Because the CFG edge from the completion of the body to this "*loop phis*" join node is back-edge, the *loop-phis* join node is somewhat different from the *join nodes* found in *if*, *expr-if*, or *switch* statement subtrees.

This difference manifests itself at the syntactic level in the way the ϕ-functions held in these *join nodes* are expressed. In the case of any non-loop *join node*, the definitions of all of the variables used as parameters of ϕ-functions in that *join node* will syntactically precede that *join node*. (This is important, because in general, SafeTSA does not allow "forward references" to undefined entities; there is always a definition or at least declaration of variables, statement labels, etc. before they are used.)

The "back-edge *join nodes*" found at the head of loops are, however, more complicated than the non-loop *join nodes*. For example, a *while* loop's *loop-phis* may contain a ϕ-function for a source program variable that is used and modified in the *body* of the loop, and in this case, there is no syntactic location at which the *join node* can be placed that will avoid necessitating "forward references". If the *join node* (with its ϕ-functions) is placed at the beginning of the loop, then the uses of the variable in the *body* can properly refer to that ϕ-function's result, but the ϕ-function itself would need a forward reference in order to access the definition in the *body*. Conversely, if the *join node* is placed at the end of the body, its ϕ-functions can successfully refer backwards to the appropriate variable definitions, but uses of the ϕ-function's result in the body will require a forward reference to these ϕ-functions at the end of the body.

The solution adopted by SafeTSA is that, for all ϕ-functions associated with loop back-edges, the "declaration" of the ϕ-function assignments and their types (which can be placed at the beginning of the loop and named *join-types* in the grammar) are separated from the "definition" of the ϕ-functions where their parameters are given (which occur syntactically at the end of the loop body and is named *join-values*).

The other *join-node* found in the *while* production is the *exit-join*. It is necessary, because like switch statements, Java while loops can be exited by break statements embedded inside the body. The *exit-join* serves the same role as the *switch* production's *break-phi* join node. That is, *exit-join* represents the normal exit of the *while* statement and it can be reached from the *condition* (when its value is false, thus exiting the loop) or from any break statements in the body that are associated with the while loop. In the absence of break statements, the *exit-join* will be degenerate.

5.6 Do Loops and Continue Statements

SafeTSA's *do* production is similar to *while* production. The main difference is that the *condition* comes after the *body* both syntactically and in the dynamic execution:

do → `DO` *label-def*
join-types$_{\text{loop-phis}}$
statements$_{\text{body}}$
join$_{\text{continue-phis}}$
expr-with-value$_{\text{condition}}$
join-values$_{\text{loop-phis}}$
join$_{\text{break-phis}}$
continue → `CONTINUE` *label-use*

A second apparent syntactic difference is that do loops have an addition *continue-phis* join node. All of the Java loop constructions (i.e., for, do, and while loops) can have associated *continue* statements, which are like *break* statements and never complete normally, but they only skip to the end of the loop body and repeat the loop test, rather than exiting the entire structure. The *continue-phis* join node has as its predecessors: the normal completion of the *body* (if it is PCN) and all of the do statement's associated continues inside the *body*. (For *while* loops, each associated continue statements simply adds an additional predecessor to the *while* statement's *loop phis*.) In the absence of continue statements, the *continue-phis* join node is degenerate, and if it has no PCN predecessors, then the *condition* will be unreachable and *loop-phis* will also be degenerate.

This also effects *break-phis*, since its predecessors are the *condition* if it is reachable and all of the associated break statements inside the loop body. Normally, Java loops are PCN, but if the *condition* of a do loop is unreachable and there are no associated break statements, then the *break-phis* join node will also be unreachable and the entire do statement will be NCN. If there are no breaks, or the *condition* is unreachable and there is only one associated break statement, then *break-phis* is a degenerate join node, but the entire do statement is PCN.

5.7 For Loops

Although the *for* statement production looks more complicated than the *while* and *do* productions, the behavior of for statements is actually quite similar to that of while statements:

for → `FOR` *label-def* *expression*$_{\text{init}}$
join-types$_{\text{loop-phis}}$
expr-with-value$_{\text{condition}}$
statements$_{\text{body}}$
join$_{\text{continue-phis}}$
expression$_{\text{step}}$
join-values$_{\text{loop-phis}}$
join$_{\text{break-phis}}$

The main difference is that it adds an *init* expression that is evaluated once before the loop is entered and a *step* expression that is evaluated during each iteration after the loop body. In SafeTSA, the *init* expression's content could just as easily be moved to a new statement directly preceding the for, but the *init* is retained for fidelity with the source program.

One might think that this is true of the *step* expression as well, and that a for loop is just "syntactic sugar" for a *while* statement preceded by an *init* statement

and whose body has an extra *step* statement tacked on the end, but this would be incorrect. A *continue* associated with *while* statement should transfer control to the end of the *for* statement's *body* immediately before the *step* statement (rather than the end of the *step* statement), so that the *continue*'s associated with the for statement are predecessor of the *continue-phis* join node (as is, the normal completion of the *body*, if the *body* is PCN). If there are no associated continues and the *body* is NCN, then the *step* will be unreachable and the *loop-phis* join node will be degenerate (since the back-edge is infeasible and the loop is not really a loop). If the body is PCN and there are no associated continues, or the body is NCN and there is only one continue then *continue-phis* join node will be degenerate. As with *while* loops the *break-phis* join node will be degenerate if and only if there are no associated break statements in the loop body, but in all cases the entire *while* subtree is always PCN.

5.8 Return Statements

SafeTSA has two productions for return statements, but only one is allowed in any given context:

void-return → `VOIDRETURN`
return-value → `RETURN` *expr-with-value*

The *void-return* production is equivalent to the Java statement "return;" and is only allowed in constructors and methods with a void return type. It execution serves to terminate the current method and return control to the caller. The *return-value* production is used to represent all other Java return statements, and is allowed only in methods with non-void return types. The type of the expression value must match the return type of the method. Both types of return statements are NCN.

5.9 Try Catch Statements and Exceptions

Java's try statements are used to manage exceptions, which may be produced by exceptional conditions during the execution of various instructions or by the explicit execution of throw statements in the current method or in a method that is called by the current method.

try → `TRY` $statements_{\text{body}}$
$join_{\text{exceptions}}$
{`CATCH` *java-class ssa-var-def* $_{\text{exception-object}}$ *statements*}*
$join_{\text{normal-completion}}$
{`FINALLY` {*join statements*}*}?
throw → `THROW` *type expr-with-value*

Java's semantics require that the local variables of the method containing the try/catch block be available during the handling exceptions within *catch* (and *finally* clauses. In order to allow this SafeTSA uses a conservative static analysis to determine at what locations in the try body and catch clause statements exceptions may occur. This is done by classifying a subset of individual instructions as *potentially exception-causing instructions* (PEI) and by identifying explicit throw statements. (PEI instructions include both instructions that will—under some circumstances—cause exceptions directly, (e.g., division instructions that throw an exception if the

denominator is zero) and also all "call" instructions (i.e., *newObject*, *callStatic*, *callInstance*, and *dispatch*), since—in general—the called method may throw an exception rather than completing normally.) At runtime, if an exception occurs, one of the catch clauses will be chosen to execute based on the dynamic class of the exception object raised at the exception-causing location. The SafeTSA representation, however, conservatively treats all of the potentially exception-causing locations as predecessors of a single *exceptions* join node, even when a static type analysis might be able reduce the number of catch clauses reachable from particular potentially exception-causing locations. Furthermore, this *exceptions* join node is treated as the predecessor of all of the catch clauses.

The *normal-completion* join node has as its predecessors the try statement body (if it is PCN) and the catch clauses that are PCN. A try statement (without a finally clause) is considered PCN if the try statement body is PCN or at least one of the catch clauses is PCN. (If there is a finally clause, then it also has to be PCN, for the entire try statement to be PCN.)

The *try/catch* production is further complicated by support for Java's "finally" clauses, which are executed whenever the try/catch/finally statement is exited, whether it be from an uncaught exception, a return, a break, a continue, the normally completion of the body of the try, or the normal completion of one of the catch clauses. This is because it is difficult to generate SSA code that will have the correct semantics in all the contexts (local variables, predecessors and successors), in which the finally code must execute. As a result SafeTSA FINALLY clauses[9] contain multiple copies of the finally clause statements, one for each successor node the finally statements may have:

—One for the normal completion of the try body (if it is PCN) and the catches that are PCN. (if applicable)

—One for the uncaught exceptions in the try body and the exceptions thrown in any of the catch clauses. (if applicable)

—One for each surrounding control structure that has associated break statements inside the try body or one of the catches. (if applicable)

—One for each surrounding loop that has associated continue statements inside the try body or one of the catches. (if applicable)

—One for the return statements inside the try statement. (if applicable)

Each of these copies of the finally statements will then be executed in the appropriate context.

6. ENCODING OF SAFETSA

6.1 Inherent Safety

At the lowest level, a program representation, such as SafeTSA, is a mapping between binary strings and computer programs. We consider a program representation to be inherently safe with respect to some properties if it maps every binary strings to a program that satisfy those properties.

[9]Our current SafeTSA implementation does not correctly handle finally clauses.

The properties that SafeTSA's encoding guarantees include syntactic correctness, type separation, and dominator scoping. The syntactic correctness of SafeTSA classes is governed by context-free grammar which controls the correct placement of fields and methods within the class, the placement of control structures within the method and nested within other control structures, and the placement of SafeTSA instructions within the control structure. The placement of certain control structure elements is also constrained by the PCN/NCN reachability analysis described above. Instruction operands are constrained by type separation and dominator scoping, such that it is impossible for undefined or incorrectly typed values to be used by a SafeTSA instruction.

In addition to these properties, there are a few additional properties that, because SafeTSA supports Java's binary compatibility rules, must be verified at link time. These include the subclass and interface relationships, inter-class consistency of the method and field signatures, and compliance with the access modifiers of members of external classes. Unless these verifiable inter-class properties are violated, SafeTSA's inherent properties guaranty that SafeTSA classes cannot violate the type and memory safety constraints of the Java virtual machine.

In order to convert the in-memory tree structure representing a SafeTSA class into a flat on-the-wire representation, the encoder performs a depth-first traversal of the CST serializing the contents into a sequence of symbols. Each node is serialized by the symbol for its node type, followed by the serialization of its children and other contents (e.g., SSA instructions) as appropriate for that node type.

The ordering of the symbols is designed in such a way, so that for each point in the program, it is possible—without looking at the symbols after that point—to enumerate the set of candidates for the next symbol that can result in a valid program. In other words, the validity of any symbol depends only on the sequence of prior symbols and not on later symbols. This property is established in different ways for different kinds of symbols.

The most fundamental constraint on Control Structure Tree nodes comes from an context-free LL1 grammar which governs the nesting of control structure nodes within the control structure tree. Grammar production corresponds to the types of a CST nodes, so that whenever a node type is required, the list of valid node type candidates can be enumerated based on its parent and predecessor nodes and the grammar. Likewise, valid targets for break and continue nodes can be enumerated by its examining parent nodes to determine the control structures they are nested within. It should be noted that it is possible to maintain and incrementally update the control flow graph and dominator tree as each new CST node is updated.

SSA Instructions are serialized in a similar manner. The first symbol identifies the instruction opcode, and then, depending on the opcode, it may be followed by the serializations of types, method identifiers, etc. that parameterize the instruction. Given these, the presence of a result SSA variable—and its type—and the number—and types—of all operands are determined, and the individual operands may then be serialized.

In this way, the entire SafeTSA program can be converted to a sequence of symbols, each chosen from an enumerable pool of valid symbols. Given this, it is possible to create a binary prefix encoding, such that the beginning of any sequence

of bits will select one symbol from the enumerated pool. If the remaining bits, are then applied iteratively to each next symbol in the sequence, the whole program can be rendered as a binary stream, and each binary stream will correspond to a valid program, a valid program with extra bits at the end, or the beginning of a valid program.[10]

6.2 Encoding Operands

If SafeTSA could be processed according to a pre-order traversal of the dominator tree, the numbering scheme described in Section 3.3 could be easily implemented by maintaining an array-backed stack of operands of each type. As each instruction would be processed, the operands would simply be represented by their index in the array, and the instructions output result could be pushed onto the top of the stack. As each each block of instructions went out of scope, their result values would be popped off the top of the stack.

In practice, it is not quite as simple. Although SafeTSA's CST constrains the control flow graph and ensures that each instruction is only processed after all the instructions that dominate it, instructions are not necessarily processed according to a strict pre-order traversal of the dominator-tree. This is typically because NCN sub-statements allow more precise dominator information to be computed than can be derived from the general form of a control structure.

For example, compare

```
L2: if (b)
L3:   x = 1;
    else
L4:   throw new RuntimeException();
L5: return x;
```

with the program in Figure 1 and its dominator tree (Figure 5). (Note that the statements here are labeled so they correspond structurally with the similarly labeled blocks in Figure 5.) In that program $\mathcal{L}_5$ is preceded in the control flow graph by $\mathcal{L}_3$ and $\mathcal{L}_4$ but $\mathcal{L}_5$'s immediate dominator is $\mathcal{L}_2$. In this program, however, L4 completes abnormally, so L5's predecessor is L3, and L3 is the immediate dominator of L5. SafeTSA's CST grammar, however, does not take this into account when ordering the components of the if statement, so they would be processed in the order: L2, L3, L4, L5. This is not a pre-order traversal because L3 and L4 are siblings in the dominator tree but L5 is a child of L3. The problem is that the SSA variables defined in L3 have to be removed popped off the stack before processing L4, but need to be put back on the stack before processing L5.

In addition, SafeTSA ϕ-function operands are processed at the end of their enclosing control structure, but they need to look up their operands according to the SSA variable scope of the corresponding join node predecessor.

To handle these complications, the basic array-backed stack needed to be augmented with an ability to make a snapshot (which is backed by a copy-on-write

[10] As implemented, SafeTSA has two exceptions where a binary string may not be a prefix of a valid SafeTSA program if it contains a string constant with UTF-8 encoding errors or if there is an instruction that needs to an operand of a "safe" type for which there are no instances available.

Table I. Class-wide Symbols and Constants for `A`

Type	Value	Index
integer constant	0	$\mathbb{Z}_0$
double constant	2.0	$\mathbb{R}_0$
class	java.lang.Object	0
class	A	1
static method	A.foo: $\mathbb{C}(A)_0 \times \mathbb{Z}_1 \rightarrow \mathbb{R}$	0
instance field	A.field1: $\mathbb{D}$	0
instance field	A.field2: $\mathbb{D}$	1

mechanism), to perform look-ups on the snapshots, to take intersections of the snapshots (used to merge the scopes of the predecessors of join nodes), and the ability to replace the primary stack with a snapshot or intersection of snapshots. The complexity of these operations is proportional to the distance in the dominator tree between a snapshot and its use, which is usually small.

6.3 Low Level Encoding

Thus at a low-level, SafeTSA files can be thought of as consisting of a sequence of symbols that make up the compilation unit. Each symbol is chosen from an implicitly enumerated finite set of alternatives whose membership is determined only be the preceding symbols. The number for the choice is encoded using a binary prefix encoding based on the size of the alternative set. This can be imagined as a complete binary tree where the different members of the alternative set are the leaves of the tree and the two edges descending from each node are labeled "0" or "1" and the prefix encoding is the path from the root to the leaves. The SafeTSA encoder uses a constant time algorithm to compute the prefix encoding from the number of the choice in alternative set and the size of the alternative set. Similarly, the decoder is able to use a constant time algorithm to recreate the choice from the alternative set and the binary sequence. This encoding does not involve a probabilistic compression of the data; but better compression could be achieved–at the cost of increased decoding time—by using a context-dependent dynamic model of symbol probabilities to create a binary tree with high-probability leaves at a lower depth (and a shorter prefix encoding) in a manner similar to a Huffman encoding, or the probabilities could be used to drive an arithmetic encoding of the symbols.

The symbols that compose the high level elements of the SafeTSA file come in several different types which dictates how their alternative set is constructed. Each constant, for example, is serialized as one symbol identifying the kind of constant (int, float, string, etc.) and then a variable number of symbols that encode the actual constant based on its kind: integer constants are encoded using a straight forward enumeration from 0 to $2^{32} - 1$; strings, however, are serialized a sequence of choices out of an alternative set of 256 possibilities where different values of each UTF-8 byte maps to the enumerations 0 to 255 and the enumeration 256 is used for an "end of string" symbol.

6.4 An Encoding Example

As an example, of how the encoding works, let us consider again the method `A.foo()` from Figure 1-8. Table I contains the class-level linkage information for class `A`, and Table II shows the binary encoding for the method `A.foo()`. The

Table II. Encoding of `A.foo()`

Symbol	Index / No. Choices	Binary Encoding
statement blockgroup	1/12	001
nullcheck	4/20	0100
`A`	1/2	1
$\mathbb{C}(A)_0$	0/1	—
getfieldInstantce	16/20	11100
`A.field1`	0/2	0
$\mathbb{S}(A)_0$	0/1	—
end blockgroup	0/20	0000
IF	3/12	011
expression blockgroup	1/3	10
apply	19/20	11111
igt	100/165	10111111
$\mathbb{Z}_1$	1/2	1
$\mathbb{Z}_0$	0/2	0
end blockgroup	0/20	0000
use:	—	—
$\mathbb{B}_0$	0/1	—
then:	—	—
statement blockgroup	9/13	1100
apply	19/20	11111
dmul	165/185	11101100
$\mathbb{R}_0$	0/2	0
$\mathbb{R}_1$	1/2	1
end blockgroup	0/20	0000
end statements	12/13	1111
else:	—	—
statement blockgroup	9/13	1100
getfieldInstance	16/20	11100
`A.field2`	1/2	1
$\mathbb{S}(a)_0$	0/1	—
apply	19/20	11111
dmul	165/185	11101100
$\mathbb{R}_2$	2/3	11
$\mathbb{R}_1$	1/3	10
end blockgroup	0/20	0000
end statements	12/13	1111
join:	—	—
ϕ	0/2	0
$\mathbb{R}_2$	2/3	11
$\mathbb{R}_3$	3/4	11
end join	1/2	1
RETURN	11/12	1111
empty expression	0/3	0
use:	—	—
$\mathbb{R}_2$	2/3	11
end statements	0/1	—

symbols in the left column of Table II are simply a linearized form of the control structure tree and instructions found in Figure 8. Each of symbols is considered to be an element of an enumeration of valid symbols at that program point, and the second column of Table II shows what that symbol's 0-based index into the enumeration and the size of the enumerated list. The third column of Table II shows the prefix code used to represent that index, which can only be interpreted if the size of the enumeration is known. The encoding for the entire method is simply the concatenation of the encoding of the individual symbols:

```
00101001111000000001110111111011111110000011001111111101100
01000011111100111001111111111011001110000011110111111111011
```

The method for creating the enumeration of valid symbols, depends on both the type of symbol demanded and on the context. For example, in the outermost context, there are 12 different kinds of statements allowed, and IF is number 10 (i.e., the 11th), so it is encoded as 11110. But inside the then and else parts of the if statement, there is a 13th possible symbol in the statement context: "end statements," which is not allowed in the outermost statement sequence, until after the RETURN. There are 20 different basic instruction types (getfieldInstance is number 16 and apply is number 19), and there are 185 primitive functions (integer greater-than comparison is number 100 and double multiplication is number 165). The operands, such as $\mathbb{R}_2$, is encoded according to type-separation and dominator-scoped naming scheme described in Section 3.3.

7. EXPERIMENTAL RESULTS

7.1 SafeTSA File Sizes

It might seem that the large number of distinct names and the extra ϕ-functions would make programs in static single assignment form (SafeTSA programs) significantly less compact than programs in stack machine code (Java bytecode programs). SafeTSA, however, has several features which reduce its file size. First, it does not explicitly represent the output variable names. Second, it is amenable to redundancy elimination optimizations that reduce the number of instructions that need to be encoded. In our SafeTSA compiler, the redundancy elimination optimizations act on the ϕ-functions as well as the regular instructions, so that optimized SafeTSA programs are transmitted in "pruned SSA form" [Choi et al. 1991]. Third, since the inherently safe encoding of SafeTSA uses a prefix encoding that only needs about $\log_2(n)$ bits to represent a symbol that was chosen from n possible valid candidate symbols. Because the number of valid candidates for operands is limited by type-separation and dominator scoping, less bits are needed for each operand than would otherwise be required, especially when encoding short methods and classes.

To assess whether these SafeTSA features mitigate any size increase, we measured the sizes of Java Grande Forum Sequential Benchmarks classes represented in:

(1) Java source code,

(2) Java classfiles,

(3) gzipped Java class files

(4) SafeTSA's encoded binary files with base-level optimizations

(5) SafeTSA encoded files after common subexpression elimination had been applied, and
(6) gzipped SafeTSA encoded files after common subexpression elimination.

It should be noted that SafeTSA's encoding features differ from standard compression techniques because they do not take advantage of probable repetitions but rather avoid wasting bits on things that are useless or impossible. In theory, the two are orthogonal, but in practice SafeTSA's binary-level encoding will mask much of the repetition from a general purpose compression routine such that it should not be expected to provide much additional compression (but it might help with some of the string constant). It is likely, however, that integrating compression techniques based on Huffman or Arithmetic encoding would provide additional compression (c.f., [Stork 2006]).

Since there are no standard benchmarks for measuring Java class file sizes, the classes of the Java Grande Forum Sequential Benchmarks were utilized. The Java Grande Forum Sequential Benchmarks are divided into three sections and are meant to assess a virtual machines performance in executing scientific and engineering code:

—Section 1 contains microbenchmarks
—Section 2 contains application kernels
—Section 3 contains complete scientific and engineering applications.

The benchmarks needed to be modified slightly in order to work around incomplete features in the SafeTSA system. In particular, the inner-classes of JGFAssignBench and JGFMethodBench were split out into separate source files: AssignTester, MethodTester, MethodTester2. In addition, the Section 1 benchmarks, JGFSerialBench and JGFExceptionBench, were excluded due to bugs in the SafeTSA front-end compiler. The Java Grande Forum Sequential Benchmark infrastructure classes that do not belong to any one benchmark were also excluded from this analysis.

The Java classfiles were obtained using the version 1.2.2 of IBM's Jikes java to bytecode compiler configured to generate no debugging information. The SafeTSA files were obtained using the front-end Java to SafeTSA compiler/encoder configured to apply a base-level of optimizations. The base-level optimizations included the propagation of constants and constant static final fields required by the Java Language Specification [Joy et al. 2000]. It also included dead-code elimination and ϕ-function folding, because those optimizations eliminate the excess phi-functions being generated unnecessarily by the front-end SafeTSA compiler providing a pruned SSA form. Version 1.3.5 of the Free Software Foundation's gzip utility was used at the default level 6, "that is, biased towards high compression at expense of speed" [loup Gailly 2002], to produce the gzipped files.

Table III gives average (mean) and median sizes for the Java classes that make up the three benchmarks sections. File sizes are given for gzipped Java source, Java class files, gzipped Java class files, base-optimized SafeTSA files, SafeTSA files optimized with common subexpression elimination (CSE), and CSE-optimized and gzipped SafeTSA files. Encoded but ungzipped SafeTSA files are significantly and consistently smaller than uncompressed Java class files. Compared to gzipped Java class files, ungzipped SafeTSA files are competitively sized. In Section 1,

Table III. Median and Mean File Sizes for each Benchmark Section

	Section 1		Section 2		Section 3	
	Median	Mean	Median	Mean	Median	Mean
Gzipped Java source	1680	1328	451	950	863	1378
Java class file	1537	3138	354	881	1215	1861
Gzipped class file	799	1081	322	611	626	974
SafeTSA files	870	1821	258	538	514	1148
SafeTSA w/CSE	870	1795	258	532	514	1116
Gzipped SafeTSA w/CSE	905	1544	307	573	501	1061
Number of Files	11		35		45	

the gzipped Java class files are smaller, but in Section 2, the SafeTSA files are smaller. In Section 3 the median SafeTSA size is smaller than the median Java class file, but the average Java class file is smaller than the average SafeTSA file. As we will see when we look at individual benchmarks and files, the difference is that SafeTSA tends to be more compact on small files, whereas gzipped class files are more compact for large files. This should not surprising, the number of bits SafeTSA needs to encode each operand increases logarithmically with the number of possible valid operands, so the size savings from SafeTSA's encoding decreases as the file size increases. Applying CSE, has only a modest effect on file size, decreasing SafeTSA files by about 1% overall. Applying gzip to SafeTSA files was not expected to be as effective as applying it to Java class files, because SafeTSA inherently-safe encoding will obscure repeating patterns because repeated symbols will not start and end at byte boundaries and may map to different bit-patterns in different contexts. Even so, gzip was able to substantially reduce the average file size for the SafeTSA files in Section 1 and Section 3.

Figure 9 shows the total size of the classes within each of the Java Grande Benchmarks. For many of the benchmarks, the SafeTSA files are as small or smaller than the gzipped Java classfiles, this is especially true in the Section 2 benchmarks. The reason for this is that compared to the Section 1 and the Section 3 benchmarks, the Section 2 benchmarks generally have smaller classes, which the SafeTSA encoding represents most efficiently. Generally the SafeTSA file sizes track closely with the compressed Java class files. The exceptions are: arith, math, and euler with the size of euler being almost entirely attributable to the euler.Tunnel class. (The arith and math benchmarks each consist of only a single class, i.e., JGFArithBench and JGFMathBench, and JGFAssignBench is the only significant class in the assign benchmark.)

The euler.Tunnel class contains a couple of large methods (specifically, calculateDamping, calculateDeltaT, and calculateDummyCells) that contain for-loops with long bodies and many local variables. The long linear sequences of instructions reduces the space-saving characteristics of SafeTSA's operand encoding (because there are a linearly increasing number of operand possibilities for each instruction, which increases the size of each operand logarithmically), and the large number of local variables increases the number of phi-functions needed. This program structure is more likely to be found in a scientific code benchmarks like the Java Grande benchmarks than in general purpose object-oriented Java programs. But even in such benchmarks that show atypically large SafeTSA file sizes, the SafeTSA files are still smaller than the ungzipped Java classfiles.

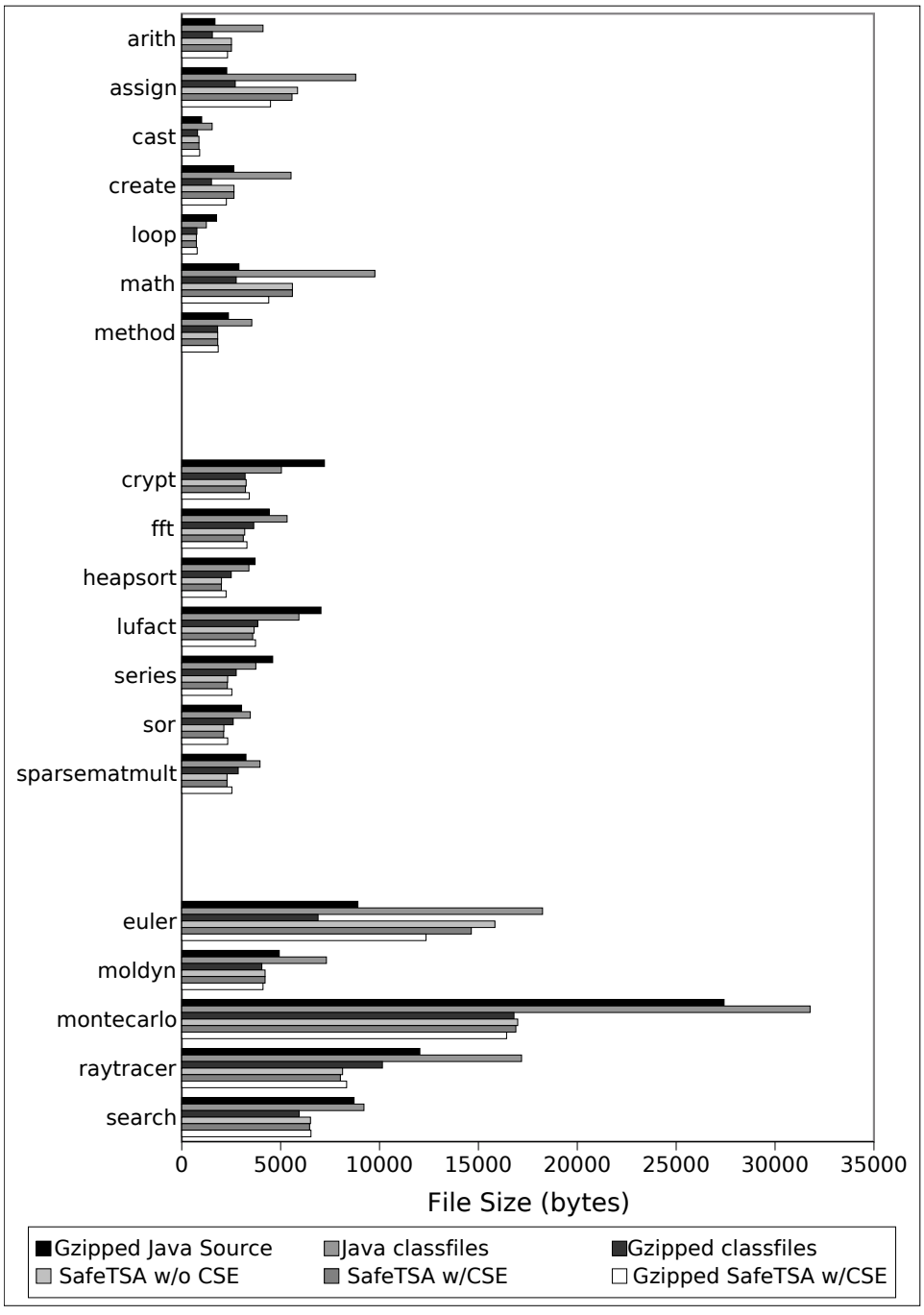

Fig. 9. Sizes of the Java Grande Forum Benchmarks

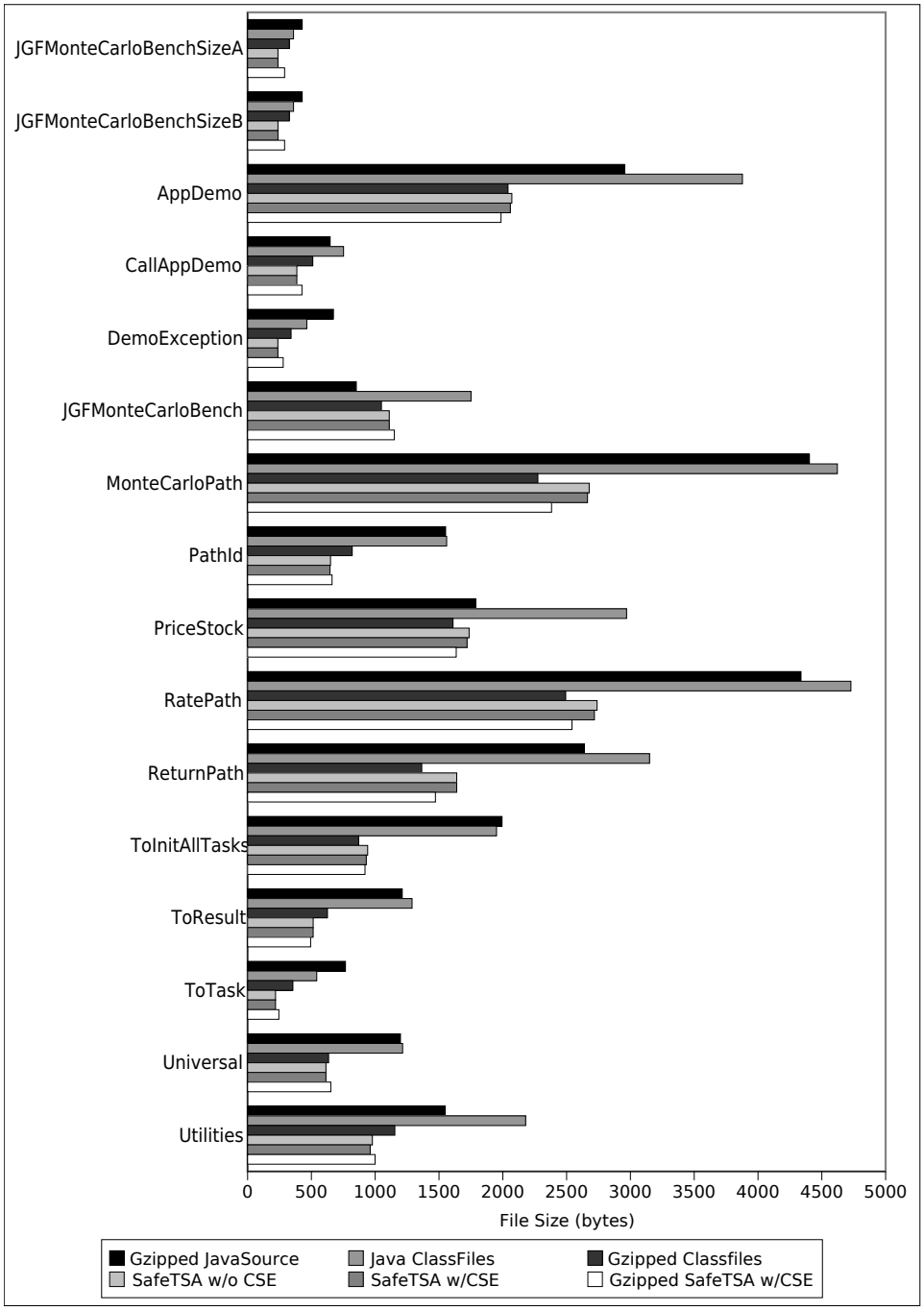

Fig. 10. Sizes of Classes in the Montecarlo Benchmark

The Montecarlo benchmark, which has the largest number of component classes, was selected for closer examination. Figure 10 shows the sizes of each of the Montecarlo benchmark's classes for the same six representations. For all of these classes, the SafeTSA files (both gzipped and non-gzipped) and the gzipped Java class files are approximately half of the size of the corresponding non-gzipped Java class file. Notwithstanding, the atypical behavior of the euler benchmark, the observed reductions in the Montecarlo are typical of the other benchmarks. (Overall, we observed a 43% reduction in total size of the entire benchmark suite from uncompressed Java classfiles to compressed and fully-optimized SafeTSA files.)

For the majority (10 out of 16) of classes making up the Montecarlo benchmark, the gzipped SafeTSA files are smaller than the gzipped Java class files. (This also holds across the entire Java Grande Forum Sequential Benchmark Suite, where for 67 of the 90 classes, the gzipped and fully-optimized SafeTSA files are smaller than the uncompressed Java classfile.) This also hold for the non-gzipped SafeTSA files, which are smaller the the corresponding Java classfiles for 9 of the 16 classes making up the Montecarlo benchmark. It is also interesting to note, that in general, when the SafeTSA files are smaller than the gzipped classfile, the non-gzipped SafeTSA files tend to also be smaller than the gzipped SafeTSA file. Whereas when the gzipped class file is smallest, the gzipped-SafeTSA file also tends be smaller than the non-gzipped SafeTSA file. This indicates that at least some of the redundancies being picked up by gzip are orthogonal to the constraints enforced by the SafeTSA encoder and suggests that if an even more compact representation is desired, probabilistic compression could be profitably integrated into SafeTSA's encoding at the expense of additional encoder and decoder complexity.

In summary, for every single class in the Java Grande Forum Sequential Benchmarks, the SafeTSA files are substantially smaller than uncompressed Java classfiles (by 43% overall). In addition, a majority of SafeTSA files are even smaller than the gzipped Java class files, but these tend to be the smaller files, and in aggregate SafeTSA files representing the Java Grande Forum Sequential benchmarks are slightly (less than 10%) larger than gzipped Java classfiles. Even so, SafeTSA are competitive with compressed Java classfiles.

7.2 Decoder Performance

While it is important to have small file sizes, these files need to decoded and executed on the fly, so the speed at which data can be extracted from the files is also important. In order to evaluate whether SafeTSA files can be decoded efficiently, we ran a prototype SafeTSA decoder on the encoded SafeTSA files representing each of the classes in Sections 1, 2, and 3 of Java Grande Sequential Benchmarks. In order to isolate just the decoding times, the decoder (which is written in Java) was extracted into a stand-alone module that was run from a benchmark that invoked it 100 times over each SafeTSA class file and used Java's System.currentTimeMillis() to time each execution, the fastest of the 100 executions was used in further analysis. This procedure should minimize the cost of just-in-time compilation, disk I/O, since by the last execution everything should in cache and the decoder's code should be adequately optimized by executing Java virtual machine.

The benchmark were measured on a Shuttle PC with a 2GHz Pentium M processor, 2MB of cache, a 1GB of RAM, under a Debian-compiled Linux 2.6.16 Kernel.

JGFArithBench	2648 bytes	7 ms
JGFAssignBench	3067 bytes	8 ms
JGFCreateBench	2661 bytes	11 ms
JGFMathBench	5730 bytes	36 ms
lufact.Linpack	1414 bytes	5 ms
euler.Tunnel	14836 bytes	89 ms
moldyn.md	2772 bytes	7 ms
montecarlo.AppDemo	2061 bytes	5 ms
montecarlo.MonteCarloPath	2723 bytes	7 ms
montecarlo.RatePath	2763 bytes	8 ms
raytracer.RayTracer	2401 bytes	8 ms
search.SearchGame	2055 bytes	5 ms

Table IV. The 12 Benchmarks with the Longest Decoding Times

The SafeTSA decoder was executed in single user mode using the HotSpot client JIT compiler and the incremental garbage collector (i.e., `-Xincgc`) in Sun's Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-b03).

Of the 88 classes analyzed, 41 of the classes required less than one millisecond to decode, and another 35 had decoding times of less than five milliseconds. These were excluded from further examination for two reasons: First, they are fast enough that in real life, hard disk seek times and network latencies are likely to dominate. Secondly, the standard Java timer facility (System.currentTimeMillis()) has a precision of only one millisecond, so the expected imprecision of a measurement of five milliseconds is 20%, and would be even greater for smaller measurements. It seems safe to conclude that on modern PC, for most Java classes, the decoding time is insignificant.

For comparison, we followed a similar procedure with a test harness that attempted to time verification of Java classfiles by Sun's JVM implementation by using a subclass of ClassLoader to invoke the `defineClass` and `resolveClass` methods in such a manner that the referenced classes would already be resolved, but the class under consideration would not be. The results were that after a couple iterations, the measured time went to less than one millisecond and thus was also insignificant.

Table IV lists the twelve Java Grande benchmark classes whose SafeTSA decoding times were at least five milliseconds along with their SafeTSA file sizes, and decoding times. JGFArithBench, JGFAssignBench, JGFCreateBench, and JGFMathBench are microbenchmarks from Section 1, Linpack is the main class of the Section 2 LUFact benchmark, and the remaining classes come from the Section 3 benchmarks: Euler, MolDyn, MonteCarlo, RayTracer, and Search. This includes all of the Java Grande benchmark classes whose SafeTSA file size exceeds two kilobytes plus Linpack, which is 1.4 kilobytes. All of these classes are much larger than the typical Java class.

Two of the benchmarks have long enough decoding times to be of a concern: the Tunnel class in the Section 3 Euler benchmark and the Section 1 JGFMathBench benchmark. As shown in Figure 12, these are also the slowest in terms of processing throughput. Since these are also the large classes, weighing in at over 5 and nearly 15 kilobytes for the SafeTSA file, the lower throughput reflects non-linear asymptotic performance. It should be noted, that the prototype is written in an

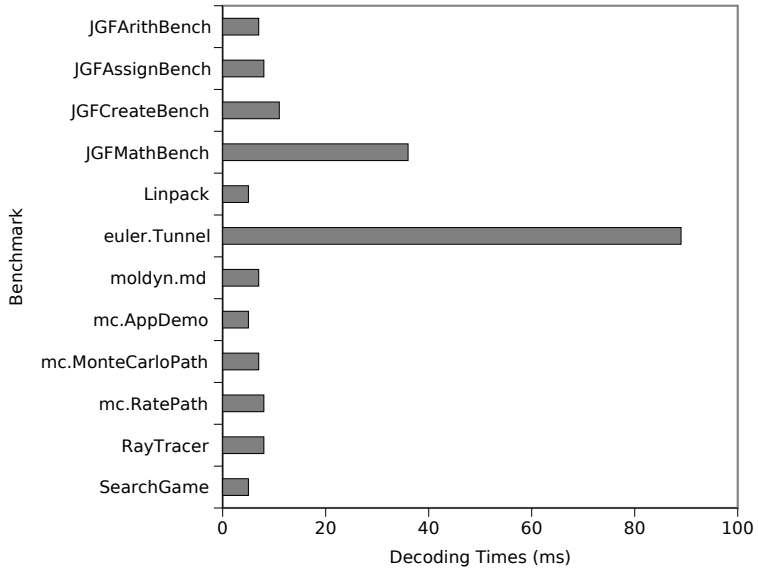

Fig. 11. Decoding Times for Select Benchmark Classes

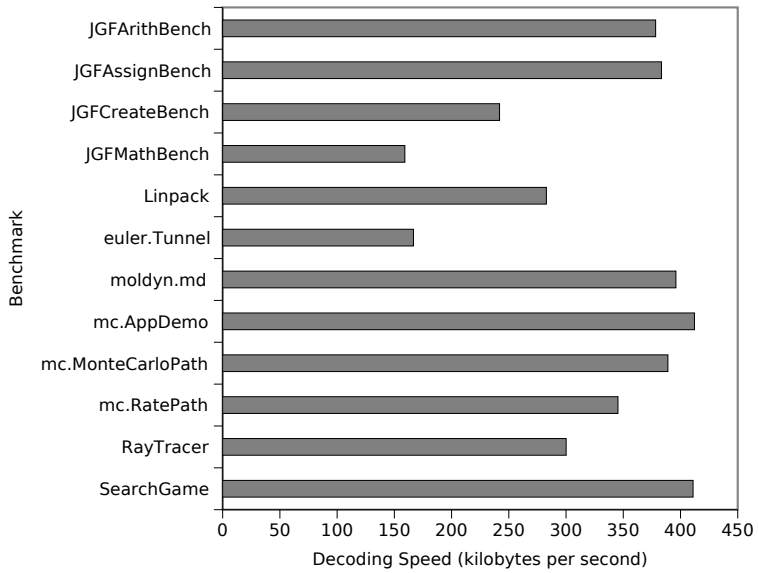

Fig. 12. Decoding Speeds for Select Benchmark Classes

object-oriented style in Java and profiling reveals that a substantial amount of the execution time is spent in garbage collection, which could be avoided with a table-driven C language implementation. But in any case, these are abnormally large Java classes. For typical class sizes, the SafeTSA decoding time is small enough as to be insignificant.

8. RELATED WORK

The Java language is normally compiled into Java bytecode (JVML), which is a stack-based language that is designed to be executed by a Java Virtual Machine (JVM) implementation, such as the one found in Sun's Java Platform Standard Edition. Prior to execution of JVML, the JVM bytecode must be verified using an iterative dataflow analysis and is either interpreted or translated from its stack-oriented form into register-based machine code. SafeTSA replaces JVML's stack-storage for temporaries and local variables with virtual registers in SSA Form.

Besides SafeTSA, the Low Level Virtual Machine (LLVM) [Lattner and Adve 2004] is the only persistent program representation based on SSA Form we are aware of. It differs from SafeTSA in that it is designed for use with a local compilation and optimization infrastructure rather than for transporting safe code. So unlike SafeTSA, it is not Java specific, does not retain source program control structures, and does not enforce a high-level type system that can be used for fine-grained language-based security.

Many optimizing compilers, including those found in Java Virtual Machine implementations, utilize SSA based intermediate representations. Intel Research's StarJIT Java Virtual Machine is particularly noteworthy for having recently introduced a type-system to verify the correctness null- and bounds-check optimizations [Menon et al. 2006]. Compared to SafeTSA's null-checked object reference and bounds-checked array element reference types, their approach is more general than SafeTSA's (which can only support redundancy eliminations) but results in a more complex type system. The StarJIT intermediate representation includes additional "proof values" as SSA operands and result values, and potentially unsafe instructions require "proof value" operands whose type matches the property to be proved. The complexity of such a type-checking such a system is dependent upon the complexity of the proofs: Menon et al. suggest that an integer linear programming tool would be sufficient to check the proofs needed to perform most common optimizations [Menon et al. 2006]. StarJIT, however, does not actually need to check the proofs, since they are produced internally and only used to convey dependencies between optimization phases. The array bounds check elimination in StarJIT is based on the ABCD algorithm [Bodík et al. 2000], which uses π functions to introduce additional names to track information learned from conditional branches.

Another internal compiler representation, λJVM [League et al. 2001], has some similarities to SafeTSA. Like SafeTSA, it is Java specific, and it uses A Normal Form (ANF) which is similar to SSA [Appel 1998]. As an internal representation, however, λJVM lacks a verifiable externalization.

Besides SafeTSA, Stork's Compressed Syntax Trees [Amme et al. 2001; Stork 2006] is the only other inherently safe program representation that we are aware of. Unlike SafeTSA, Compressed Syntax Trees is a general framework for compressing

(source) programs based on an extended grammar. This was inspired by our earlier work on encoding the syntax trees of Oberon programs using Slim Binaries [Franz and Kistler 1997; Kistler and Franz 1999].

It is common practice to combine multiple Java class files into a single Jar archive [JARSPEC 1999]. The Jar format is based on the Zip file format and uses Katz's DEFLATE algorithm [Deutsch 1996]. The Jax tool [Tip et al. 2002] operates on the set of byte code classes comprising an application and performs optimizations on it, primarily by removing unused methods, fields, and constant pool entries in libraries producing a smaller Jar archive for the program.

Horspool and Corless's clazz format [Horspool and Corless 1998] gets better compression by separating the class file into different sections and applies custom encodings (mostly based on start-step-stop codes) to each of several different data streams within the class file. They note that the structure of their encoding of branch targets, ensures that they always target the beginning of an instruction, so this would not need to be verified by the class loader; thus, clazz is inherently safe with respect to this limited property. The jazz format [Bradley et al. 1998] achieves greater compression than the clazz format by taking advantage of redundancies among different classes within the archive rather than compressing them individually. Pugh's format [Pugh 1999] gets even greater compression through a variety of techniques that also worked on entire archives of Java class files [Pugh 1999]. The compression gained by these techniques is achieved by identifying predictable patterns in the Java class files (especially in the symbolic linking information found in the constant pool). In contrast, the inherently-safe SafeTSA encoding ignores these patterns, and gains compactness only by eliminating illegal possibilities; it is possible to combine these two (c.f., [Stork 2006]).

The biggest commercial competitor to the Java language and platform is Microsoft's .NET platform with its Common Intermediate Language (CIL) [ECMA International 2002]. Like Java's bytecode CIL is a stack-based representation which needs to be verified using a dataflow analysis and translated into register based machine code by the .NET runtime. CIL, however, has some restrictions on the use of the stack across block boundaries that reduces the complexity of these analyses and transformations compared to Java byte code.

Evans and Fraser have also examined the design of interpretable instruction sets which are more space efficient than native instruction sets [Ernst et al. 1997; Fraser and Proebsting 1999; Evans and Fraser 2001]. Unlike SafeTSA, their work does not provide safety and is focused on decompression speed and interpretability.

9. SUMMARY

Conventional type-safe mobile code representations, such as the Java Classfile Format, use a stack-based bytecode which must be verified using a type-checking technique based on iterative dataflow analysis. In this paper, we have examined an alternative program representation, SafeTSA, which utilizes static single assignment form to create a just-in-time compiler-friendly program representation without sacrificing safety or compactness.

SafeTSA has several novel features that accomplish this end. The integrity and type-correctness of SafeTSA's SSA form instructions is maintained through *type*

separation (i.e., the segregation of values of different types into disjoint namespaces) and *dominator-based variable scoping* (which ensures that instructions may only reference local SSA variables that are guaranteed to be defined at that program point). SafeTSA also provides special types to facility the safe elimination of redundant null- and bounds-checks when the code is produced by the programmer, and retains high-level control structure information that would otherwise need to be rediscovered by the just-int-time compiler. In addition, SafeTSA employs an inherently safe encoding that maps binary prefix codes into valid SafeTSA programs; this reduces file size and enforces type-safety, alleviating the need for separate verification pass.

We have translated the classes found in the Sequential Java Grande Benchmarks into SafeTSA. Our experience indicates that SafeTSA file sizes are similar to compressed Java classfiles, and the time spent by our prototype SafeTSA decoder to process a typical Java class is similar to the average seek time of a typical hard disk.

SafeTSA is a compact compiler-friendly program representation that can be used as a drop-in replacement for Java bytecode in the Java runtime environment. It is hoped that the lessons learned in the development of SafeTSA will inspire future developments in virtual machine and intermediate representation design.

A. JAVA FEATURES NOT IMPLEMENTED IN SAFETSA

Although, the SafeTSA language, as implemented, can handle a large subset of the Java source language described in the Java Language Specification, Second Edition [Joy et al. 2000], there are several features that remain unimplemented. Although, some of these items involve substantial Engineering, there are not any unresolved fundamental issues that would prevent their incorporation into SafeTSA:

finally clauses. The handling of finally clauses discussed above is not implemented. Instead only try/catch statements without finally clauses are currently supported.

inner classes. SafeTSA does not include support for Java's nested inner classes.

synchronized keyword. SafeTSA does not include support for Java's synchronized classes, methods, and blocks.

B. PRIMITIVE FUNCTIONS

Primitive Functions and their Types

Function	*Description*	*Type*
bband	logical and	bool → bool → bool
bbeq	boolean equality	bool → bool → bool
bbnot	logical not	bool → bool
bbor	logical or	bool → bool → bool
bbneq	boolean not equal (xor)	bool → bool → bool
badd	bitwise and on a byte	byte → byte → byte
b2c	coerce byte to char	byte → char
b2d	coerce byte to double	byte → dbl

Function	*Description*	*Type*
b2f	coerce byte to float	byte → float
b2i	coerce byte to int	byte → int
b2l	coerce byte to long	byte → long
b2s	coerce byte to short	byte → char
bcompl	bitwise complement on a byte	byte → byte
bdec	decrement a byte (input left unchanged)	byte → byte
beq	equality between bytes	byte → bool → byte
bgt	greater than test between bytes	byte → byte → bool
bgte	greater-than or equal-to test	byte → byte → bool
binc	increment a byte (input left unchanged)	byte → byte
blt	less than test between bytes	byte → byte → bool
blte	less-than or equal-to test	byte → byte → bool
bmul	multiplication of two bytes	byte → byte → byte
bneg	arithmetic negation of a byte	byte → byte
bneq	not-equal test on two bytes	byte → byte → bool
bor	bitwise or on two bytes	byte → byte → byte
bshl	shift left of a byte	byte → int → byte
bshr	shift right of a byte (sign preserved)	byte → int → byte
bsub	subtraction of one byte from another	byte → byte → byte
bushl	alias for bshl	byte → int → byte
bushr	logical unsigned shift right of a byte	byte → int → byte
bxor	bitwise xor on bytes	byte → byte → byte
bdiv	division on bytes	byte → byte → byte
bmod	modulus on bytes	byte → byte → byte
sadd	add on shorts	char → char → char
sand	bitwise and on shorts	char → char → char
s2b	coerce short to byte	char → byte
s2c	coerce short to char	char → char
s2d	coerce short to double	char → dbl
s2f	coerce short to float	char → float
s2i	coerce short to int	char → int
s2l	coerce short to long	char → long
scompl	bitwise complement of a short	char → char
sdec	decrement a short (input unchanged)	char → char
seq	equality of two shorts	char → char → bool
sgt	greater than test beteween shorts	char → char → bool
sgte	greater-than or equal-to test	char → char → bool
sinc	increment a short (input unchanged)	char → char
slt	less than test between shorts	char → char → bool
slte	less-than or equal-to test	char → char → bool
smul	multiplication of shorts	char → char → char
sneg	arithmetic negation of a short	char → char
sneq	inequality of two shorts	char → char → bool
sor	bitwise or of two shorts	char → char → char
sshl	shift left of a short	char → int → char

Function	*Description*	*Type*
sshr	shift right of a short (sign preserved)	char → int → char
ssub	subtraction of one short from another	char → char → char
sushl	alias for sshl	char → int → char
sushr	logical unsigned shift right of a short	char → int → char
sxor	bitwise exclusive-or of two shorts	char → char → char
sdiv	divide a short by a short	char → char → char
smod	one short modulo another	char → char → char
cadd	character addition	char → char → char
cand	bitwise and of characters	char → char → char
c2b	coerce character to byte	char → byte
c2d	coerce character to doulbe	char → dbl
c2f	coerce character to float	char → float
c2i	coerce character to integer	char → int
c2l	coerce character to long	char → long
c2s	coerce character to short	char → char
ccompl	bitwise-complement of a character	char → char
cdec	decrement a character (input unchanged)	char → char
ceq	equality test on characters	char → char → bool
cgt	greater than test on characters	char → char → bool
cgte	greater-than or equalt-to test on characters	char → char → bool
cinc	inrement of a character (input unchanged)	char → char
clt	less than test on characters	char → char → bool
clte	less-than or equal-to test on characters	char → char → bool
cmul	multiplication of characters	char → char → char
cneg	arithmetic negation of a character	char → char
cneq	not-equal test between characters	char → char → bool
cor	bitwise or of characters	char → char → char
cshl	shift left of a character	char → int → char
cshr	shift right of a character (sign preserved)	char → int → char
csub	subtraction of one character from another	char → char → char
cushl	alias for cshl	char → int → char
cushr	logical unsigned shift right	char → int → char
cxor	exclusive or of two chracters	char → char → char
cdiv	division of one character by another	char → char → char
cmod	one character modulo another	char → char → char
iadd	integer addition	int → int → int
iand	bitwise and on integers	int → int → int
i2b	coerce an integer to a byte	int → byte
i2c	coerce an integer to a character	int → char
i2d	coerce an integer to a double	int → dbl
i2f	coerce an integer to a float	int → float
i2l	coerce an integer to a long	int → long
i2s	coerce an integer to a short	int → char
icompl	bitwise complement of an integer	int → int
idec	decrement an integer (input unchanged)	int → int

Function	*Description*	*Type*
ieq	equality test on integers	int → int → bool
igt	greater than test on integers	int → int → bool
igte	greater-than or equal-to test on integers	int → int → bool
iinc	increment an integer (input unchanged)	int → int
ilt	less than test on integers	int → int → bool
ilte	less-than or equal-to test on integers	int → int → bool
imul	integer multiplication	int → int → int
ineg	arithmetc negation of an integer	int → int
ineq	not-equal test on integers	int → int → bool
ior	integer bitwise or	int → int → int
ishl	integer shift left	int → int → int
ishr	integer shift right (sign preserved)	int → int → int
isub	integer subtraction	int → int → int
iushl	alias for ishl	int → int → int
iushr	logical unsigned integer shift right	int → int → int
ixor	integer exclusive or	int → int → int
idiv	integer divide	int → int → int
imod	modulo on integers	int → int → int
ladd	addition of two longs	long → long → long
land	bitwise and of two longs	long → long → long
l2b	coerce a long to a byte	long → byte
l2c	coerce a long to a char	long → char
l2d	coerce a long to a double	long → dbl
l2f	coerce a long to a float	long → float
l2i	coerce a long to an integer	long → int
l2s	coerce a long to a short	long → char
lcompl	bitwise complement of a long	long → long
ldec	decrement of a long (input unchanged)	long → long
leq	equality test on longs	long → long → bool
lgt	greater than test on longs	long → long → bool
lgte	greater-than or equal-to test on longs	long → long → bool
linc	increment a long (input unchanged)	long → long
llt	less than test on longs	long → long → bool
llte	less-that or equal-to test on longs	long → long → bool
lmul	long multiplication	long → long → long
lneg	arithmetic negation of a long	long → long
lneq	not-equal test on longs	long → long → bool
lor	bitwise or of two longs	long → long → long
lshl	long shift less	long → int → long
lshr	long shift right (sign preserved)	long → int → long
lsub	long substraction	long → long → long
lushl	alais for lshsl	long → int → long
lushr	logical unsigned long shift right	long → int → long
lxor	bitwise xor of longs	long → long → long
ldiv	division of a long by along	long → long → long

Function	*Description*	*Type*
lmod	a long modulo a long	long → long → long
fadd	addition of floats	float → float → float
f2b	coerce float to byte	float → byte
f2c	coerce float to character	float → char
f2d	coerce float to double	float → dbl
f2i	coerce float to integer	float → int
f2l	coerce float to long	float → long
f2s	coerce float to short	float → char
fdec	decrement a float (input unchanged)	float → float
fdiv	float division	float → float → float
feq	equality test on floats	float → float → bool
fgt	greater than test on floats	float → float → bool
fgte	greater-than or equal-to test on floats	float → float → bool
finc	increment a float (input unchanged)	float → float
flt	less than test on floats	float → float → bool
flte	less-than or equal-to test on floats	float → float → bool
fmod	a float modulo a float	float → float → float
fmul	float multiplication	float → float → float
fneg	arithmetic negation of a float	float → float
fneq	not-equal test on a float	float → float → bool
fsub	subtraction of a float from a float	float → float → float
dadd	addtion on doubles	dbl → dbl → dbl
d2b	coerce a double to a byte	dbl → byte
d2c	coerce a double to a character	dbl → char
d2f	coerce a double to a float	dbl → float
d2i	coerce a double to an integer	dbl → int
d2l	coerce a double to a long	dbl → long
d2s	coerce a double to a short	dbl → char
ddec	decrement a double (input unchanged)	dbl → dbl
ddiv	divide a double by a double	dbl → dbl → dbl
deq	equlaity test on doubles	dbl → dbl → bool
dgt	greater that test on doubles	dbl → dbl → bool
dgte	greater-than or equal-to test on doubles	dbl → dbl → bool
dinc	increment a double (input unchanged)	dbl → dbl
dlt	less-than test on a double	dbl → dbl → bool
dlte	less-than or equal-to test on a double	dbl → dbl → bool
dmod	a double modulo a double	dbl → dbl → dbl
dmul	product of two doubles	dbl → dbl → dbl
dneg	arithmetic negation of a double	dbl → dbl
dneq	not-equal test on doubles	dbl → dbl → bool
dsub	one double subtracted from another	dbl → dbl → dbl

ACKNOWLEDGMENT

We'd to thank Cristian Petrescu-Prahova who helped us recognize the connection between referential integrity and a stack discipline during pre-order traversal of the

dominator tree. We would also like to acknowledge everyone who has contributed to the SafeTSA project in some way, especially Philipp Adler, Alexander Apel, Hartmut Arlt, Niall Dalton, Andreas Finn, Andreas Hartmann, Thomas Heinze, Tino Mai, Marc-André Möller, Jan Peterson, Prashant Saraswat, and Ning Wang.

This investigation has been supported in part by the Deutsche Forschungsgemeinschaft (DFG) under grants AM-150/1-1 and AM-150/1-3, by the National Science Foundation (NSF) under grant CCR-9901689, and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-99-1-0536.

REFERENCES

Amme, W., Dalton, N., Fröhlich, P., Haldar, V., Housel, P. S., von Ronne, J., Stork, C. H., Zhenochin, S., and Franz, M. 2001. Project transprose: Reconciling mobile-code security with execution efficiency. In *Proceedings of the Second DARPA Information Survivability Conference and Exposition*. IEEE Computer Society, Los Alamitos, California, 196–210.

Amme, W., Dalton, N., von Ronne, J., and Franz, M. 2001. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the SIGPLAN'01 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 137–147.

Appel, A. W. 1998. Ssa is functional programming. *SIGPLAN Not. 33,* 4, 17–20.

Bodík, R., Gupta, R., and Sarkar, V. 2000. Abcd: eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 321–333.

Bradley, Q., Horspool, R. N., and Vitek, J. 1998. Jazz: an efficient compressed format for java archive files. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, Indianapolis, Indiana, 7.

Breazu-Tannen, V., Coquand, T., Gunter, C. A., and Scedov, A. 1991. Inheritance as implicit coercion. *Information and Computation 93*, 172–221.

Choi, J.-D., Cytron, R., and Ferrante, J. 1991. Automatic construction of sparse data flow evaluation graphs. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 55–66.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13,* 4, 451–490.

Deutsch, P. 1996. Deflate compressed data format specification version 1.3. RFC 1951, The Internet Engineering Task Force.

ECMA International 2002. *Standard ECMA-335: Common Language Infrastructure*, 2nd ed. ECMA International.

Ernst, J., Evans, W., Fraser, C. W., Proebsting, T. A., and Lucco, S. 1997. Code compression. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 358–365.

Evans, W. S. and Fraser, C. W. 2001. Bytecode compression via profiled grammar rewriting. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 148–155.

Franz, M. and Kistler, T. 1997. Slim Binaries. *Communications of the ACM 40,* 12 (Dec.), 87–94.

Fraser, C. W. and Proebsting, T. A. 1999. Finite-static code generation. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 270–280.

GCC 2005. Gnu compiler collection (gcc) internals. http://gcc.gnu.org/onlinedocs/gccint/.

Horspool, R. N. and Corless, J. 1998. Tailored compression of java class files. *Software: Practice and Experience 28,* 12, 1253–1268.

JARSPEC 1999. Jdk 1.3 — jar file specification.

Joy, B., Steele, G., Gosling, J., and Bracha, G. 2000. *Java Language Specification*, 2 ed. Addison-Wesley Professional, Boston, MA, USA.

Kistler, T. and Franz, M. 1999. A Tree-Based alternative to Java byte-codes. *International Journal of Parallel Programming 27,* 1 (Feb.), 21–34.

Lattner, C. and Adve, V. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Press, Palo Alto, California.

League, C., Trifonov, V., and Shao, Z. 2001. Functional Java bytecode. In *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics*. International Institute of Informatics and Systemics, Orlando, FL.

Lengauer, T. and Tarjan, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Prog. Lang. and Sys. 1,* 1 (July), 121–141.

loup Gailly, J. 2002. *Gzip Manpage*. Free Software Foundation, Boston.

Menon, V. S., Glew, N., Murphy, B. R., McCreight, A., Shpeisman, T., Adl-Tabatabai, A.-R., and Petersen, L. 2006. 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*. ACM Press, New York, NY, USA, 397–408.

Pugh, W. 1999. Compressing java class files. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 247–258.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1988. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 12–27.

Stork, C. H. 2006. Compressed abstract trees and their applications. Ph.D. thesis, University of California, Irvine.

Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., and Streeter, D. 2002. Practical extraction techniques for java. *ACM Trans. Program. Lang. Syst. 24,* 6, 625–666.

Wegman, M. N. and Zadeck, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13,* 2, 181–210.

Xi, H. and Pfenning, F. 1998. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 249–257.