A Certifying Compiler for Java

Christopher Colby Peter Lee George C. Necula*
Fred Blau Mark Plesko Kenneth Cline
Cedilla Systems Incorporated
4616 Henry Street
Pittsburgh, Pennsylvania 15213
Hackers@CedillaSystems.com

Abstract

This paper presents the initial results of a project to determine if the techniques of *proof-carrying code* and *certifying compilers* can be applied to programming languages of realistic size and complexity. The experiment shows that: (1) it is possible to implement a certifying native-code compiler for a large subset of the Java programming language; (2) the compiler is freely able to apply many standard local and global optimizations; and (3) the PCC binaries it produces are of reasonable size and can be rapidly checked for type safety by a small proof-checker. This paper also presents further evidence that PCC provides several advantages for compiler development. In particular, generating proofs of the target code helps to identify compiler bugs, many of which would have been difficult to discover by testing.

1 Introduction

In earlier work, Necula and Lee developed *proof-carrying code* (PCC) [11, 13], which is a mechanism for ensuring the safe behavior of programs. In PCC, a program contains both the code and an encoding of an easy-to-check proof. The validity of the proof, which can be automatically determined by a simple proof-checking program, implies that the code, when executed, will behave safely according to a user-supplied formal definition of safe behavior. Later, Necula and Lee demonstrated the concept of a *certifying compiler* [14, 15]. Certifying compilers promise to make PCC more practical by compiling high-level source programs into optimized PCC binaries completely automatically, as opposed to depending on semi-automatic theorem-proving techniques. Taken together, PCC and certifying compilers provide a possible solution to the code safety problem, even in applications involving mobile code [12].

In this paper, we present the first results from a project to determine if PCC and certifying compilers can be applied to programming languages of realistic size and complexity. We show that: (1) it is possible to implement a certifying native-code compiler for a language that has objects and classes, user-defined exceptions and exception handling, and floating-point arithmetic; (2) the compiler is freely able to apply many standard local and global optimizations; and (3) the PCC binaries it produces are of reasonable size and can be rapidly checked by a small proof-checker.

In this paper, we support these claims by presenting some design and implementation details of an optimizing compiler called Special J that compiles Java bytecode [7] into target code for the Intel x86 architecture [5]. While space limitations prevent us from giving a thorough account of the design and implementation of Special J, we can illustrate the main features and techniques of the system through the use of a running example, focusing mainly on advanced language features such as objects, exceptions, and floating-point arithmetic. In particular, we hope to highlight the fact that the compiler and the target code produced by it are largely conventional, except for a small number of assembly language annotations that are used by the proof-generation and proof-checking infrastructure.

After a review of some of the background for this work in Section 2, we present in Section 3 a small Java program that is the basis for the examples that run throughout the rest of the paper. The example, though small, is not a "toy" in the sense that it makes use of some of the advanced features of Java. With this example, we can discuss the main phases of certifying compilation (Section 3), verification condition generation (Section 4) and finally proving (Section 5), with an emphasis on the annotations produced by the compiler and the checking obligations that are entailed.

Throughout the development of Special J, we encountered many situations where PCC helped us identify and localize bugs in the compiler. Many of these bugs would have been extremely difficult to discover by standard testing techniques. We believe that this has saved us months of development time. On the other hand the fact that the compiler must insert special annotations into the target code introduces new opportunities for bugs. We try to give some feel for this development process in Section 6. Finally, we conclude with our plans for future work and some thoughts on the prospects for a practical PCC system.

*George Necula's current address is Computer Science Division, University of California, Berkeley, 783 Soda Hall, Berkeley, CA 94720.

†This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "Proof-Carrying Code for Information Assurance", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-98-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

To appear in the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00).

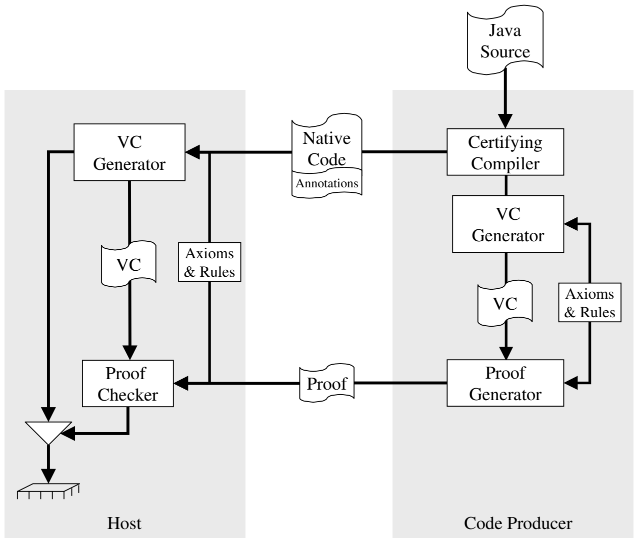

Figure 1: The architecture of our PCC implementation. The code producer uses a certifying compiler to generate annotated native code. A verification condition (VC) is then derived from this and then proved by the proof generator. The annotated code and proof together constitute the PCC binary which is transmitted to the host.

2 Background

The purpose of *proof-carrying code* is to make it possible for a host system to determine if a program will behave safely, prior to installing and executing it. This is accomplished by requiring that the program's producer provide evidence, in some easy-to-verify form, that the program is well-behaved. As the name implies, this evidence often takes the form of a mathematical proof of a safety property, although other forms of evidence are also possible. The potential engineering advantage of PCC derives from the fact that it is usually easy to check a proof of a program even if generating it is difficult. Thus, the hard work of determining safety is shifted from the program's consumer to its producer.

As a practical matter, the program's producer usually has more information available for reasoning about the program's safety, and thus can obtain a proof more easily than can the host. For example, suppose programs are transmitted from the producer to the host in native-code form. Suppose further that the producer generates native code by writing programs in a type-safe language such as ML or Java. The producer then knows that, barring any bugs in the compiler, the generated target code for each source program is also type-safe. Of course, compilers are never bug-free. But by arranging for the compiler to attach enough evidence (in the form of a proof) that each compilation it carries out did in fact preserve type safety, any host that receives the target code can examine this evidence to satisfy itself that the code is in fact type safe. Following Necula and Lee, we refer to this concept of a compiler that automatically generates proof-carrying code as a *certifying compiler* [15].

Figure 1 shows the overall architecture of our implementation of PCC. On the left side of the figure we see the host's part of the process, which begins when a native-code binary and its proof are received from the code producer. In order to ensure that the given proof is in fact a safety proof for the given program, the exact predicate to be proved is derived via a one-pass inspection of the program. The resulting safety predicate has the property that its logical validity implies that the code behaves safely. In the terminology of program verification, the safety predicate is referred to as the *verification condition*, or simply VC, and the process of deriving it from the code is called *VC generation* [2].

We will say more about the nature of the VC's in the next section. However, it is important to point out that a number of annotations are provided with the native code. Some of these annotations are purely optional and serve only to simplify VC generation. On the other hand, some annotations, such as loop invariants, are required in order to make automatic VC generation possible at all. The examples of the next section will illustrate both kinds of annotations.

Once a VC is obtained, the proof can then be checked to see if it is in fact a valid proof of the VC. The proofs are written in a logical proof system that is defined by a collection of proof rules. The precise notion of safety that is enforced by the system is thus defined by the combination of the VC generation process and the logical proof rules that govern what can be written in the predicates and proofs.

The proof rules, and the proofs themselves, are represented in a variant of the Edinburgh Logical Framework (LF) [3]. In LF, the inference rules of the proof system are defined as an LF signature. Type checking the LF representation of a proof is then sufficient for proof checking. LF is a small language which can be efficiently type checked with a small program. Furthermore, Necula and Lee showed how some type information can be elided from the LF terms and then reconstructed during type checking [16], thereby greatly reducing the size of the encoded proofs.

The right side of Figure 1 shows the code producer's part of the process. A *certifying compiler* is used to take a source program with known safety properties (typically derived via type checking) and then compile it into annotated native code. In order to generate an appropriate proof, the producer derives the same VC that the host will derive, and then submits the VC to a *proof generator*. The proof generator then constructs the LF representation of the proof in the logical system defined by the proof rules.

As we will see in the examples later in the paper, the annotations deposited by the compiler into the native code allow VC's of a predictable form to be generated. This fact is exploited by the proof generator so that it can be guaranteed to find a proof automatically for any correct output produced by the compiler. Hence, the combination of the certifying compiler, VC generator, and proof generator can be viewed by the programmer as a "black box," with essentially the same functionality as a conventional compiler, except, of course, that the target programs carry proofs.

3 Compiling Java into Annotated Binaries

Our compiler accepts class files containing Java bytecodes and produces annotated x86 assembly language. In the front-end of the compiler the program is analyzed to determine the class hierarchy and the layout of the objects. We use a standard object layout with each object containing a pointer to a descriptor of the class to which it belongs. Following this pointer the object contains space for storing the per-instance fields, starting with the fields of the most distant ancestor and ending with the fields added by the class to which the object belongs. The class descriptor table (CDT) contains the same meta information that the class file contains, such as the number and types of fields and the

```
/* Polynomial objects. */
class Poly {
  Poly(float[] coefficients) { ... }

  /* Evaluate the polynomial at x */
  float eval(float x) {
    float term = 1.0f;
    float result = 0.0f;
    for (int i=0; i<coefficients.length; i++) {
        result += coefficients[i] * term;
        term *= x;
    }
    return result;
  }

  private float[] coefficients;
/* Exception in case no root is found */
class NotFound extends Exception { ... }
```

```
/* Polynomial root-finding application */
public class Root {
  /* Find the root of the given polynomial */
  static float find(Poly p, float precision,
                    float x, float dx)
  throws NotFound { ... }

  /* Main method for the application */
  public static float example() {
    float root = 0.0f;
    float[] coefficients = {1.0f, 1.0f};
    Poly p = new Poly(coefficients);
    try {
      root = Root.find(p, 0.0001f, 0.0f, 0.01f);
    }
    catch (NotFound exn) { root = exn.val; }
    finally { return root; }
} }
```

Figure 2: Excerpts of the source code for the sample polynomial root-finding application.

```
_eval__4root4PolyF:
     pushl %ebp
     movl  %esp, %ebx
     movl  8(%ebp), %ebx     ; get this
     movl  4(%ebx), %eax     ; get this.coefficients
     testl %eax, %eax        ; null check
     je    L43
L42: movl  4(%eax), %edx     ; get coefficients.length
     testl %edx, %edx        ; if 0 then skip loop
     jle   L47
L46: flds  LC1               ; initialize term
     fldz                    ; initialize result
     xorl  %edx, %edx        ; initialize i
     movl  4(%eax), %ecx     ; get coefficients.length
     jmp   L31
L44: fxch  %st(1)            ; result on top of FPU
L31: ANN_LOOP(
       INV = {(lt edx (sel4 rm (add eax 4))),
              (ge edx 0),
              (type f7 jfloat),
              (type f6 jfloat)},
       MODREG = {edi,edx,eflags,esi,
                 f0,f1,f2,f3,f4,f5,f6,f7,fflags})
L38: flds  8(%eax, %edx, 4)  ; load coefficients[i]
     fmul  %st(2), %st(0)    ;  * term
     faddp                   ;  + result
     fxch  %st(1)            ; term on top of FPU
     fmuls 12(%ebp)          ;  * x
L36: incl  %edx              ; i++
     cmpl  %ecx, %edx        ; i < coefficients.length?
     jl    L44               ;  loop back if yes
     jmp   L32
L47: fldz                    ; if skipped loop,
     fldz                    ;  result = 0.0
L32: fxch  %st(1)            ; result on top of FPU
     fstp  %st(1)            ; remove term
     movl  %ebp, %esp        ; return result
     popl  %ebp
     ret
L43: call  __Jv_ThrowNullPointer
ANN_UNREACHABLE
     nop
```

Figure 3: Target code for the `Poly.eval` method.

```
_example__4root4Root:
   ...                       ; create float array (elided)
     call  __Jv_newFloatArray
   ...                       ; initialize array (elided)
     call  __Jv_bcopy
   ...                       ; create Poly object (elided)
     call  ___4root4PolyAF
   ...                       ; enter try block
                             ; install catch-all
INSTALL_1HANDLER(__CL_4java4lang9Throwable,L51)
     movl  -8(%ebp), %eax
     movl  %eax, -16(%ebp)   ; spill p
     movl  -4(%ebp), %ecx
     movl  %ecx, -12(%ebp)   ; spill coefficients
     movl  $0, -20(%ebp)     ; initialize and spill root
                             ; install catch handler
INSTALL_1HANDLER(__CL_4root8NotFound,L52)
     pushl LC3               ; arg 0.01f
     pushl LC0               ; arg 0.0f
     pushl LC4               ; arg 0.0001f
     pushl -16(%ebp)         ; arg p
     call  _find__4root4RootL4root4PolyXFFF ; call find
     addl  $16, %esp
     fstps -20(%ebp)         ; store result in root
UNINSTALL_HANDLERS(2)        ; uninstall catch & catch-all
     nop
L56: flds  -20(%ebp)         ; finally block
     movl  %ebp, %esp
     popl  %ebp
     ret
L52: testl %eax, %eax        ; catch handler block
     je    L58               ; null check on exn
L57: movl  4(%eax), %ebx     ; get exn.val
     movl  %ebx, -20(%ebp)   ; store into root
UNINSTALL_HANDLERS(1)        ; uninstall catch-all
     jmp   L56               ; go to finally block
L51: flds  -20(%ebp)         ; catch-all handler block
     movl  %ebp, %esp
     popl  %ebp
     ret
L58: call  __Jv_ThrowNullPointer
ANN_UNREACHABLE
     nop
```

Figure 4: Target code for the `Root.example` method.

number and signatures of methods. The CDT also contains a pointer to the virtual method table (VMT).

In addition to these tables the compiler generates annotated assembly language. Except for the annotations the output of the compiler is similar to that of conventional compilers. In order to describe the annotations and a few more details regarding the compilation strategy we will use a sample Java program whose code is shown in Figure 2. Three classes are defined. Class `Poly` implements polynomials, which are represented as an array of coefficients. The method `eval` evaluates a polynomial at a given value. Class `Root` provides the main method, `example`, which creates a polynomial object and then invokes the method `find` to search for a root of the polynomial. If no root is found, then the `NotFound` exception is thrown.

The result of certifying compilation of the methods `eval` and `example` is shown in Figures 3 and 4, respectively. These figures, which use the Gnu syntax for x86 assembly code, show that the output of certifying compilation is largely conventional, except for annotations such as `ANN_LOOP` and `ANN_UNREACHABLE`. The names of the external symbols have been "mangled" so that each name includes package, class, and type information to aid in resolving overloaded symbols. Also, to simplify the presentation, the target code for `example` uses two macros, which are defined in Figure 5.

In the code for `eval`, the `ANN_LOOP` annotation provides a loop invariant that states assumptions on the live registers and stack slots that are modified in the loop that starts at label `L31`. Each loop invariant must declare a set of conditions that are claimed to hold every time the execution reaches that point in the program. Also, to simplify the job of the VC generator, the loop invariant annotation must also declare the set of registers that might change their value in between successive executions of the loop.

One typical invariant condition is a typing declaration for the contents of a machine register. For example, the last two conditions in the loop invariant state that FPU registers `f6` and `f7` (addressed by operands `%st(0)` and `%st(1)` at loop entry) contain valid concrete representations of the Java `float` type (i.e., valid IEEE single-precision numbers). In the absence of optimizations, a compiler need only generate these kinds of conditions; this is simple to achieve.

For the purposes of our presentation, we have taken the output of the Special J compiler for the `eval` method and applied further global optimizations by hand. Specifically, we have hoisted and eliminated several null-pointer and array-bounds checks. (The code we show for `example`, on the other hand, is exactly the output produced by Special J.) To support the certification stage in the presence of such optimizations, we have also added the first two invariant conditions shown in the invariant at label `L31`. They state that the `edx` register is greater than or equal to 0 and less than the contents of the integer stored at address $\texttt{eax} + 4$, which is the length of the `coefficients` array. The functions `ge` and `lt` are x86 signed comparison operators and will be discussed further in Section 5. We will say more about loop invariants in general in Section 4, and about this loop invariant in particular in Section 5.

To compile exceptions we use a setjmp/longjmp scheme. Even though this scheme penalizes programs that execute `try` statements but do not raise exceptions, we preferred it to the table-based scheme used by other Java compilers (and supported by the Java Virtual Machine) because the tables consume space even in code that does not use them.

The `example` method shows a typical code sequence for a try-catch-finally block. The `INSTALL_1HANDLER` macro (Figure 5) shows that an exception handler is a 5-word structure, stored on the run-time stack. Annotations indicate which immediate values are to be interpreted as symbolic addresses, and signal to the VC generator the point at which the installation of an exception handler has been completed. The `UNINSTALL_HANDLERS` macro is used to pop a given number of handlers off the stack. User exceptions are thrown by a run-time system routine called `__Jv_Throw` (not shown here), which uninstalls all of the handlers up to and including the one to which it throws. The `UNINSTALL_HANDLERS` macro is then used to uninstall any remaining handlers prior to exiting the try block. An exception throw restores `ebp` to the value stored in the handler and restores `esp` to its value before the handler was pushed onto the stack.

```
; Build and install a new exception handler
macro INSTALL_1HANDLER(Class,Block) =
    call  __Jv_GetExcHandler  ; Get current handler cell
 ANN_SYMBOLADDR
    pushl $Block              ; Code for the handler
 ANN_SYMBOLADDR
    pushl $Class              ; Exception class
    pushl %ebp                ; Current frame pointer
    pushl $1                  ; Number of catch blocks
    pushl (%eax)              ; Link to old handler
    movl  %esp, (%eax)        ; Install new handler
 ANN_INSTALLEDJAVAHANDLER(Block)

; Uninstall the last Num exception handlers
macro UNINSTALL_HANDLERS(Num) =
    call  __Jv_GetExcHandler  ; Get current handler cell
    movl  <20*(Num-1)>(%esp), %ebx ; Get old handler
    addl  $<20*Num>, %esp     ; Remove handlers
    movl  %ebx, (%eax)        ; Restore old handler
 ANN_UNINSTALLEDJAVAHANDLER(Num)
```

Figure 5: The exception-handler macros.

While `example` does not have any loops (and thus no `ANN_LOOP` annotations), the use of exception handlers leads to complex control flow. In Section 4.4, we will describe how annotations such as `ANN_INSTALLEDJAVAHANDLER` allow the VC generator to account for the many possible control paths in this method.

4 Verification Condition Generation

The design of the verification condition generator is largely conventional, as described for example in [14]. For the benefit of those readers not familiar with verification condition generation for unstructured languages we will briefly review below the major issues. What differentiates our implementation of VC generation is its use of annotations to understand the object layout (Section 4.3) and the control flow in the presence of exceptions (Section 4.4).

4.1 A Glossary of Predicate Constructors

First we describe the meaning of several predicate constructors that we use in our system to express both proof obligations and assumptions about the object layout and the class hierarchy. The following are expression constructors:

- `jfloat`, `jint` and others are used to denote symbolically the set of valid representations for Java types `float` and `int`.

- `(jarray` T`)` denotes the type of Java array objects whose elements have type T.
- `(jinstof` C`)` denotes the type of objects that are compatible with the class C. Classes and interfaces are denoted in assertions using the symbol name of their class descriptor table.
- `(ptr` T`)` denotes the type of pointers to values of type T. This does not correspond to a Java type.
- `(add` E E'`)` denotes the 32-bit signed addition of E and E'. There are similar operators for other integer and floating-point arithmetic operations.
- `(sel4` M A`)` denotes the contents of address A in memory M.
- `(upd4` M A E`)` denotes a memory like M but in which address A has been bound to the value of E.

The following are predicate constructors:

- `(size` T B`)` means that the size of a value of type T is B bytes.
- `(type` E T`)` means that expression E denotes a value that is a concrete representation for the type denoted by T.
- `(lt` E E'`)` means that E is less than E' using 32-bit signed comparison. There are other signed comparison operations such as `le`, `gt`, etc.
- `(ult` E E'`)` means that E is less than E' using 32-bit unsigned comparison. There are other unsigned comparison operations such as `ule`, `ugt`, etc.
- `(saferd4` E`)` means that it is safe to read four bytes starting at the address denoted by E.
- `(nonnull` E`)` means that E is not null.
- `(jextends` C D`)` means that class C extends class D.
- `(jimplements` C I`)` means that class C implements interface I.
- `(vmethod` C O S`)` means that a method with signature S is at byte-offset O in class C's virtual method table.
- `(ifield` C O T`)` means that an instance field of type T is at byte-offset O in an object of class C.

The VC generator knows the meaning of some of these constructors and it uses them to express the semantics of x86 machine instructions. Examples of such constructors are `add`, `sel4` and `lt`. But for the majority of constructors the VC generator does not try to interpret them. Instead, it is up to the prover and the proof checker to do this interpretation. For this purpose their meaning is expressed in terms of logical rules of inference, as will be shown in Section 5.

4.2 Overview of VC Generation

VC generation works with respect to a particular safety policy. VC generation examines the code and the meta-data (e.g., class descriptors and virtual method tables) and checks a variety of simple syntactic conditions such as that there are no jumps outside the code segment. When the VC generator encounters operations that could violate the safety policy (such as memory operations) it produces proof obligations that, if satisfied, guarantee the safety of the operation.

To examine a single control path, the VC generator scans the sequence of machine instructions from beginning to end. As it scans the path, it keeps track of (A) a set of logical predicates that are known to be true at this point in the scan, and (B) symbolic values representing the contents of each *register*. In our implementation, the registers are the Pentium machine registers including the floating-point registers (named `f0`,...,`f7`), the condition-code registers (`eflags` and `fflags`), the stack slots of the current frame (named `loc1`,...,`loc`n with `loc1` being the name of the slot holding the first incoming argument and `loc2` that of the second incoming argument or of the return address if only one argument has been pushed by the caller), and a single pseudo-register named `rm` representing the contents of all other memory locations.

During its scan of the code, the VC generator performs the following steps for each machine instruction:

1. If there are safety restrictions on the instruction (e.g., if it accesses memory), use the current values of (B) to output a proof obligation representing a sufficient condition to establish safety for this instruction. This proof obligation is expressed as a predicate, and must be proved under the current assumptions in (A).

   For example, if the VC generator encounters a read instruction `movl 4(%eax), %ebx` it will create the proof obligation `(saferd4 (add` eax `4))`, where eax is the current symbolic contents of register `eax` and `add` denotes the 32-bit two's-complement addition as performed by the x86 processor. This proof obligation can be satisfied when the address `(add` eax `4)` can be proved readable under the current assumptions in (A). (See Section 4.1.)

   The VC generator recognizes certain patterns of memory addresses as stack addresses and thus a similar read instruction `movl -20(%ebp), %ebx` would be trated as a simple move instruction from pseudo-register `loc`i to `ebx` for some integer i. The value of i depends on the number of arguments in the stack frame (see above).

2. If the instruction changes the value of any registers, update (B) using the current values of (B).

   For example, in the case of the memory-read instruction from above the symbolic value of the register `ebx` becomes `(sel4` rm `(add` eax `4))`, where rm is the current symbolic value of the `rm` register. The `sel4` operator is used to construct a symbolic expression denoting the 32-bit word contained, in a given memory state, at a given address. (See Section 4.1.)

3. If necessary, update (A) using the current values of (B). For instance, following a conditional branch the VC generator will generate a new predicate (the condition for that arm) that will be used as an assumption for the remainder of the path.

For example, the code directly following the sequence of instructions:

```
cmpl %eax, %ebx
jl Label
```

will be processed in the scope of an additional assumption (`ge` *ebx eax*), where *eax* and *ebx* are the current symbolic contents of the registers `eax` and `ebx` respectively. Here `ge` denotes the result of the comparison performed by the x86 conditional branch instruction `jge` (i.e., when the `jl` branch is not taken).

Above we have seen how the VC generator scans a single control path. Essentially, it is encoding the operational semantics of the machine architecture (Pentium in this case). But VC generation must have a way to handle situations in which there are a large or infinite number of distinct paths (e.g., cascaded if-then-else structures, loops with exits), and in which paths have non-local control (e.g., method calls). The solution to all of these cases is fundamentally the same: break the paths with logical invariants at appropriate locations.

When the VC generator encounters in the code an annotation of the form

`ANN_LOOP(INV=`P`, MODREG=`R`)`

it knows that it is at the entry point of a loop, that the set of predicates P are invariants of the loop, and that only registers mentioned in the set R can be modified around the loop. When a scan of a control path hits an annotation of the above form, it performs a modified form of the three VC-generation steps described near the beginning of this section:

1. A proof obligation is generated by substituting the current register values into P. This ensures that this particular control path indeed establishes the initial invariant condition.

2. For each register $r \in R$, the symbolic value of r is set to a fresh copy of r. These values are given names like `eax_3` if this is the third fresh copy of register `eax`. For all other registers r' that do not change in the loop, the scan remembers the current symbolic value of r' in order to verify later that r' was indeed left unmodified by the loop.

3. An assumption is generated for the remainder of the scan by substituting *the register values computed in the previous step* into P. This is the invariant assumption.

Then the scan continues. If it ever hits the above loop annotation again, the scan stops, a proof obligation is generated exactly as described above, and in addition for each register $r' \notin R$, a proof obligation (`=` e e') is generated, where e was the symbolic value of r' that the scan remembered at the loop entry, and e' is the current symbolic value of r'.[1]

Note that the loop-invariant annotation is not trusted. It merely functions as a "hint" to the VC generator, which then verifies that the invariant does indeed hold. This is an example of the principle that nothing in the untrusted binary is trusted, not even the annotations.

[1] Unless a bug in the compiler caused it to output a bad set R, the expressions e and e' should always be syntactically equivalent. Because this case is so common, the VC generator checks syntactic equivalence and does not output the proof obligation in this case.

There is a similar invariant mechanism that the compiler can insert at points of high branching or join factors, or at joins of cascaded if-then-else statements, in order to prevent the possibility of exponential blowup of expanding a DAG into a tree of paths.

The correctness of the VC generation strategy described here is proved in [14].

4.3 Handling of Class Meta-data

Before the VC generator scans the body of a method it examines the description of the classes contained in the scanned executable. First it checks the well-formedness of the class description tables using a procedure similar to the corresponding stage in the Java bytecode verifier [7].

Then the VC generator initializes the set of assumptions (A) that will be used while certifying the code. For each class in the executable and each class in the host-resident trusted library (e.g., the JDK), the VC generator creates assumptions about the object layout (using `ifield` predicates), about virtual method table layout (using `vmethod` predicates), about the current class hierarchy (using `jextends` and `jimplements` predicates), and others. (See Section 4.1.) Intuitively, these assumptions will allow the certification of field access, virtual method invocation and casting in the untrusted code. Examples of such initial assumptions and how they are used will be given in Section 5.

4.4 Handling of Exceptions

VC generation must examine all possible control paths of a program. Exceptions introduce new control paths. Consider the `example` object code in Figure 4. There are nine possible control paths from the call of `find` to a `ret` instruction:

1. All code terminates normally.

2. `find` throws a `NotFound` exception to the catch handler, which...

   (a) terminates normally.

   (b) throws `NullPointerException`.

   (c) throws an exception while uninstalling the catch-all handler during the call to `__Jv_GexExnHandler`.

3. `find` throws a `Throwable` object (other than a `NotFound` exception) to the catch-all handler

4. `find` terminates normally, but throws an exception while uninstalling the catch and catch-all handlers during the call to `__Jv_GetExnHandler`. (Symmetric to cases 2 and 3.)

Cases 2c and 4 should never happen with a well behaved runtime system. Nevertheless, the symbolic evaluator in the VC generator does not differentiate between calls to untrusted code and calls to trusted code such as `__Jv_GexExnHandler`. Instead, it conservatively assumes that any call may potentially throw any exception.

The VC generator keeps track of these control paths by maintaining a stack of installed handlers as it scans the code. The compiler helps the VC generator do this by annotating the places where the handlers are installed and uninstalled. The following annotations are shown in Figure 5:

- `ANN_INSTALLEDJAVAHANDLER(`$H_1, \ldots, H_k$`)` tells the VC generator that the machine instruction immediately preceding this annotation is a memory write that installs a handler descriptor containing a sequence of k handlers whose code begins at addresses $H_1, \ldots, H_k$. This annotation is used to install the exception handlers in a `try` statement with k `catch` blocks.

- `ANN_UNINSTALLEDJAVAHANDLER(`n`)` tells the VC generator that the machine instruction immediately preceding this annotation is a memory write that uninstalls n handler descriptors.

For instance, consider again the code in Figure 4. At the point of the call to `find`, the VC generator knows from the two preceding `ANN_INSTALLEDJAVAHANDLER` annotations that handlers at labels `L51` and `L52` have been pushed onto the global stack of active handlers. Therefore, the call to `find` may result in an immediate transfer of control to `L52` (case 2 above) or to `L51` (case 3 above), in addition to the possibility of a normal return (cases 1 and 4). The VC generator considers all of these paths as if the function call were suceeded by a fictitious multiway branch instruction. Additionally, before proceeding to consider each of the current handler instructions, the VC generator adds the assertion that the register `eax` contains an instance of the exception class handled by the respective handler.

The annotations for installing and uninstalling exception handlers have yet another purpose. Recall from the discussion at the beginning of this section that the VC generator makes the assumption that the stack slots are not aliased. With this assumption it is sound to consider the stack slot as an extension of the register file and not as arbitrary memory locations. This saves a large number of memory safety proof obligations and also effectively undoes the modifications to the program that a spiller in the register allocation phase might have done. To ensure that this non-aliasing assumption is sound the VC generator does not allow saving to arbitrary memory locations of registers that are known to contain stack addresses.[2]

In our compilation scheme for exceptions the longjmp data structure is stored on the stack (to allow safe operation in the presence of multiple threads) and the installation of a new handler involves storing the address of the topmost handler into a per-thread global variable (whose address is returned by the `__Jv_GetExcHandler`). This is done in our example by the last instruction in the expansion of the `INSTALL_1HANDLER` macro shown in Figure 5. Thus another purpose of the `ANN_INSTALLEDJAVAHANDLER` annotation is to mark such special memory writes. In the absence of the annotation the VC generator would stop and complain that its non-aliasing assumptions about the stack frame might be violated.

5 Certifying the Annotated Binaries

It is beyond the scope of this paper to describe the certification of the entire root-finding program. Instead, we focus in detail on the hand-optimized loop in the `eval` method. The certification process for this segment is shown in Figure 7. For simplicity, the `fxch` instruction at the loop entry has been moved to PC `119`; this has no effect on VC generation, but makes the control flow of the example easier to discuss.

First we explain some notation used in Figure 7. There are three columns in the figure. The first column shows a single control path through a fragment of disassembled object code. The second column shows the VC generation along this control path. The third column shows the generated proof of the VC. Recall from Section 4.2 the three steps that VC generation performs on each machine instruction. The proof obligations that step 1 outputs are labeled with "`prove:`" in the figure. This happens at two points:

- The `flds` instruction reads an array element. The safety preconditions for this memory read generate two proof obligations, which are shown immediately above the `flds` instruction.
- At the `jl` instruction, the control-flow path that takes the branch must reestablish the loop invariant. The resulting four proof obligations are shown below the `jl` instruction. (See Section 4.2 for further explanation of loop invariants.)

The assumptions that step 3 generates are labeled with "`A`n" in the figure. Recall that these assumptions hold for the remainder of the path; hence, they are labeled so the safety proof can use them. Assumptions are generated at two points:

- The loop invariant supplies four assumptions (`A37`–`A40`) for the loop body. These are shown immediately after the `ANN_LOOP` construct. (See Section 4.2 for further explanation of loop invariants.)
- At the `jl` instruction, the control-flow path that takes the branch generates an assumption from the current symbolic condition-code information. This assumption (`A41`) is shown immediately after the `jl` instruction.

There are 36 additional assumptions (`A1`–`A36`) that are already in scope by the time that the VC generator reaches the code shown in the figure. Six of these (`A10` and `A30`–`A34`) are needed in the proofs and thus are shown in the figure before the loop entry. Step 2, which performs the symbolic execution of the code, is not shown in Figure 7 due to lack of space. See Section 4.2 for further information about loop invariants and other details of VC generation.

Recall from the discussion of loop invariants in Section 4.2 that the VC generator creates fresh copies of every register in the `MODREG` set during the VC generation of the loop body. This is why there are subscripted forms of register names such as `eax_3` in the example.

After VC generation, the proof generator certifies the safety obligations output during step 1 of VC generation. The figures show these proofs next to their respective obligations. The proofs are built upon the labeled assumptions in scope. The proof rules used in these examples are shown in Figure 6. They are given in an abridged form of Elf [17], an implementation LF [3]. In the curried notation of Elf, the inputs are the premises and the output is the conclusion. The notation "`pf` P" means a proof of predicate P. In these proof rules, a capitalized variable is considered to be universally quantified.

For example, consider the proof of

```
(type (fmul f7_3 loc1_1) jfloat)
```

near the bottom of Figure 7. The proof is the application of axiom `fmulf` to two arguments, `A39` and `A33`. Consulting Figure 6, `fmulf` is a curried two-argument function over two universally quantified variables (denoted by capital letters), `E` and `E'`. This function, when given proofs of the predicates `(type E jfloat)` and `(type E' jfloat)`, yields a proof of

[2] This does not include the memory locations that are known to be in the stack frame. They are considered just like extra registers.

```
 instFld: pf (ifield C OFF T) ->
          pf (type E (jinstof C)) ->
          pf (nonnull E) ->
          pf (type (add E OFF) (ptr T)).

 tyField: pf (type ADDR (ptr T)) ->
          pf (type M mem) ->
          pf (type (sel4 M ADDR) T).

rdArray4: pf (type A (jarray T)) ->
          pf (type M mem) ->
          pf (nonnull A) ->
          pf (size T 4) ->
          pf (arridx OFF 4 (sel4 M (add A 4))) ->
          pf (saferd4 (add A OFF)).

tyArray4: pf (type A (jarray T)) ->
          pf (type M mem) ->
          pf (nonnull A) ->
          pf (size T 4) ->
          pf (arridx OFF 4 (sel4 M (add A 4))) ->
          pf (type (sel4 M (add A OFF)) T).
```

```
   faddf: pf (type E jfloat) -> pf (type E' jfloat) ->
          pf (type (fadd E E') jfloat).

   fmulf: pf (type E jfloat) -> pf (type E' jfloat) ->
          pf (type (fmul E E') jfloat).

 szfloat: pf (size jfloat 4).

 geswap: pf (ge E E') -> pf (le E' E).

    lt_b: pf (le 0 E) -> pf (lt E E') ->
          pf (ult E E').

 ge_add1: pf (lt E E'') -> pf (ge E E') ->
          pf (ge (add E 1) E').

 sub0chk: pf (neq E 0) -> pf (nonnull E).

  below1: pf (ult I LEN) -> pf (below I LEN).

   aidxi: pf (below I LEN) ->
          pf (arridx (add (imul I SIZE) 8) SIZE LEN).
```

Figure 6: The proof rules used in Figure 7.

`(type (fmul E E') jfloat)`. In other words the multiplication (`fmul`) of any two Java floats is a Java float. In this case: `E` is `f7_3`, and assumption `A39` is the proof of its type; `E'` is `loc1_1`, and assumption `A33` is the proof of its type.

We now describe the certification example in Figure 7 in detail. To relate back to Section 4.3, the examples in this section do not illustrate any global initial assumptions, but they do illustrate assumptions that the VC generator extracts from the Class Description Tables (CDTs) of untrusted code. Assumption `A10` of Figure 7 was produced by the VC generator when it scanned the CDT of the `Poly` class. This assumption says that there is an instance field that is an array of floats at byte-offset 4 within a `Poly` object. (Referring to the Java source in Figure 2, this is the `coefficients` field.)

This loop begins at PC `109` (or label `L31` in Figure 3) with an invariant predicate that is parsed from an `ANN_LOOP` annotation (originally shown in Figure 3) in the object file. The invariant says (1) that floating-point registers `f6`, which holds source variable `term`, and `f7`, which holds source variable `result`, are indeed loaded with `jfloat` values; and (2) that `edx`, which holds source variable `i`, is nonnegative and less than the value in memory location `(add eax 4)`, which is the length of the `this.coefficients` array. Register `eax` holds `this.coefficients`, and an array object stores its length at byte-offset 4 and its components starting at byte-offset 8.

This invariant must be established every time control flows to PC `109`. This can happen in two ways: during initial loop entry and through the conditional jump from PC `121`. Figure 7 does not show how the loop invariant is established upon initial loop entry; rather, it illustrates the control path that begins at PC `109` with the loop invariant as an assumption (assumptions `A37`–`A40`) and ends back at PC `109` with the loop invariant as a proof obligation (the four proof obligations shown after the end of the code segment). There is a subtlety in assumption `A37`. Whereas almost all of the registers in the loop invariant were replaced by fresh subscripted forms of themselves in assumptions `A37`–`40` (for reasons discussed earlier in this section), register `eax` was replaced by `(sel4 rm_1 (add loc2_1 4))`. The reason is that, unlike the other registers that appear in the invariant, `eax` is not modified around the loop. This fact is given as part of the `ANN_LOOP` annotation in the object file: `eax` is not included in the `MODREG` set and is thus marked as unmodified by the loop. (See Figure 3 for the MODREG set of this invariant.) As described above and in Section 4.2, for every register in the `MODREG` set, the VC generator creates a fresh subscripted form of the register at the beginning of the loop. But for each register that is not in the `MODREG` set, the VC generator starts the scan of the loop body with the symbolic value of that register upon loop entry, and then verifies or emits a proof obligation at the end of the loop that its symbolic value did not change (and hence is valid for all loop iterations). In this case, `eax` holds the hoisted computation of the `this.coefficients` field, which is at byte-offset 4 from `this`. (`this` was passed in stack slot `loc2`.)

As explained above, assumption `A10` gives the location and type of `this.coefficients`. Also, there are some relevant assumptions that were generated before the loop entry: `loc2` (`this`) is a nonnull object that is Java-castable to `Poly`, `loc1` (`x`) is a `jfloat`, and the value at byte-offset 4 of `loc2` (`this.coefficients`) is not `0`. This last assumption came from a null check that was hoisted out of the loop, so it should not be surprising that it will be needed to certify the loop.

The first instruction in the loop is a memory read of `this.coefficients[i]` into the floating-point unit. This instruction induces two proof obligations. First, the address must be safe to read; second, its value must be a `jfloat`. The proofs of these two obligations are fairly intricate, but one can get a quick intuition for them by ignoring the proof rules and just looking at the assumptions they use. In this case, they use `A10` (location and type of the `coefficients` field), `A30`–`A32` (`this` is a nonnull object that is Java-castable to `Poly`), `A34` (enough to prove that `this.coefficients` is nonnull), and `A37`–`A38` (enough to prove that `edx` is in bounds).

After the floating-point computations, we trace the path that loops back to PC `109`, hence generating assumption `A41`

```
OBJECT CODE                    VC GENERATION                                   PROOF GENERATION
                               A10: (ifield __CL_4root4Poly 4
                                            (jarray jfloat))
                               ...
                               A30: (type rm_1 mem)
                               A31: (nonnull loc2_1)
                               A32: (type loc2_1 (jinstof __CL_4root4Poly))
                               A33: (type loc1_1 jfloat)
                               A34: (neq (sel4 rm_1 (add loc2_1 4)) 0)
(L31:)
 109: ANN_LOOP(
        INV = {
          (lt
            edx
            (sel4 rm (add eax 4))),
          (ge edx 0),
          (type f7 jfloat),
          (type f6 jfloat)},
        MODREG = ...)
                               A37: (lt
                                      edx_3
                                      (sel4 rm_1
                                        (add (sel4 rm_1 (add loc2_1 4)) 4)))
                               A38: (ge edx_3 0)
                               A39: (type f7_3 jfloat)
                               A40: (type f6_3 jfloat)

                             prove: (saferd4                                   (rdArray4
                                      (add (sel4 rm_1 (add loc2_1 4))            (tyField (instFld A10 A32 A31) A30)
                                           (add (imul edx_3 4) 8)))              A30 (sub0chk A34) szfloat
                                                                                 (aidxi 4 (below1 (lt_b (geswap A38)
                                                                                                       A37))))
                             prove: (type (sel4 rm_1                           (tyArray4
                                            (add (sel4 rm1 (add loc2_1 4))       (tyField (instFld A10 A32 A31) A30)
                                                 (add (imul edx_3 4) 8)))        A30 (sub0chk A34) szfloat
                                          jfloat)                                (aidxi 4 (below1 (lt_b (geswap A38)
      flds 8(%eax, %edx, 4)                                                                            A37))))
 10d: fmul %st(2), %st(0)
 10f: faddp
 111: fxch %st(1)
 113: fmuls 12(%ebp)
 116: incl %edx
 117: cmpl %ecx, %edx
 119: fxch %st(1)
 121: jl 109
                               A41: (lt (add edx_3 1)
                                        (sel4 rm_1
                                          (add (sel4 rm_1 (add loc2_1 4))
                                               4)))

                             prove: (lt (add edx_3 1)                          A41
                                        (sel4 rm_1
                                          (add (sel4 rm_1 (add loc2_1 4))
                                               4)))

                             prove: (ge (add edx_3 1) 0)                       (ge_add1 A37 A38)

                             prove: (type (fmul f7_3 loc1_1) jfloat)           (fmulf A39 A33)

                             prove: (type (fadd                                (faddf
                                            (fmul                                (fmulf
                                              (sel4 rm_1                           (tyArray4
                                                (add (sel4 rm_1 (add loc2_1 4))      (tyField (instFld A10 A32 A31) A30)
                                                     (add (imul edx_3 4) 8)))        A30 (sub0chk A34) szfloat
                                              f7_3)                                  (aidxi 4 (below1 (lt_b (geswap A38)
                                            f6_3)                                                           A37))))
                                          jfloat)                                  A39)
                                                                                 A40)
```

Figure 7: The certification of the hand-optimized loop in `Poly.eval`.

that states that an increment of `edx` by `1` is still less than the length of `this.coefficients`.

Finally, the VC generator outputs the four conditions to reestablish the loop invariant. Proving that the new values of `f6` and `f7` are of type `jfloat` is straightforward, using standard proof rules such as `faddf` and `fmulf` to descend inductively into their symbolic structure and build on top of the initial types of `f6` and `f7` and the type of the accessed array element. But proving that `(add edx 1)` is still nonnegative is subtle because of 32-bit modular arithmetic. Hence the `ge_add1` proof rule shown in Figure 6. This rule intuitively says that if there exists some number `E''` that is bigger than `E` using signed 32-bit comparison, then adding `1` to `E` using signed 32-bit arithmetic will not cause its value to overflow and become -2^{31}. Even though the underlying domain of logical values is the set of integers, the special x86 operators such as `csublt` and `add` cast them to the appropriate signed or unsigned 32-bit numbers. For instance, `(csublt` x y`)` means that $(x + 2^{31}) \bmod 2^{32} - 2^{31} < (y + 2^{31}) \bmod 2^{32} - 2^{31}$ See Section 6 for an example of how this rule protects against an unsafe optimization.

The proof of the entire `eval` method (not just this loop) is 246 bytes.

6 Developing a PCC-generating Compiler

The previous sections hint at the complexity and tedium of reasoning about the correctness of an optimizing compiler and run-time system for a realistic programming language. In this section, we demonstrate how PCC helps to automate this detailed reasoning that otherwise must be done by hand.

Of the four main components of our PCC system, only the VC generator and proof checker are used by the host. In other words, these two components, which we refer to as the *PCC infrastructure*, must be incorporated into the trusted computing base of the host. Thus, their correctness has a direct bearing on safety. The compiler and proof generator, on the other hand, make use of the PCC infrastructure to check every target program and proof that they produce. Hence, safety violations that result from compiler and proof generator bugs will always be caught before transmitting them to the host, assuming no bugs in the PCC infrastructure.

For this reason, it is important that the PCC infrastructure be simple and small. The current VC generator is written in C and consists of approximately 23K lines of code. About 2.5K lines of this is a plug-in to the main VC generator that implements Java constructs. Another 4K lines or so implement a largely generic symbolic evaluator for the x86 instruction set, and the remainder is spread over tasks such as debugging assertions, binary-file parsing, LF representation, and so forth. The current proof checker is also written in about 1.4K lines of C code, including debugging assertions. It is largely unchanged from the proof checker described in [14], and is generic with respect to the set of proof rules. For our current Java system, we use a proof system defined in about 130 rules, taking up about 700 lines of LF specification. Taken all together, the PCC infrastructure, including both the VC generator and the proof checker, compiles and links into a single 52KB executable. All line counts include whitespace and comments.

The compiler is implemented in about 33K lines of ML code, and the proof generator in about 9K lines. Both components are still under heavy development, and hence are growing steadily. The compiler has been under development for approximately 12 months by two full-time programmers, with some contributions by a small number of part-time programmers. The proof-generator has been under development for about 5 months by one full-time programmer.

Part of the rapidity of the development process can be attributed to our use of the ML language (specifically, the Objective Caml dialect [6]). However, a major factor has also been our use of the proof generator as a debugging tool. In particular, we have observed that many bugs in the compiler and in our understanding of the run-time system interface are exposed by the proof generation process. Furthermore, the debugging output produced by the proof generator oftentimes allows us to identify quickly the nature of the bug. Many of these bugs would have been extremely difficult to uncover by standard testing techniques. Therefore, we believe that PCC has saved us possibly many weeks of development time.

To illustrate this point, consider Figure 8, which shows excerpts of the diagnostic output of our proof generator when bugs are inserted into the target code for the `eval` method.

Figure 8a shows the output when the `faddp` instruction within the loop in Figure 7 is replaced by a similar floating-point add that fails to pop the FPU stack. This would be a typical manifestation of a bug in which the register allocator loses track of the state of the FPU stack. The VC generator uses a register named `ftop` to denote the position of the FPU stack pointer. In this case, the position of `ftop` after each iteration of the loop is off by one. However, `ftop` was not in the `MODREG` set in the loop invariant and so the VC generator must ensure that the value of `ftop` is preserved between loop iterations. With the correct code shown in Figure 7, the VC generator's symbolic evaluation produced a value of `6` for `ftop` both at the loop entry and after one iteration. But when the `faddp` instruction is altered as described above, the VC generator outputs the additional proof obligation `(= 5 6)` because the value of `ftop` at the end of the loop body has changed (to `5`) and must be proven equal to the value at the beginning of the loop body (`6`). The proof generator outputs a diagnostic explaining that it cannot prove this predicate, reports where in the code this proof obligation is (`0x0109`, which is the location of the loop invariant), and describes the kind of proof obligation (`CKINVEQ`, which means an equality predicate induced by a register not marked as modified in the `MODREG` set of an invariant). In our experience, register allocation bugs of this sort almost always result in such nonsensical proof obligations.

Figure 8b is an example of a code-generation bug outside the scope Figure 7, so the reader must refer to Figure 3, which shows the entire code for the `eval` method. Block `L32` is the block that returns the result of the method; this block assumes upon entry that local variable `term` is on top of the FPU stack and that local variable `result` is the next value in the FPU stack. There are two ways that `L32` can be reached. The "normal" case is via the `jmp` instruction at the loop exit. The "rare" case is from block `L47` immediately before it. The rare case happens when `eval` is invoked on a polynomial with a length-0 `coefficients` array. In this case, the `jle` instruction in block `L42` skips the loop altogether. But at this point not even the initial values for `term` and `result` have been pushed onto the FPU stack. Therefore, block `L47` is compensation code that puts the FPU stack into a consistent state by loading 0.0 values for `term` and `result` before joining up with the "normal" case at `L32`. Figure 8b shows what happens when one of the `fldz` instructions in block `L47` is deleted. This is an example of a bug in the compensation routine of the register allocator.

```
Failed to prove _eval__4root4PolyF...
At %!COMM CKINVEQ  0x0109...
  Under assumptions [...]...
  Could not prove (= 5 6)
```

(a)

```
Failed to prove _eval__4root4PolyF...
At %!COMM RET  0x0126...
  Under assumptions [...]...
  Could not prove (type f0_1 jfloat)
```

(b)

Figure 8: Proof generator diagnostics when (a) the **faddp** in the loop in Figure 7 is replaced by **fadd %st,%st(1)**, and (b) one of the **fldz** instructions in block **L47** in Figure 3 is removed.

```
ANN_LOOP(INV = {(le edx (sel4 rm (add eax 4))), (ge edx 0), (type f7 jfloat), (type f6 jfloat)}
         MODREG = ...)

Failed to prove _eval__4root4PolyF
At %!COMM MEMRD  0x0109...
  Under assumptions
      [...,
       (type loc1_1 jfloat),
       (nonnull loc2_1),
       (type loc2_1 (jinstof __CL_4root4Poly)),
       (neq (sel4 rm_1 (add loc2_1 4)) 0),
       (le edx_3 (sel4 rm_1 (add (sel4 rm_1 (add loc2_1 4)) 4))),
       (ge edx_3 0),
       (type f7_3 jfloat),
       (type f6_3 jfloat)]...
  Could not prove
      (saferd4 (add (sel4 rm_1 (add loc2_1 4)) (add (imul edx_3 4) 8)))
    Could not prove
        (arridx (add (imul edx_3 4) 8) 4 (sel4 rm_1 (add (sel4 rm_1 (add loc2_1 4)) 4)))
      Could not prove
          (below edx_3 (sel4 rm_1 (add (sel4 rm_1 (add loc2_1 4)) 4)))
        Could not prove
            (ult edx_3 (sel4 rm_1 (add (sel4 rm_1 (add loc2_1 4)) 4)))
```

Figure 9: Proof generator diagnostic when **lt** is replaced by **le** in the loop invariant of Figure 7 (shown underlined).

In this case, the proof generator sees that **result** register **f0** does not contain a valid floating-point value. We note that this bug, which is typical of compensation errors in the compiler, could be extremely difficult to catch by testing because the control-flow path in question is unlikely to be taken in typical test inputs.

While these examples show the benefits of using a proof system to check the output of a compiler, the requirement that the compiler generate annotations introduces opportunities for bugs that are not present in a non-certifying compiler. Indeed, at one point during development we saw a significant proportion of our bugs turning up as errors in the generated invariants. For an example, consider Figure 9, in which the **lt** in the loop invariant from Figure 7 has mistakenly been replaced by an **le**. Unlike the previous figures, here we show more details of the diagnostic output of the proof generator. The output provides the address of the instruction at which the proof generation failed, followed by a partial list of the assumptions collected to this point. Then, a sequence of the subgoals that have failed to prove is listed.

With this error, the proof generator is unable to prove the safety of the array access in the loop, because the invariant allows the loop counter to equal, rather than be strictly less than, **coefficients.length**. In the diagnostic output, the last subgoal, **(ult edx_3 ...)**, is similar to one of the assumptions, **(le edx_3 ...)**, which immediately indicates an error in the code generated for the loop counter or an error in the generated loop invariant.

In some cases, seemingly correct optimizations are shown to be incorrect during proof generation. Consider, for example, a slight change to the **eval** loop in Figure 7 in which the loop counter is incremented by 2 instead of 1. In this case, the proof generator emits the diagnostic message shown (in excerpted form) in Figure 10. The diagnostic indicates that the loop counter, when incremented by 2, cannot be proven to be greater than or equal to 0, due to the possibility that it might become -2^{31} in twos-complement arithmetic. See the end of Section 5 for further explanation. While this situation is unlikely to occur in practical settings, it illustrates the fact that PCC checks safety in all possible execution scenarios.

```
Failed to prove _eval__4root4PolyF
At %!COMM INV1  0x0109...
  Under assumptions [...]...
  Could not prove
      (ge (add edx_3 2) 0)
```

Figure 10: Proof generator diagnostic when the **incl** instruction in Figure 7 is replaced with **addl $2,%edx**.

Finally, we found that the proof generator helped us to find errors in our specification of the run-time system. To see an example of this, consider Figure 11. This figure shows the specification for the run-time system routine called **__Jv_newFloatArray**, which is used to allocate new Java floating-point arrays. The specification is given as a pair of a precondition, which states that the length parameter must be non-negative, and a postcondition, which states that the array object is returned in register **eax** and that

```
PRE = {(ge loc1 0))}
POST = {(type eax (jarray jfloat)),
        (= (sel4 rm (+ eax 4)) loc1')}

Failed to prove _example__4root4Root
Under assumptions [...]...
Could not prove
    (safebcopy (add eax_2 8) _CT_4root4Root_5 8)
```

Figure 11: Proof generator diagnostic when the underlined postcondition is omitted from `__Jv_newFloatArray`.

the object's length field is initialized with the value of the length parameter. In our first version of this specification, we erroneously omitted the length-field initialization condition from the postcondition. With this omission, the proof generation of the `example` method in Figure 4 fails, with a diagnostic message that indicates that the new float array is not safe to initialize. We do not have the space in this paper to provide the details of this example.

7 Experimental Results

The Special J compiler and the associated prover are still under active development and we continue to extend the range of programs being compiled and validated. In this section we present some early experimental results with the intention to shed some light on the answers to questions such as how large type safety proofs are and how expensive theorem proving and proof checking are compared to compilation.

We show in Table 1, for a few of our internal test cases the proof sizes compared with the code sizes (both in binary form as they appear in the object file), and the theorem proving and checking times compared with the compilation times. This data shows that proofs for these small programs are relatively small, on average about 85% of the code size. The proof checking time is negligible compared with the compilation and theorem proving time. All measurements were performed on a machine using a Pentium Pro running at 300Mhz. The cost of proof checking is in practice linear in the size of the proofs. Also, when the proofs certify type safety their size is also linear in the size of the program to be type checked, at least for the type systems of Java and safe-C (used in Touschtone).

8 Related Work

The whole area of certifying compilation can be viewed as a special instance of result checking [20], in which the software being checked is a compiler and the property being checked is well typedness. In this case, as it is true for result checking in general, it turns out to be easier to verify properties of the output of the compilation instead of verifying the correctness of the compiler implementation.

The starting point and inspiration for the Special J compiler described in this paper was the Touchstone certifying compiler [15] whose source language is a small safe subset of the C programming language. The major advance of Special J over Touchstone is the scope of the source language compiled which in turn necessitates the handling of nontrivial run-time mechanisms such as object representation, dynamic method dispatch and exception handling. A further contribution of the present paper over previous publications on proof-carrying code lies in a detailed presentation of the code-annotation mechanism by which a verification condition generator can process machine code that involves indirect function calls or non-local control flow.

We currently use a semantic model of first-order types based on that described in [14]. To handle objects in an efficient way we had to extend the verification condition generator with a module that understands the details of Java-object representation. A more elegant solution could probably be obtained using the more complete semantic model of types of Appel and Felty [1], provided it is first extended to handle mutable data structures.

The literature contains reports on a number of certifying compilers. One of the most well-known is Sun's `javac` compiler for Java to Java bytecodes. Its purpose is quite similar to that of Special J. The issues are however much simpler because the language of bytecodes is so much more abstract than the optimized assembly language that we want to generate and check. Other certifying compilers that maintain types through compilation but drop them before final code generation are TIL [19] and Flint [18]. Most related to Special J are the Popcorn [8] and Cyclone [4] certifying compilers whose output language is typed assembly language (TAL) [10, 9]. A TAL program contains assembly language along with typing annotations and TAL pseudo-instructions that are used by the TAL type checker to check the assembly language program. Upon close inspection a TAL program looks very similar to our annotated assembly language. This suggests that the certifying compilation aspect of Special J and Popcorn are similar in principle. We use proof-carrying code as the output language because it is more general and relies on a simpler trusted component (the proof checker compared to the TAL type checker).

9 Conclusion

We have implemented a PCC system and certifying compiler that generates optimized x86 PCC binaries from Java source programs. The certifying compiler is largely conventional, except that it produces a small number of annotations in the target code to support VC and proof generation. The compiler performs register allocation and some global optimizations, including a form of partial redundancy elimination. The annotated target code is then processed by a small and fast PCC infrastructure consisting of a verification-condition (VC) generator, proof generator, and proof checker. Assembling the code and proof together results in a PCC binary that any host can quickly and reliably check for type safety.

The VC generator and proof checker are quite complete and have been stable for several months. The compiler and proof generator, on the other hand, are still under heavy development. At present, the compiler handles a large subset of the Java features, including objects, exceptions, and floating-point arithmetic. However, there are several key features that have yet to be implemented, including threads and dynamic class loading. Also, a number of important optimizations are not yet finished, including the elimination of null-pointer and array-bounds checks, and the stack allocation of non-escaping objects. Each new optimization typically requires additional support (often in the form of new proof rules) in the proof generator. And, as with any optimizing compiler for a large language, a considerable amount of performance tuning and debugging is still required. But as we have shown in this paper, PCC provides an excellent debugging tool for this compiler development.

In summary, we believe that our experience thus far allows us to conclude that PCC does indeed "scale up" to

Test	Code Size (bytes)	Proof Size (bytes)	(% of code)	Compilation time (ms)	Proving time (ms)	Checking time (ms)
inspection	2120	1244	58%	1820	2210	3.48
trees	1052	1404	133%	980	1240	3.60
root	864	450	52%	735	920	1.04
gc2	816	374	46%	407	460	0.88
spsmall	292	342	117%	344	310	0.67
basic	242	130	53%	157	230	0.41
fdivtest	248	320	129%	188	158	0.69
subtest	232	320	138%	188	157	0.69
handle	212	94	44%	172	313	0.34
floatcmp	208	112	54%	141	157	0.22
handle3	200	110	55%	188	166	0.30
ackerman	132	36	28%	110	94	0.06

Table 1: The experimental results for a few of our internal compiler test cases. We show the machine code size, the size of the binary encoding of proofs, and the time it takes to compile, prove, and check type safety.

handle the enforcement of type safety for languages of realistic size and complexity. Since the size of the annotations and the proofs required for certifying type safety increases linearly with the size of the program we do expect our system to scale up to the certification of large programs.

During our development project, the main impediment to further progress has been in the conventional aspects of compiler development; rarely did the special requirements imposed by PCC get in the way. In future work, we plan to release our current system for public use. Also of great interest is to extend the safety policy to go beyond Java type safety, in particular to allow enforcement of some constraints on the use of resources such as execution time and memory.

References

[1] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 243–253, January 2000.

[2] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[3] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[4] Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4), 1999.

[5] Intel. *Intel Architecture Software Developer's Manual.* Intel Corporation, 1997.

[6] Xavier Leroy. The ZINC experiment, an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[7] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.

[8] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for Systems Software*, Atlanta, May 1999.

[9] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1998. Extended version published as CMU technical report CMU-CS-98-178.

[10] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[11] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, October 1996.

[12] George Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1997.

[13] George C. Necula. Proof-carrying code. In *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[14] George C. Necula. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[15] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Canada, June 1998. ACM Press.

[16] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.

[17] Frank Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[18] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[19] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[20] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.