Type-Based Verification of Assembly Language for Compiler Debugging*

Bor-Yuh Evan Chang[1]
bec@cs.berkeley.edu

Adam Chlipala[1]
adamc@cs.berkeley.edu

George C. Necula[1]
necula@cs.berkeley.edu

Robert R. Schneck[2]
schneck@math.berkeley.edu

[1]Department of Electrical Engineering and Computer Science
[2]Group in Logic and the Methodology of Science
University of California, Berkeley

ABSTRACT

It is a common belief that *certifying compilation*, which typically verifies the well-typedness of compiler output, can be an effective mechanism for compiler debugging, in addition to ensuring basic safety properties. Bytecode verification is a fairly simple example of this approach and derives its simplicity in part by compiling to carefully crafted high-level bytecodes. In this paper, we seek to push this method to native assembly code, while maintaining much of the simplicity of bytecode verification. Furthermore, we wish to provide experimental confirmation that such a tool can be accessible and effective for compiler debugging. To achieve these goals, we present a type-based data-flow analysis or abstract interpretation for assembly code compiled from a Java-like language, and evaluate its bug-finding efficacy on a large set of student compilers.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Verification

Keywords: abstract interpretation, assembly code, bytecode verification, certified compilation, dependent types

1. INTRODUCTION

It is a widely held belief that automatic verification of low-level code can greatly aid the debugging of compilers by checking the compiler output. This belief has led to research, such as *translation validation* [PSS98, Nec00, RM99], which aims to verify complete correctness of a compiler (i.e., that the output of a compiler is semantically equivalent to the source program). We are concerned in this paper with the simpler technique of *certifying compilation*, in which the output of a compiler is checked for some internal consistency conditions, typically well-typedness in a certain type system. In particular, bytecode verification [LY97, GS01, Ler03] checks that the output of bytecode compilers is well typed. For this to be possible with a relatively simple algorithm, the bytecode language was carefully designed to carry additional information necessary for checking purposes and to include some high-level operations that encapsulate complex sub-operations, such as method dispatch or downcasting in the class hierarchy.

It is reasonable to expect that some bugs in a bytecode compiler can be detected by verifying the compiler output. In this paper, we go a step forward and extend the bytecode verification strategy to assembly language programs, while maintaining a close relationship with existing bytecode verification algorithms and preserving the features that make bytecode verification simple. The main motivation for going to the level of the assembly language is to reap the benefits of these techniques for debugging native-code compilers, not just bytecode compilers. A native-code compiler is more complex and thus, there is more room for mistakes. Additionally, in a mobile-code environment, type checking at the assembly language level results in eliminating the JIT compiler from the safety-critical code base. However, what distinguishes our approach from other certifying compilation projects is that we hope to obtain a verification algorithm that can be explained, even to undergraduate students, as a simple extension of bytecode verification, using concepts such as data-flow analysis and relatively simple types. In fact, undergraduate students in the compiler class at UC Berkeley have been the customers for this work, both in the classroom and also in the laboratory where they have used such verification techniques to improve the quality of their compilers.

The main contributions of this paper are as follows:

1. We describe the construction of a verifier, called Coolaid, using type-based abstract interpretation or data-flow analysis for assembly code compiled from a Java-like source language. Such a verifier does not require annotations for program points inside a procedure, which reduces the constraints on the compiler. We

*This research was supported in part by NSF Grants CCR-0326577, CCR-0081588, CCR-0085949, CCR-00225610, and CCR-0234689; NASA Grant NNA04CI57A; a Sloan Fellowship; an NSF Graduate Fellowship; an NDSEG Fellowship; and California Microelectronics Fellowships. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

TLDI'05, January 10, 2005, Long Beach, California, USA.

found that the main extension that is needed over bytecode verifiers is a richer type system involving a limited use of dependent types for the purpose of maintaining relationships between data values.

2. We provide experimental confirmation on a set of over 150 compilers produced by undergraduates that type checking at the assembly level is an effective way to identify compiler bugs. The compilers that were developed using type-checking tools show visible improvement in quality. We argue that tools that are easy to understand can help introduce a new generation of students to the idea that language-based techniques are not only for optimization, but also for improving software quality and safety.

In Section 2, we provide an overview of Coolaid, our assembly-level verifier for the Java-like classroom language Cool. Sections 3 to 5 describe the formal details of how the verifier works. In Section 6, we describe our experience using Coolaid in the compiler class, and in Section 7, we discuss related work.

2. OVERVIEW

Coolaid is an assembly-level abstract-interpretation-based verifier designed for a type-safe object-oriented programming language called Cool (Classroom Object-Oriented Language [Aik96])—more precisely, for the assembly code produced by a broad class of Cool compilers. The most notable features of Cool are a single-inheritance class hierarchy, a strong type system with subtyping, dynamic dispatch, a type-case construct, exceptions, and self-type polymorphism [BCM+93]. For our purposes, it can be viewed as a realistic subset of Java or C# extended with self-type polymorphism. Cool is the source language used in some undergraduate compilers courses at UC Berkeley and several other universities; this instantly provides a rich source of (buggy) compilers for experiments. We emphasize that Coolaid could not alter the design of the compilers, as it was not created until long after Cool had been in use.

There are two main difficulties with type-checking assembly code versus source code:

1. Flow sensitivity is required since registers are re-used with unrelated type at different points in the program; also, memory locations on the stack may be used instead of registers as a result of spill or to meet the calling convention.
2. High-level operations are compiled into several instructions with critical dependencies between them that must be checked. Furthermore, they may become interleaved with other operations, particularly after optimization.

The first problem is also present in bytecode verification and is addressed by using data-flow analysis/abstract interpretation to get a flow-sensitive type-checking algorithm that assigns types to registers (and the operand stack) at each program-point [Ler03, LY97]. However, the second is avoided with high-level bytecodes (e.g, `invokevirtual` for method dispatch in the JVML).

Coolaid, like bytecode verifiers, verifies by performing a data-flow analysis over an abstract interpretation. Abstract

$$\mathcal{R}_{\mathrm{BV}}(p) = \begin{cases} \mathit{Init} & \text{if } p \text{ is the start of the method} \\ \bigsqcup\{\langle S,R\rangle_p \mid \mathit{Instr}_{\mathrm{BV}}(p') : \mathcal{R}_{\mathrm{BV}}(p') \to_{\mathrm{BV}} \langle S,R\rangle_p\} & \text{otherwise} \end{cases}$$

Figure 1: Computing by abstract interpretation the abstract state $\mathcal{R}_{\mathbf{BV}}(p)$ at program point p. $\mathit{Instr}_{\mathbf{BV}}(p)$ is the instruction at p; $\sqcup$ denotes the join over the lattice; Init is the initial abstract state given by the declared types of the method arguments.

interpretation [CC77] successively computes over-approximations of sets of reachable program states. These over-approximations or *abstract states* are represented as elements of some lattice, called an *abstract domain.*

Consider first how this would be done in a bytecode verifier for Cool. The abstract domain is the Cartesian product lattice (one for each register) of the lattice of types; that is, the abstract state is a mapping from registers to types. The ordering is given by the subtyping relation, which is extended pointwise to the register state. The types are given almost completely by the class hierarchy, except with an additional type `null` to represent the singleton type of the `null` reference, $\top$ to represent unknown or uninitialized values, and (for convenience) $\perp$ to represent the absence of any value. As usual, the subtyping relation $<:$ follows the class inheritance hierarchy with a few additional rules for `null`, $\top$, and $\perp$ (i.e., is the reflexive-transitive closure of the "extends" relation and the additional rules). More precisely, let the class table T map class names to their declarations; then the subtyping relation (which is implicitly parameterized by T) is defined judgmentally as follows:

$\boxed{\tau_0 <: \tau_1}$

$$\frac{T(C_0) = \texttt{class } C_0 \texttt{ extends } C_1 \ \{\ \dots\ \}}{C_0 <: C_1} \qquad \frac{}{\mathsf{null} <: C}$$

$$\frac{}{\tau <: \top} \qquad \frac{}{\perp <: \tau} \qquad \frac{}{\tau <: \tau} \qquad \frac{\tau_0 <: \tau' \quad \tau' <: \tau_1}{\tau_0 <: \tau_1}$$

Note that since Cool is single-inheritance (and without interfaces), the above structure is a join semi-lattice (i.e., every finite set of elements has a least upper bound).

We can now describe the bytecode verifier as a transition relation between abstract states. Let $\langle S,R\rangle_p$ denote the abstract state at program point p where S and R are the types of the operand stack and registers, respectively. We write $bc\colon \langle S,R\rangle_p \to_{\mathrm{BV}} \langle S',R'\rangle_{p'}$ for the abstract transition relation for a bytecode bc; we elide the program points for the usual transition from p to $p{+}1$. For example, we show below the rule for `invokevirtual`:

$$\frac{\tau <: C \quad \tau'_0 <: \tau_0 \quad \cdots \quad \tau'_{n-1} <: \tau_{n-1}}{\begin{array}{l}\texttt{invokevirtual } C.m(\tau_0,\tau_1,\dots,\tau_{n-1}) : \tau_n \\ \quad : \langle \tau'_{n-1} :: \cdots :: \tau'_1 :: \tau'_0 :: \tau :: S, R\rangle \to_{\mathrm{BV}} \langle \tau_n :: S, R\rangle\end{array}}$$

where $C.m(\tau_0,\tau_1,\dots,\tau_{n-1}) : \tau_n$ indicates a method m of class C with argument types $\tau_0,\tau_1,\dots,\tau_{n-1}$ and return type τ_n. The first premise checks that the type of the receiver object at this point is a subtype of its (source-level) static type, while the other premises check conformance of the arguments. Note that the abstract transition for `invokevirtual` does not branch to the method as in the concrete semantics, but rather proceeds after the return

```
class Seq {
  Seq next() { ... }
}

class SubSeq extends Seq { }

class Main {
  void scan(SubSeq s) {
    Seq x = s;
    do {
      x = x.next();
    } while (x != null);
  }
}
```

(a) Cool

```
void scan(SubSeq);
  Code:
   0:  aload_1             // load s from r1
   3:  invokevirtual #2    // Seq.next()
                           // (call x.next())
  10:  ifnonnull 3         // x != null
  13:  return
```

(b) JVML

```
1  Main.scan:
        ⋮
2            r_x := r_y
3  Loop:
4            branch (= r_x 0) L_dispatch_abort
5            r_t := mem[(add r_x 8)]
6            r_t := mem[(add r_t 12)]
7            r_arg0 := r_x
8            r_ra := &L_ret
9            jump [r_t]
10 L_ret:
11           branch (= r_rv 0) L_done
12           r_x := r_rv
13           jump Loop
```

(c) SAL

Figure 2: An example program shown at the source, bytecode, and assembly levels.

with an assumption about return type (just like in type-checking).

The verification itself proceeds by symbolically executing the bytecode of each method using the abstract interpretation transition $\rightarrow_{\mathrm{BV}}$. An abstract state is kept for each program point, initially $\perp$ everywhere except at the start of the method, where the locations corresponding to the method arguments are typed according to the method's typing declaration. At each step, the state at the following program point is weakened according to the result of the transition. If no transition is possible (e.g., because a method call would be ill-typed), the verification fails. At return points, no transition is made, but the current state is checked to be well-typed with respect to the declared return type; otherwise, the verification fails.

To handle program points with multiple predecessors in the control-flow graph—join points—we use the join operation of the abstract domain. (Since there are no infinite ascending chains in this lattice, a widen operator is not required at cut points for termination.) Thus, the abstract states are computed as the least fixed point of equations in Figure 1. The verification succeeds if the least fixed point is computed without the verification failing due to a lack of any transition or due to an ill-typed return.

For example, consider the Cool program (written in Java syntax) given in Figure 2(a), along with the compilation of `Main.scan` to bytecode (JVML) in (b). We show below a computation of the abstract state at line 3.

	First Iteration	Second Iteration
$\mathcal{R}_{\mathrm{BV}}(3)$	$\langle \mathtt{SubSeq} :: S, R\rangle$	$\langle \mathtt{Seq} :: S, R\rangle$

The first time $\mathcal{R}_{\mathrm{BV}}(3)$ is computed, τ is `SubSeq` and then the `invokevirtual` is okay because $\mathtt{SubSeq} <: \mathtt{Seq}$. However, this requires that $\mathcal{R}_{\mathrm{BV}}(3)$ is weakened again because of the loop before we reach a fixed point.

We now extend these ideas to do verification on the assembly level. Coolaid works with a generic untyped assembly language called SAL; we hope SAL is intuitive and to streamline the presentation, we postpone formally presenting it until Section 5. However, note that in examples, we often use register names that are indicative of the source variables to which they correspond (e.g., $\mathbf{r}_x$) or the function they serve (e.g., $\mathbf{r}_{ra}$) though they correspond to one of the n machine registers. In Section 3, we describe the appropriate lattice of abstract states, and in Section 4, we describe the abstract transition relation $\rightarrow$. We close this section with an example illustrating the difficulty of assembly-level verification.

Consider again the example program given in Figure 2 where the compilation of `Main.scan` to assembly code (SAL) is shown in (c). Note that the `invokevirtual` bytecode in line 3 of (b) corresponds to lines 4–10 of (c); `invokevirtual` is expanded into (1) a null-check on the receiver object, (2) finding the method through the dispatch table, (3) saving the return address, and (4) finally an indirect jump. The simple rule for `invokevirtual` is largely due to the convenience of rolling all of these operations into one atomic bytecode. For example, references of type C mean either `null` or a valid reference to an object of class C. Since dynamic dispatch requires the receiver object to be non-null, it is convenient to make this check part of the `invokevirtual` bytecode. In the assembly code, these operations are separated, which then require the typing of intermediate results (e.g., dispatch tables) and tracking critical dependencies. To fully illustrate this issue, let both $\mathbf{r}_{child}$ and $\mathbf{r}_{parent}$ have static type `Seq` in our verification, but suppose $\mathbf{r}_{child}$ actually has dynamic type `SubSeq` and $\mathbf{r}_{parent}$ has dynamic type `Seq` during an execution. Now consider the following code:

```
1        branch (= r_child 0) L_dispatch_abort
2        r_t := mem[(add r_child 8)]
3        r_t := mem[(add r_t 12)]
4        r_arg0 := r_parent
5        r_ra := &L_ret
6        jump [r_t]
7  L_ret:
```

An initial implementation following bytecode verification might assign the type $\mathbf{r}_t : \mathsf{meth}(\mathtt{Seq}, 12)$ saying $\mathbf{r}_t$ is method at offset 12 of class `Seq` (since it was found through $\mathbf{r}_{child}$, which has type `Seq`). On line 6, we can then recognize that $\mathbf{r}_t$ is a method and check that $\mathbf{r}_{arg_0} : \mathtt{Seq}$, which succeeds since $\mathbf{r}_{parent} : \mathtt{Seq}$. However, this is unsound because at run-time, we obtain the method from `SubSeq` but pass as the receiver object an object with dynamic type `Seq`, which may lack expected `SubSeq` features.[1]

One way to resolve this unsoundness is to make sure that the receiver object passed to the method is the same object on which we looked up the dispatch table. We will now describe a type system to handle these difficulties.

[1]This was first observed as an unsoundness in the Touchstone certifying compiler for Java [CLN+00] by Christopher League [LST03].

3. ABSTRACT STATE

At the assembly level, high-level bytecodes are replaced by series of instructions, primarily involving address computation, that may be re-ordered and optimized. We have found it useful to not always create new types for intermediate values, but to keep certain intermediate expressions in symbolic form. Rather than assigning types to registers, we assign types to *symbolic values*. Thus, our abstract state consists of a mapping Σ from registers to expressions (involving symbolic values) and a mapping Γ from symbolic values to types.

abstract state	$\langle \Sigma \,ş\, \Gamma \rangle$	
value state	Σ	$::= \mathbf{r}_0 = e_0, \mathbf{r}_1 = e_1, \ldots, \mathbf{r}_{n-1} = e_{n-1}$
type state	Γ	$::= \cdot \mid \Gamma, \alpha : \tau$
symbolic values	α, β	

We assume some total ordering on symbolic values, say from least recently to most recently introduced.

3.1 Values

The language of expressions has the following form:

$$\text{expressions} \quad e ::= n \mid \alpha \mid \&L \mid e_0 + e_1 \mid e_0 - e_1 \mid e_0 \cdot e_1 \mid \cdots$$

These are the expressions ae of the assembly language (for which see Section 5), except replacing registers with symbolic values. Note $\&L$ refers to the code or data address corresponding to label L.

We define a normalization of expressions to values. For Coolaid, we are only concerned about address computation and a few additional constraints to express comparison results for non-null checks and type-case.[2] Thus, the values are as follows:

$$\begin{array}{ll} \text{values} & v ::= n_0 \cdot \&L + n_1 \cdot \alpha + n_2 \mid \alpha \; R \; n \\ \text{relations} & R ::= = \mid \neq \mid < \mid \leq \mid > \mid \geq \end{array}$$

Note that the form of the address computation allows indexing into a statically allocated table with a constant multiple and offset of a symbolic value (e.g., a class tag) or indexing into a table given by a symbolic value (e.g., a dispatch table) by a constant offset. No other address forms are necessary in Coolaid.

The symbolic values represent existentially quantified values, for which the inner structure is unknown or no longer relevant. Coolaid will often choose to *freshen* registers, forgetting how its value was obtained by replacing it with a fresh symbolic value, which is assigned an appropriate type. In particular, during normalization we might choose to forget values (replacing subexpressions with fresh symbolic values) while retaining types (by assigning appropriate types to the new symbolic values). Thus, we use a type-directed judgment $\Gamma \vdash e \Downarrow v \triangleright \Gamma'$ for the normalization of expression e to value v, yielding a possibly extended Γ for new symbolic values. In most cases, the new symbolic value can be implicitly typed as $\top$ (i.e., unknown contents); for example, should a program multiply two pointer types, Coolaid determines that it is not worth retaining any information either about the structure of the value or its type. Coolaid also has a distinguished type of initialized machine words (as described in Section 3.2), and in this case, although the fact that the value is a product can be forgotten, we still wish to treat the new value as an initialized machine word. Simplifying arithmetic on words is in fact the only interesting case when Γ is extended. For example,

$$\frac{\begin{array}{lll} \Gamma \vdash e_0 \Downarrow \alpha_0 \triangleright \Gamma' & \Gamma'(\alpha_0) = \mathsf{word} & \\ \Gamma' \vdash e_1 \Downarrow \alpha_1 \triangleright \Gamma'' & \Gamma''(\alpha_1) = \mathsf{word} & (\beta \text{ fresh}) \end{array}}{\Gamma \vdash e_0 \cdot e_1 \Downarrow \beta \triangleright \Gamma''[\beta \mapsto \mathsf{word}]}$$

where we write $\Gamma[\alpha \mapsto \tau]$ for the updated map that is the same as Γ except α maps to τ. It is fairly straightforward to define the normalization.

One of the more important uses of the value state is to convey that two registers are equal, which can be represented by mapping them to the same value. This is necessary, for instance, to handle a common compilation strategy where a value in a stack slot is loaded into a register to perform some comparison that more accurately determines its type; Coolaid must realize that not only the scratch register used for comparison but also the original stack slot has the updated type. We write that $\langle \Sigma \,ş\, \Gamma \rangle \models r_0 = r_1$ to mean that the abstract state $\langle \Sigma \,ş\, \Gamma \rangle$ implies that registers r_0 and r_1 are equal, and define this as follows:

$$\langle \Sigma \,ş\, \Gamma \rangle \models r_0 = r_1 \text{ if and only if}$$
$$\Gamma \vdash \Sigma(r_0) \Downarrow v \triangleright \Gamma' \text{ and } \Gamma \vdash \Sigma(r_1) \Downarrow v \triangleright \Gamma''$$

which simply says that $r_0 = r_1$ precisely when their contents normalize to the same value.

3.2 Types

We use a (simple) dependent type system extending the non-dependent types used in bytecode verification. While we could merge the reasoning about values described in the previous section into the type system (for example, introducing singleton types for integer constants), we have found it more convenient to separate out the arithmetic and keep the type system simpler.

Primitive Types. Though not strictly necessary to prove memory safety, we distinguish two types of primitive values: one for completely unknown contents (e.g., possibly uninitialized data) and one for an initialized machine word of an arbitrary value. One could further distinguish word into words used as machine integers versus booleans and perhaps catch more bugs.

types	$\tau ::= \top$	unknown contents
	$\mid \mathsf{word}$	machine word
	$\mid \bot$	absence of a value
	$\mid \ldots$	

Reference Types. To safely index into an object via an object reference, we must ensure the reference is non-null. Furthermore, sometimes we have and make use of knowledge of the exact dynamic type. Thus, we refine references types to include the type of possibly-null references bounded above (C), the type of non-null references nonnull C bounded above, the type of possibly-null references bounded above and below (exactly C), the type of non-null references bounded above and below (nonnull exactly C), and the type of `null` (null).[3] For self-type polymorphism, we also consider object references where the class is known to be the same

[2]As typical for assembly language, we have expression operators corresponding to arithmetic comparisons $=$, $<$, etc.

[3]Putting aside historical reasons, one might prefer to write C for non-null references and maybenull C for possibly-null references, viewing non-null references as the core notion.

as another symbolic value ($\mathsf{classof}(\alpha)$). Finally, we have pointers to other types, which arise, for example, from accessing object fields or indexing into a compiler-generated table (e.g., dispatch tables). Though not expressed in the abstract syntax, Coolaid only uses single-level pointers (i.e., $C\ \mathsf{ptr}$ but not $C\ \mathsf{ptr}\ \mathsf{ptr}$).

types	$\tau ::= \ldots$	
	$\mid [\mathsf{nonnull}]\ b$	object reference of class given by bound b [possibly `null` if not nonnull]
	$\mid \mathsf{null}$	the `null` reference
	$\mid \tau\ \mathsf{ptr}$	pointer to a τ
	$\mid \ldots$	
bounds	$b ::= C$	bounded above by C
	$\mid \mathsf{exactly}\ C$	bounded above and below by C
	$\mid \mathsf{classof}(\alpha)$	same class as α
classes	C	

Dispatch Table and Method Types. For method dispatches, we have types for the dispatch table of an object ($\mathsf{disp}(\alpha)$) and a method obtained from such a dispatch table ($\mathsf{meth}(\alpha, n)$). A similar pair is defined for the dispatch table and methods of a specific class ($\mathsf{sdisp}(C)$ and $\mathsf{smeth}(C, n)$). We also define a type for initialization methods ($\mathsf{init}(\alpha)$ and $\mathsf{sinit}(C)$).

types	$\tau ::= \ldots$	
	$\mid \mathsf{disp}(\alpha)$	dispatch table of α
	$\mid \mathsf{meth}(\alpha, n)$	method of α at offset n
	$\mid \mathsf{sdisp}(C)$	dispatch table of class C
	$\mid \mathsf{smeth}(C, n)$	method of class C at offset n
	$\mid \mathsf{init}(\alpha)$	initialization method of α
	$\mid \mathsf{sinit}(C)$	initialization method of class C
	$\mid \ldots$	

Tag Type. To handle a type-case (or a down cast), we need a type for the class tag of an object. The class tag is the run-time representation of the dynamic type of the object. In addition to the object value whose tag this is, we keep a set of the possible integers that the tag could be.

types	$\tau ::= \ldots$	
	$\mid \mathsf{tag}(\alpha, N)$	tag of α with possible tags N
tag sets	N	

Subtyping. As with bytecode verification, the ordering on the abstract domain elements is largely defined in terms of subtyping. Though we have extended the language of types a fair amount, the lattice of types remains quite simple—flat except for reference types. Since our types now depend on symbolic values, we extend the subtyping judgment slightly to include the context mapping symbolic values to types—$\Gamma \vdash \tau_0 <: \tau_1$. The basic subtyping rules from before carry over (extended with Γ). Then, we add the expected relations between exactly, nonnull and possibly-null references.

$$\frac{}{\Gamma \vdash \mathsf{nonnull}\ C <: C} \qquad \frac{}{\Gamma \vdash \mathsf{exactly}\ C <: C}$$

$$\frac{}{\Gamma \vdash \mathsf{nonnull\ exactly}\ C <: \mathsf{nonnull}\ C}$$

$$\frac{}{\Gamma \vdash \mathsf{nonnull\ exactly}\ C <: \mathsf{exactly}\ C}$$

Non-null references are also ordered following the class hierarchy.

$$\frac{\Gamma \vdash C_0 <: C_1}{\Gamma \vdash \mathsf{nonnull}\ C_0 <: \mathsf{nonnull}\ C_1}$$

Finally, some slightly subtle handling is required for a precise use of classof. If α has type C, we would like to be able to use values of type $\mathsf{classof}(\alpha)$ as being of type C.

$$\frac{\Gamma \vdash \Gamma(\alpha) <: q'\ C}{\Gamma \vdash q\ \mathsf{classof}(\alpha) <: q\ C}$$

$$\frac{\Gamma \vdash \Gamma(\alpha) <: q'\ \mathsf{exactly}\ C}{\Gamma \vdash q\ \mathsf{classof}(\alpha) <: q\ \mathsf{exactly}\ C}$$

In these rules q and q' might either, both, or neither be nonnull. Observe that the structure of abstract states allows instances of classof where nothing is provable from these rules; for example, we might have $\alpha : \mathsf{classof}(\beta)$ and $\beta : \mathsf{classof}(\alpha)$; however, we can prevent this by restricting the type of a symbolic value to not depend on "newer" symbolic values (following the ordering on symbolic values). Note that the structure of abstract transitions does not allow such states to be created by observing this restriction.

3.3 Join

It remains to define the join operation on abstract states. Intuitively, the core of the join operation is still determined by subtyping; however, some extra work must be done to join values and dependent types.

We consider values (i.e., v) as a (fancy) labeling of equivalence classes of registers, so the lattice of value states is the lattice of finite conjunctions of equality constraints among registers. The join algorithm is then essentially a special case of the algorithm for the general theory of uninterpreted functions given by Gulwani *et al.* [GTN04] and by Chang and Leino [CL05].

First, we evaluate each expression of the states to join so that we are only working with values. Let $A_0 = \langle \Sigma_0 \,;\, \Gamma_0 \rangle$ and $A_1 = \langle \Sigma_1 \,;\, \Gamma_1 \rangle$ denote these states, and let $A = \langle \Sigma \,;\, \Gamma \rangle$ be the result the join. The resulting value state Σ will map all registers to values. Let us momentarily denote a value in the joined state as the corresponding pair of values in the states to be joined. Then we can define the resulting value state as follows:

$$\Sigma(r) = \langle \Sigma_0(r), \Sigma_1(r) \rangle$$

Finally, we translate pairs of values $\langle v_0, v_1 \rangle$ to single values and yield the new type state Γ; if the structures of v_0 and v_1 do not match, then they are abstracted as a fresh symbolic value. More precisely, let $\ulcorner \cdot \urcorner$ be the translation of the pair of values to a single value:

$$\ulcorner \langle \alpha_0, \alpha_1 \rangle \urcorner \stackrel{\text{def}}{=} \beta$$
$$\quad \text{where } \beta \text{ fresh and } \Gamma(\beta) = \langle \Gamma_0, \Gamma_0(\alpha_0) \rangle \sqcup_{<:} \langle \Gamma_1, \Gamma_1(\alpha_1) \rangle$$

$$\ulcorner \langle n_0 \cdot \&L + n_1 \cdot \alpha_0 + n_2, n_0 \cdot \&L + n_1 \cdot \alpha_1 + n_2 \rangle \urcorner \stackrel{\text{def}}{=} n_0 \cdot \&L + n_1 \cdot \ulcorner \langle \alpha_0, \alpha_1 \rangle \urcorner + n_2$$

$$\ulcorner \langle \alpha_0\ R\ n, \alpha_1\ R\ n \rangle \urcorner \stackrel{\text{def}}{=} \ulcorner \langle \alpha_0, \alpha_1 \rangle \urcorner\ R\ n$$

$$\ulcorner \langle v_0, v_1 \rangle \urcorner \stackrel{\text{def}}{=} \beta \text{ where } \beta \text{ fresh and } \Gamma(\beta) = \top \qquad \text{(otherwise)}$$

Note that each distinct pair of symbolic values maps to a fresh symbolic value. We write $\sqcup_{<:}$ for the join in the types lattice. Although the subtyping lattice is flat at dependent types, we use the same join operation at values to update the dependencies. For example, $\mathsf{disp}(\alpha)$ and $\mathsf{disp}(\beta)$ would be joined to $\mathsf{disp}(\ulcorner\langle\alpha,\beta\rangle\urcorner)$. More precisely, let σ_0 and σ_1 denote substitutions from the symbolic values in A to A_0 and A_1, respectively, given by the above translation. Then,

$$\langle\Gamma_0,\tau_0\rangle \sqcup_{<:} \langle\Gamma_1,\tau_1\rangle \stackrel{\text{def}}{=} \text{the least } \tau \text{ such that} \\ \Gamma_0 \vdash \tau_0 <: \sigma_0(\tau) \text{ and } \Gamma_1 \vdash \tau_1 <: \sigma_1(\tau)$$

Observe that Coolaid takes a rather simple approach to joining values. In particular, registers are often freshened to be pure symbolic values at join points. As a trivial example, Coolaid is able to verify a program that takes a pointer, adds one, and then subtracts one; but if a join point intervenes, the fact that the register contains a value that is just one more than a pointer type may be forgotten (unless that register in the other state also contains one more than the same pointer type). This simplification did not seem to cause difficulties in practice, with the many student compilers of our experiments.

Since there are a finite number of registers, there is a bounded number of equivalence classes. The join only increases the number of equivalence classes. Since there are no infinite ascending chains in the types lattice, the abstract interpretation will terminate (without requiring a widen operation at cut points).

4. VERIFICATION

As in Section 2, we describe the verification procedure by the abstract transition relation

$$I\colon \langle\Sigma \fatsemi \Gamma\rangle_p \to \langle\Sigma' \fatsemi \Gamma'\rangle_{p'}.$$

As before, this determines a verification procedure by the fixed-point calculation over the equations analogous to those of Figure 1. In this section, we define some interesting cases of the abstract transition relation by following the verification of an example. All the abstract transition and typing rules are collected together in Appendix A.

We first consider in detail the assembly code in Figure 2(c), which performs a dynamic dispatch in a loop. Suppose the abstract state before line 2 is as follows:

$$\begin{array}{l}\langle \mathbf{r}_y = \alpha_y^1, \mathbf{r}_{self} = \alpha_{self}^1 \fatsemi \\ \ \alpha_y^1 : \mathsf{SubSeq}, \alpha_{self}^1 : \mathsf{nonnull}\ \mathtt{Main}\rangle\end{array} \tag{1}$$

and all other registers map to distinct symbolic values that have type $\top$. (Where appropriate, we use subscripts on symbolic values to indicate the register in which they are stored and superscripts on symbolic values to differentiate them.) For the rest of this example, we usually write just what changes. Since instruction 2 is a register to register move, we simply make $\mathbf{r}_x$ map to the same value as $\mathbf{r}_y$. This changes the abstract state to

$$\langle \mathbf{r}_x = \alpha_y^1, \mathbf{r}_y = \alpha_y^1, \ldots \fatsemi \ldots\rangle$$

In general, for an arithmetic operation, we simply update the register with the given expression (with no changes to the type state):

$$\frac{}{r := ae : \langle\Sigma \fatsemi \Gamma\rangle \to \langle\Sigma[r \mapsto \Sigma(ae)] \fatsemi \Gamma\rangle}\ \mathsf{assign}$$

where we treat Σ as a substitution (i.e., $\Sigma(ae)$ is the expression where registers are replaced by their mapping in Σ).

Line 3 does not affect the state, as labels are treated as no-ops.

$$\frac{}{L\colon : \langle\Sigma \fatsemi \Gamma\rangle \to \langle\Sigma \fatsemi \Gamma\rangle}\ \mathsf{label}$$

We recognize line 4 as a null-check so that the abstract state in the false branch is

$$\langle \ldots \fatsemi \alpha_y^1 : \mathsf{nonnull}\ \mathtt{SubSeq}\rangle$$

Note that we automatically have that both the contents of $\mathbf{r}_x$ and $\mathbf{r}_y$ are non-null since we know that they are aliases (for they map to the same symbolic value). In general, the post states of a null-check are given as follows:

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ 0 \rhd \Gamma' \quad \Gamma'(\alpha) = b \quad R \in \{=,\neq\} \\ \tau = \begin{cases}\mathsf{nonnull}\ b & \text{if } \neg(\alpha\ R\ 0) \equiv \alpha \neq 0 \\ \mathsf{null} & \text{otherwise}\end{cases}\end{array}}{\mathtt{branch}\ ae\ L : \langle\Sigma \fatsemi \Gamma\rangle \to \langle\Sigma \fatsemi \Gamma'[\alpha \mapsto \tau]\rangle}\ \mathsf{nullcheck}_F$$

The true case is similar.

We recognize that line 5 loads the dispatch table of object α_y^1, and the abstract state afterwards is

$$\langle \mathbf{r}_t = \alpha_t^5, \ldots \fatsemi \alpha_t^5 : \mathsf{disp}(\alpha_y^1), \ldots\rangle$$

The basic invariant for memory accesses we maintain throughout is that an address is safe to access if and only if it is a ptr type, and thus the rule for reads is as follows:

$$\frac{\Gamma \vdash \Sigma(ae) : \tau\ \mathsf{ptr} \rhd \Gamma' \quad (\alpha \text{ fresh})}{r := \mathtt{mem}[ae] : \langle\Sigma \fatsemi \Gamma\rangle \to \langle\Sigma[r \mapsto \alpha] \fatsemi \Gamma'[\alpha \mapsto \tau]\rangle}\ \mathsf{read}$$

The above rule introduces the following typing judgment:

$$\Gamma \vdash e : \tau \rhd \Gamma'$$

which says in context Γ, e has type τ, yielding a possibly extended Γ for new symbolic values Γ'. The typing rule that determines that line 5 looks up a dispatch table is

$$\frac{\Gamma \vdash e \Downarrow \alpha + 8 \rhd \Gamma' \quad \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma''}{\Gamma \vdash e : \mathsf{disp}(\alpha)\ \mathsf{ptr} \rhd \Gamma''}$$

We determine that offset 8 of an object is a pointer to the dispatch table because knowledge of the Cool object layout is built-in.

Line 6 then looks up the appropriate method in the dispatch table, so the post state is

$$\langle \mathbf{r}_t = \alpha_t^6, \ldots \fatsemi \alpha_t^6 : \mathsf{meth}(\alpha_y^1, 12), \ldots\rangle$$

This is again a memory read, so the transition rule read applies, but the method type is determined with the following typing rule:

$$\frac{\begin{array}{ll}\Gamma \vdash e \Downarrow \beta + n \rhd \Gamma' & \Gamma'' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma''' \\ \Gamma' \vdash \beta : \mathsf{disp}(\alpha) \rhd \Gamma'' & (C \text{ has a method at offset } n)\end{array}}{\Gamma \vdash e : \mathsf{meth}(\alpha, n)\ \mathsf{ptr} \rhd \Gamma'''}$$

We get a method if we index into the dispatch table, provided a method at that offset is defined (according to the implicitly parameterized class table T).

The next two lines (7 and 8) set the first argument register (which is used to pass the receiver object) and the return address. The abstract state after line 8 is as follows (given by assign):

$$\langle \mathbf{r}_{arg0} = \alpha_y^1, \mathbf{r}_{ra} = \&\mathrm{L}_{\mathrm{ret}}, \ldots \fatsemi \ldots\rangle$$

Finally, in line 9, the method call takes place. This indirect jump is determined to be a method call since $\mathbf{r}_t$ contains a value of method type. The post state after the call must drop any information about the state prior to the call, for the callee may modify the registers arbitrarily. This is expressed by giving fresh symbolic values to all registers. The information we have about the post state is that the return value has the type specified by the method signature. Thus, the abstract state after the call is

$$\langle \mathbf{r}_{rv} = \alpha^9_{rv} \,⨾\, \alpha^9_{rv} : \mathsf{Seq} \rangle$$

and the method dispatch transition rule is as follows:

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma(ae) : \mathsf{meth}(\alpha, n) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \alpha \qquad \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma'' \qquad\qquad (*) \\ T(C) = \mathtt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_k)\ \ldots\ \} \\ \Gamma'' \vdash \Sigma(\mathbf{r}_{arg_1}) : \tau_1 \rhd \Gamma''_1\ \cdots\ \Gamma''_{k-1} \vdash \Sigma(\mathbf{r}_{arg_k}) : \tau_k \rhd \Gamma''_k \\ (\Sigma', \beta \text{ fresh}) \\ (m \text{ is the method at offset } n \text{ of class } C)\end{array}}{\mathtt{jump}\ [ae] : \langle \Sigma \,⨾\, \Gamma \rangle \to \langle \Sigma'[\mathbf{r}_{rv} \mapsto \beta] \,⨾\, \Gamma''_k[\beta \mapsto \tau_{rv}] \rangle}\ \mathsf{meth}$$

This rule is slightly simplified in that it ignores callee-save registers; however, we can easily accommodate callee-save registers by preserving the register state for those registers (i.e., $\Sigma(\mathbf{r}_{cs}) = \Sigma'(\mathbf{r}_{cs})$ for each callee-save register $\mathbf{r}_{cs}$). Also, this rule is slightly more conservative than necessary within our type system. The premise marked with $(*)$ requires that the receiver object be the same as the object from which we looked up the dispatch table. We could instead require only that it can be shown to have the same dynamic type as α (i.e., checking that $\Sigma(\mathbf{r}_{arg_0})$ has type $\mathsf{nonnull\ classof}(\alpha)$), but this restriction is probably helpful for finding bugs. Note if the declared return type of the method is the self-type, then we take τ_{rv} to be $\mathsf{classof}(\alpha^1_{self})$ (i.e., to have same dynamic type as self).

Lines 10–12 are a label, null-check, and register to register move, as we have seen before, so the abstract state before line 13 is

$$\langle \mathbf{r}_x = \alpha^9_{rv}, \ldots \,⨾\, \alpha^9_{rv} : \mathsf{nonnull\ Seq}, \ldots \rangle$$

The jump instruction at line 13 loops back with the abstract transition given by

$$\frac{(L \text{ is not a code label for a method})}{\mathtt{jump}\ L : \langle \Sigma \,⨾\, \Gamma \rangle \to \langle \Sigma \,⨾\, \Gamma \rangle_L}\ \mathsf{jump}$$

that does not modify the abstract state but makes it a predecessor of L. This weakens $Pre(3)$, and thus this loop body will be scanned again before reaching a fixed point. This transition applies only to jumps within the method, rather than calls to other functions.

Static Dispatch. All method/function calls are treated similarly in that they check that arguments conform to the appropriate types, havoc the abstract state (except for callee-save registers), assume the return value has the specified return type, and proceed to the next instruction. They differ in how the function (or class of possible functions) to be called is determined.

Static dispatch determines the method to call based on a type specified statically, so the compiler can simply emit a direct jump to the code label for the method. However, in many of the Cool compilers with which we experimented, we observed that static dispatch was with indirect jumps based on indexing into the dispatch table for the particular class or even first obtaining the dispatch table by indexing through a statically allocated, constant "prototype object" (perhaps to re-use code in the compiler). We treat all these cases uniformly by assigning types of methods and dispatch tables $\mathsf{smeth}(C, n)$ and $\mathsf{sdisp}(C)$ to the appropriate labels at initialization time. The rules to look up methods and dispatch tables for static dispatch are analogous to the ones for dynamic dispatch (see Appendix A).

Type-Case. Coolaid's handling of the type-case (or down casts) is probably the language feature most tailored to a particular compilation strategy. In fact, this is the most prominent example of a potential *Coolaid incompleteness*: a memory-safe compilation that fails to verify with Coolaid (see Section 6.1). The way that Coolaid handles type-case is based on the compilation strategy that emits comparisons between the class tag and integer constants to determine the dynamic type of an object. Following this strategy, Coolaid updates the $\mathsf{tag}(\alpha, N)$ type by filtering the set of possible tags N on branches and then updates α to the type that is the least upper bound of the remaining tags. If the set becomes empty, then we have determined an unreachable branch, so Coolaid will not follow such a branch.

5. DETAILS

In this section, we discuss three details of our approach to assembly-level verification required for a complete presentation. First, the assembly language itself. Coolaid is implemented on top of the Open Verifier framework for foundational verifiers [CCNS05, Sch04], which provides an infrastructure for abstract interpretation on assembly code (among other things)[4]. This framework works on a generic untyped assembly language called SAL by first translating from MIPS or Intel x86 assembly. The abstract syntax of SAL is as follows:

instructions	$I ::= L:$	label
	$\mid\ r := ae$	assignment
	$\mid\ r := \mathtt{mem}[ae]$	memory read
	$\mid\ \mathtt{mem}[ae_0] := ae_1$	memory write
	$\mid\ \mathtt{jump}\ L$	jump to a label
	$\mid\ \mathtt{jump}\ [ae]$	indirect jump
	$\mid\ \mathtt{branch}\ ae\ L$	branch if non-zero
labels	L	
registers	$r ::= \mathbf{r}_0 \mid \cdots \mid \mathbf{r}_{n-1}$	
asm exprs	$ae ::= n \mid r \mid \&L \mid (op\ ae_0\ ae_1)$	
integers	n	
operators	$op ::= \mathtt{add} \mid \mathtt{sub} \mid \mathtt{sll} \mid = \mid \mathtt{<>} \mid < \mid \cdots$	

SAL has a very basic set of instructions, a set of registers, and a minimal set of expressions. Macro instructions in MIPS or x86 are translated into a sequence of SAL instructions; for example the jump-and-link instruction in MIPS is translated as follows:

MIPS	SAL
`jal fun`	$\mathbf{r}_{ra} := \&\mathtt{retaddr0}$
	`jump fun`
	`retaddr0:`

Second, because the abstract interpretation lattice is defined with respect to a particular class hierarchy, Coolaid

[4] "Foundationalizing" Coolaid is discussed elsewhere [Sch04].

needs access to that information for the program being verified. In all, we will need the parent class of each user-defined class; the types of the fields of each class; and the input and output types of each method—all other needed facts must already be encoded in the data block of the compiled code in order to meet the conventions of Cool's run-time system. We access the missing data through *annotations* encoded as comments in the assembly code to be verified. (In reference to our experiments, note that we did not need to change any of the student compilers to obtain the required data, which can be trivially reconstructed from the source code.)

Finally, one key element of verification at the assembly code level is the run-time stack. The verifier must maintain an abstract state not only for registers but also for stack slots, and memory operations must be recognized as either stack accesses or heap accesses. Formally, the lattice of the abstract interpretation must be extended to handle stack frames and calling conventions. Values may be typed as stack pointers, or as the saved values of callee-save registers or the return address—the return instruction, which is just an indirect jump in SAL, must verifiably jump to the correct return address. We must even keep track of the lowest accessed stack address in order to ensure that no stack overflows can occur or that operating system-based detection mechanisms cannot be subverted (e.g., skipping over the guard page—an unmapped page of memory that separates the stack region from the heap). Fortunately, the Open Verifier framework allows Coolaid to work cooperatively with a separate, modular verifier to handle these issues, which we will not discuss further in this paper.[5]

6. EDUCATIONAL EXPERIENCE

Coolaid includes an interactive GUI that allows the user to step through the verification process, while seeing the inferred abstract value and type for each state element. Stepping back in the verification is also possible and is useful to investigate how an unexpected abstract value originated.

We used Coolaid in the undergraduate compiler course at UC Berkeley in the Spring 2004 semester. Our experiments had two main purposes. First, we wanted to test, in a controlled setting, the hypothesis that such a tool is a useful compiler-debugging aid. Second, we wanted to give the students hands-on experience with how data-flow algorithms can be used not just for compiler optimizations, but also for checking software properties. Before starting to use Coolaid, the students attended a lecture presenting how global data-flow analysis can be adapted to the purpose of type-checking low-level languages, starting with a JVML-like language and ending with assembly language.

Each semester about 150 students take the compiler class. Over the course of a semester, the students work in pairs to build a complete Cool compiler emitting MIPS assembly language. The students are supposed to construct test cases for their compilers and to run the tests using the SPIM [Lar94] simulator. An automated version of this testing procedure, with 49 tests, is used to compute a large fraction of their project grade.

In the Spring 2004 semester, the students were given access to Coolaid. They still had to write their Cool test cases,

[5]The verifier for handling the run-time stack is interesting in itself, and we intend to publish details separately; in the context of foundational verification, it is discussed briefly in Schneck [Sch04]

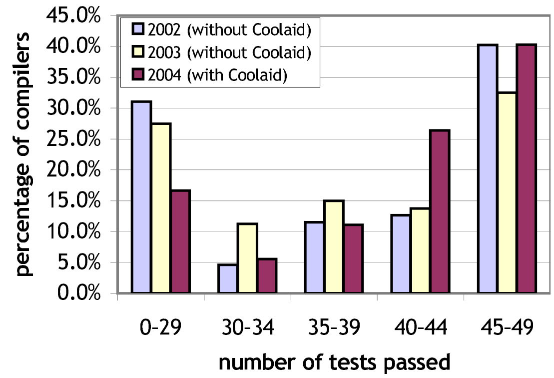

Figure 3: Performance of student compilers with and without Coolaid. The compilers are binned based on letter grades (for the automated testing component).

but the validation of a test could also be done by Coolaid, not simply by matching SPIM output with the expected output. We made a convincing case to the students that Coolaid not only can expose compilation bugs that simple execution with SPIM might not cover, but can also pinpoint the offending instruction, as opposed to simply producing the wrong SPIM output.

In order to make interesting comparisons, we have applied Coolaid retro-actively to the projects built in the 2002 and 2003 instances of the course when students did not have Coolaid available. Each class was asked to complete the same project in the same amount of time.

6.1 Results

First, we ran each compiler on the 49 tests used for grading. The number of compilers varied from year to year as follows:

2002: 87 compilers, 4263 compiled test cases
2003: 80 compilers, 3920 compiled test cases
2004: 72 compilers, 3528 compiled test cases

Figure 3 shows a histogram of how many compilers passed how many tests, with the numbers adjusted proportionally to the difference in the number of compilers each year. This data indicates that students who had access to Coolaid produced better compilers. In particular, the mean score of each team (out of 49) increased from 33 (67%) in 2002 or 34 (69%) in 2003 to 39 (79%) in 2004. This would be a measure of software quality when compilation results are run and checked against expected output (the traditional compiler testing method). Grade-wise, this is almost a letter grade improvement in their raw score.

Next, we compared the traditional way of testing compilers with using Coolaid to validate the compilation result. Each compiler result falls into one of the following categories:

- The code produces correct output and also passes Coolaid (i.e., the compilation is correct as far as we can determine).
- The code produces incorrect output and also fails Coolaid (i.e., the error is visible in the test run). This category also includes cases where the compiler crashes during code generation.
- The code produces correct output but fails Coolaid.

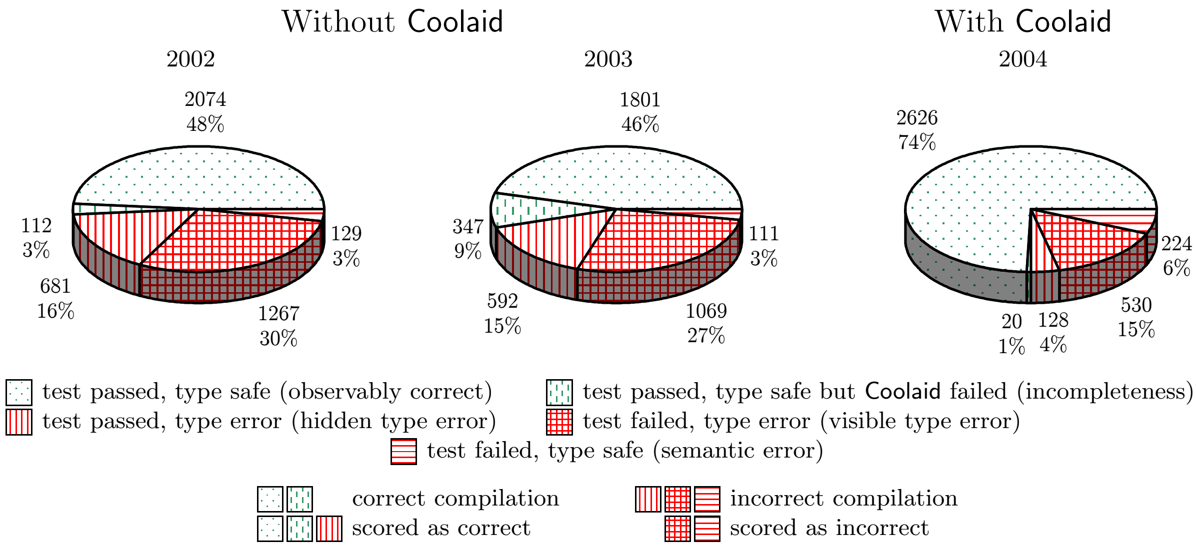

Figure 4: Behavior of test programs compiled by students and run through both the standard execution-based testing procedure and Coolaid. Horizontal lines indicate failing the standard testing procedure, while vertical lines (dotted or dashed) indicate failing Coolaid (and thus the grid pattern indicates failing both).

Typically, this indicates a compilation error resulting in ill-typed code that is not exercised sufficiently by its particular hard-wired input (). However, this case can also indicate a *Coolaid incompleteness*: a valid compilation strategy that Coolaid is unable to verify (). In order to correctly classify compilation results in this case, we have inspected them manually.

Examples of incompletenesses included using odd calling conventions (such as requiring the frame pointer be callee-save only for initialization methods) and implementing case statements by a lookup table rather than a nested-if structure. Coolaid could be changed to handle such strategies, but it is impossible to predict all possible strategies in advance.

The code produces incorrect output but passes Coolaid.

This indicates a semantic error: type-safe code that does not correspond to the semantics of the Cool source. An example of such an error would be swapping the order of operands in a subtraction. In principle, it could also indicate a *Coolaid unsoundness*: an unsafe compilation strategy that Coolaid incorrectly verifies. In fact, one surprising unsoundness was discovered and fixed while checking the student compilers: Coolaid was allowing too broad an interface to a particular run-time function. This could be prevented by wrapping Coolaid into a foundational verifier producing proofs of safety, which is work in progress as part of the Open Verifier project [CCNS05, Sch04].

The breakdown of behaviors for the code produced by the student compilers is shown in Figure 4. Observe that the percentage of compilations in each category are roughly the same in 2002 and 2003 when students did not have Coolaid despite the variance in the student population.

Several conclusions can be drawn from this data, at least as it concerns compilers in early development stages. To make our calculations clear, we will include parenthetical references to the patterns used in Figure 4.

The majority of compiler bugs lead to type errors. When the students did not have Coolaid (2002 and 2003 combined), 91% of all the failed test cases were also ill-typed; when students did have Coolaid (2004), the percentage was still 70% (/). Moreover, there are a significant number of compilation errors that are hard to catch with traditional testing. In 2002 and 2003, 16% of the tests had errors and were ill-typed, but they passed traditional validation. In 2004, that number decreased to 4%, presumably because students had access to Coolaid (/total).

Students using Coolaid create compilers that produce more type-safe programs. The percentage of compiled test cases with type errors decreased from 44% to 19% (/total). Even if we only count test cases that also produced incorrect output, there is still a decrease from 29% to 15% (/total).

On the negative side, type-checking might impose unnecessary constraints on the code generation. In 2002 and 2003, 6% of the test cases are valid, but do not type-check (/total). We note that in most cases the problem involves calling conventions in such a way that either the compiler or Coolaid could be trivially modified to avoid the problem; still, about 3% of the compiled test cases exhibit some apparently non-trivial incompleteness. This number decreased to less than 1% in 2004, presumably because students preferred to adapt their compilation scheme to Coolaid, in order to silence these false alarms. This may indicate that the tool is limiting student ingenuity. We hope to ameliorate this problem by incorporating into Coolaid the ability to handle unusual strategies used in past years. We are also exploring the possibility of having a general type of lookup tables, a feature in many unhandled compilation strategies. However, until undergraduate compilers students are ready to design certifying compilers, there is no completely general solution.

Overall, there was a slight increase (from 3% to 6% of all test cases) in programs that were type-safe but had semantic errors (/total). There is a potential concern here; what if students using Coolaid to debug do not perform sufficient testing for the semantic bugs that Coolaid does not catch? Although in all cases it seems likely that students do not sufficiently test their compilers, we do not believe

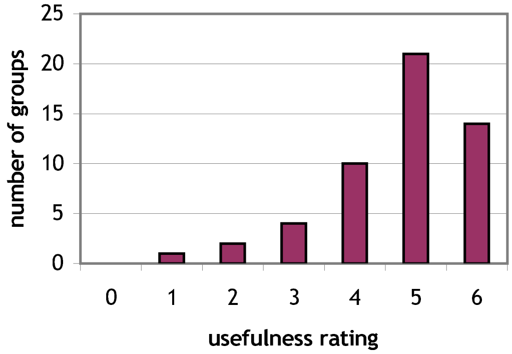

Figure 5: Student feedback from the 2004 class about the usefulness of Coolaid. 0 means "counter-productive" and 6 means "can't imagine developing/debugging a compiler without it."

that Coolaid particularly exacerbates this problem. Instead, we suspect that often purely semantic bugs are masked by additional type errors for the students without Coolaid. In any case, this increase seems rather small compared to the overall benefits of reducing type errors.

Student Feedback. As a final data point, the students in 2004 were asked to submit feedback about Coolaid, including a numeric rating of its usefulness. 52 of the 72 teams returned feedback; the results are in Figure 5.

Common negative comments tended to involve either details about the user interface, or the fact that Coolaid would not catch semantic errors that were not type errors. A favorite positive comment:

> *"I would be totally lost without Coolaid. I learn best when I am using it hands-on . . . . I was able to really understand stack conventions and optimizations and to appreciate them."*

7. RELATED WORK

Proof-carrying code [Nec97] and typed-assembly languages [MWCG99] also check memory safety of programs at the machine code level. Both traditional and more recent approaches [AF00, HST$^+$02, Cra03] focus more on generality than accessibility; their technical developments are quite involved. A wider audience can use Coolaid or Coolaid-like verifiers for compiler debugging or reinforcing compiler concepts.

Note that while the type system is more complex than for bytecode verification, it is fairly simple compared to traditional encodings of object-oriented languages into functional typed-assembly languages. This simplification is obtained by specializing to the Cool object layout and other conventions. While this sacrifices some bit of generality, it appears more feasible in the context of retrofitting existing compilers. Furthermore, we assert that encoding object-oriented languages in functional TALs may be unnatural, much like the compilation of functional languages to object-oriented intermediate languages, like the JVML; others seem to concur [CT05]. We might hope to recover some generality, yet maintain some simplicity, by moving towards an "object-oriented TAL". A design decision in [LST02] to change the compilation scheme of the type-case rather than introduce a new tag type (which they found possible but difficult in their system) provides some additional evidence for the usefulness of such a system.

Prior work has used several strategies to serve the function that our dependent types do. TALs have traditionally modeled a limited class of relationships between values using parametric polymorphism. Singleton types provide another mechanism, and the design of TALs with more complicated dependent type systems has been investigated [XH01]. League *et al.* [LST02, LST03] use existential types. A key difference of our work compared to the work mentioned above using typed-assembly or typed-intermediate languages is that we elide many more typing annotations using verification-time inference.

8. CONCLUSION

We describe in this paper how to extend data-flow based type checking of intermediate languages to work at the level of assembly language, while maintaining the ability to work without annotations inside a procedure body. The additional cost is that the checker must maintain a lattice of dependent types, which are designed to match a particular representation of run-time data structures. While this increases the complexity of the algorithm, it has the advantage that it enables the use of type-checking technology for debugging native-code compilers, not just those that produce bytecode. Furthermore, the ability to infer types reduces greatly the demands on the compiler to generate annotation, thus enabling the technique for more compilers. In fact, we were able to use the technique on existing compilers without any modifications.

We consider our experiments in the context of an undergraduate compiler class to be very successful. We found that data-flow based verification fits very well with other concepts typically covered in a compiler class (e.g., types, data-flow analysis). At the same time, it introduces students to the idea that language-based tools can be effective for improving software quality and safety. Furthermore, packaging these ideas into a tool whose interface resembles a debugger allows students to experiment hands-on with important concepts, such as data-flow analysis and types.

While our results are fairly convincing for the case of early-development compilers, it is not clear at all how they apply to mature compilers. We expect that a smaller ratio of compiler bugs result in errors that could be caught by type checking. Nevertheless, the arguments would be strong to include type checking in the standard regression testing procedure, if only for its ability to pinpoint otherwise often hard to find type-safety errors very precisely.

There were certain cases when Coolaid could not keep up with correct compilation strategies or optimizations. While this is not a big issue for bytecode compilers, because the bytecode language can express very few optimizations, it becomes a serious issue for native code compilers where representation choices have a big effect on the code generation strategy. In the context of the Open Verifier project, we are working on ways that would allow a compiler developer to specify, at a high-level, alternative compilation strategies, along with proofs of soundness with respect to the existing compilation strategies [CCNS05].

Acknowledgments. We would like to thank Kun Gao for his hard work on the implementation of Coolaid and the anonymous referees for providing helpful comments on drafts of this paper.

9. REFERENCES

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. of the 27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 243–253, January 2000.

[Aik96] Alexander Aiken. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7):19–24, July 1996.

[BCM+93] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Proc. of the 8th Annual ACM Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 29–46, October 1993.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 234–252, January 1977.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The Open Verifier framework for foundational verifiers. In *Proc. of the 2nd ACM Workshop on Types in Language Design and Implementation (TLDI'05)*, January 2005.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Proc. of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, January 2005.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proc. of the ACM 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 95–107, May 2000.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Proc. of the 30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 198–212, January 2003.

[CT05] Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *Proc. of the 32nd ACM Symposium on Principles of Programming Languages (POPL'05)*, January 2005.

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 248–260, January 2001.

[GTN04] Sumit Gulwani, Ashish Tiwari, and George C. Necula. Join algorithms for the theory of uninterpreted functions. In *Proc. of the 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, LNCS, December 2004.

[HST+02] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, July 2002.

[Lar94] J. R. Larus. Assemblers, linkers, and the SPIM simulator. In *Computer Organization and Design: The Hardware/Software Interface*, Appendix A. Morgan Kaufmann, 1994.

[Ler03] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[LST02] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 24(2):112–152, 2002.

[LST03] Christopher League, Zhong Shao, and Valery Trifonov. Precision in practice: A type-preserving Java compiler. In *Proc. of the 12th International Conference on Compiler Construction (CC'03)*, April 2003.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[MWCG99] 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.

[Nec97] George C. Necula. Proof-carrying code. In *Proc. of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, January 1997.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Proc. of the ACM 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, June 2000.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Proc. of 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 151–166, March 1998.

[RM99] Martin Rinard and Darko Marinov. Credible compilation. In *Proc. of the Run-Time Result Verification Workshop*, July 1999.

[Sch04] Robert R. Schneck. *Extensible Untrusted Code Verification*. PhD thesis, University of California, Berkeley, May 2004.

[XH01] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *Proc. of the International Conference on Functional Programming (ICFP'01)*, pages 169–180, September 2001.

APPENDIX

A. ABSTRACT TRANSITION AND TYPING

In this section, we collect the rules that define the type-based abstract interpreter. Most of the rules are described through example in Section 4.

In Figure 6, we define the abstract transition judgment $I: \langle \Sigma \fatsemi \Gamma \rangle_p \rightarrow \langle \Sigma' \fatsemi \Gamma' \rangle_{p'}$. To treat method calls uniformly and concisely, the abstract transition rules that apply to indirect jumps also apply to direct jumps, viewing `jump` L as `jump` $[\&L]$. Moreover, we assume that an initialization phase populates the initial Γ with types for the code label of each method (i.e, with $\mathsf{smeth}(C, n)$), the code label of each initialization method (i.e., with $\mathsf{sinit}(C)$), and the data label for statically allocated objects (i.e, with $\mathsf{nonnull\ exactly}\ C$). The $\mathsf{refinetag}_F$ and $\mathsf{refinetag}_T$ rules use the auxiliary function $\mathit{taglub}(N)$, which yields the class that is the least upper bound in the class hierarchy given a set of class tags N.

The typing judgment $\Gamma \vdash e : \tau \rhd \Gamma'$ is defined in Figure 7. Most types are assigned to symbolic values (and carried in Γ), so the additional types that are derived by the typing judgment are for assigning types to normalized addresses. The function $\mathit{tagof}(C)$ gives the tag for class C.

$$\boxed{I : \langle \Sigma \fatsemi \Gamma \rangle_p \to \langle \Sigma' \fatsemi \Gamma' \rangle_{p'}}$$

$$\frac{}{L\colon : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma \rangle}\ \textsf{label} \qquad \frac{(L \text{ is not a code label for a method})}{\texttt{jump}\ L : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma \rangle_L}\ \textsf{jump} \qquad \frac{}{r := ae : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma[r \mapsto \Sigma(ae)] \fatsemi \Gamma \rangle}\ \textsf{assign}$$

$$\frac{\Gamma \vdash \Sigma(ae) : \tau\ \textsf{ptr} \rhd \Gamma' \quad (\alpha \text{ fresh})}{r := \texttt{mem}[ae] : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma[r \mapsto \alpha] \fatsemi \Gamma'[\alpha \mapsto \tau] \rangle}\ \textsf{read} \qquad \frac{\Gamma \vdash \Sigma(ae_0) : \tau\ \textsf{ptr} \rhd \Gamma' \quad \Gamma' \vdash \Sigma(ae_1) : \tau \rhd \Gamma''}{\texttt{mem}[ae_0] := ae_1 : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma'' \rangle}\ \textsf{write}$$

$$\frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) : \textsf{meth}(\alpha, n) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \alpha \qquad \Gamma' \vdash \alpha : \textsf{nonnull}\ C \rhd \Gamma'' \\ T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_k)\ \ldots\ \} \\ \Gamma'' \vdash \Sigma(\mathbf{r}_{arg_1}) : \tau_1 \rhd \Gamma''_1 \quad \cdots \quad \Gamma''_{k-1} \vdash \Sigma(\mathbf{r}_{arg_k}) : \tau_k \rhd \Gamma''_k \\ (\Sigma', \beta \text{ fresh}) \\ (m \text{ is the method at offset } n \text{ of class } C) \end{array}}{\texttt{jump}\ [ae] : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma'[\mathbf{r}_{rv} \mapsto \beta] \fatsemi \Gamma''_k[\beta \mapsto \tau_{rv}] \rangle}\ \textsf{meth}$$

$$\frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) : \textsf{smeth}(C, n) \rhd \Gamma' \\ \Gamma' \vdash \Sigma(\mathbf{r}_{arg_0}) : \textsf{nonnull}\ C \rhd \Gamma'' \\ T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_k)\ \ldots\ \} \\ \Gamma'' \vdash \Sigma(\mathbf{r}_{arg_1}) : \tau_1 \rhd \Gamma''_1 \quad \cdots \quad \Gamma'' \vdash \Sigma(\mathbf{r}_{arg_k}) : \tau_k \rhd \Gamma''_k \\ (\Sigma', \beta \text{ fresh}) \\ (m \text{ is the method at offset } n \text{ of class } C) \end{array}}{\texttt{jump}\ [ae] : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma'[\mathbf{r}_{rv} \mapsto \beta] \fatsemi \Gamma''_k[\beta \mapsto \tau_{rv}] \rangle}\ \textsf{smeth}$$

$$\frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) : \textsf{init}(\alpha) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \beta \qquad \Gamma' \vdash \beta : \textsf{nonnull classof}(\alpha) \rhd \Gamma'' \\ (\Sigma' \text{ fresh}) \end{array}}{\texttt{jump}\ [ae] : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma'[\mathbf{r}_{arg_0} \mapsto \beta] \fatsemi \Gamma'' \rangle}\ \textsf{init} \qquad \frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) : \textsf{sinit}(C) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \beta \qquad \Gamma' \vdash \beta : \textsf{nonnull}\ C \rhd \Gamma'' \\ (\Sigma' \text{ fresh}) \end{array}}{\texttt{jump}\ [ae] : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma'[\mathbf{r}_{arg_0} \mapsto \beta] \fatsemi \Gamma'' \rangle}\ \textsf{sinit}$$

$$\frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ 0 \rhd \Gamma' \quad \Gamma'(\alpha) = b \quad R \in \{=, \neq\} \\ \tau = \begin{cases} \textsf{nonnull}\ b & \text{if } \neg(\alpha\ R\ 0) \equiv \alpha \neq 0 \\ \textsf{null} & \text{otherwise} \end{cases} \end{array}}{\texttt{branch}\ ae\ L : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \tau] \rangle}\ \textsf{nullcheck}_F \qquad \frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ 0 \rhd \Gamma' \quad \Gamma'(\alpha) = b \quad R \in \{=, \neq\} \\ \tau = \begin{cases} \textsf{nonnull}\ b & \text{if } \alpha\ R\ 0 \equiv \alpha \neq 0 \\ \textsf{null} & \text{otherwise} \end{cases} \end{array}}{\texttt{branch}\ ae\ L : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \tau] \rangle_L}\ \textsf{nullcheck}_T$$

$$\frac{\Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ k \rhd \Gamma' \quad \Gamma'(\alpha) = \textsf{tag}(\beta, N) \quad \Gamma'(\beta) = \textsf{nonnull}\ C \quad N' = \{n \in N \mid \neg(n\ R\ k)\} \neq \emptyset \quad R \in \{=, \neq, <, \leq, >, \geq\}}{\texttt{branch}\ ae\ L : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \textsf{tag}(\beta, N')][\beta \mapsto \textsf{nonnull}\ \mathit{taglub}(N')] \rangle}\ \textsf{refinetag}_F$$

$$\frac{\Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ k \rhd \Gamma' \quad \Gamma'(\alpha) = \textsf{tag}(\beta, N) \quad \Gamma'(\beta) = \textsf{nonnull}\ C \quad N' = \{n \in N \mid n\ R\ k\} \neq \emptyset \quad R \in \{=, \neq, <, \leq, >, \geq\}}{\texttt{branch}\ ae\ L : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \textsf{tag}(\beta, N')][\beta \mapsto \textsf{nonnull}\ \mathit{taglub}(N')] \rangle_L}\ \textsf{refinetag}_T$$

$$\frac{(\text{not a null-check nor a type refinement})}{\texttt{branch}\ ae\ L : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma \rangle}\ \textsf{branch}_F \qquad \frac{(\text{not a null-check nor a type refinement})}{\texttt{branch}\ ae\ L : \langle \Sigma \fatsemi \Gamma \rangle \to \langle \Sigma \fatsemi \Gamma \rangle_L}\ \textsf{branch}_T$$

Figure 6: Abstract transition rules.

$$\boxed{\Gamma \vdash e : \tau \rhd \Gamma'}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha \rhd \Gamma' \quad \Gamma'(\alpha) = \tau}{\Gamma \vdash e : \tau \rhd \Gamma'} \qquad \frac{\Gamma \vdash e : \tau' \rhd \Gamma' \quad \Gamma' \vdash \tau' <: \tau}{\Gamma \vdash e : \tau \rhd \Gamma'}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha + n \rhd \Gamma' \quad \Gamma' \vdash \alpha : \textsf{nonnull}\ C \rhd \Gamma'' \quad T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau\ f\ \ldots\ \} \quad (f \text{ is the field at offset } n \text{ of class } C)}{\Gamma \vdash e : \tau\ \textsf{ptr} \rhd \Gamma'}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha + 8 \rhd \Gamma' \quad \Gamma' \vdash \alpha : \textsf{nonnull}\ C \rhd \Gamma''}{\Gamma \vdash e : \textsf{disp}(\alpha)\ \textsf{ptr} \rhd \Gamma''} \qquad \frac{\begin{array}{ll} \Gamma \vdash e \Downarrow \beta + n \rhd \Gamma' & \Gamma'' \vdash \alpha : \textsf{nonnull}\ C \rhd \Gamma''' \\ \Gamma' \vdash \beta : \textsf{disp}(\alpha) \rhd \Gamma'' & (C \text{ has a method at offset } n) \end{array}}{\Gamma \vdash e : \textsf{meth}(\alpha, n)\ \textsf{ptr} \rhd \Gamma'''}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha + 8 \rhd \Gamma' \quad \Gamma' \vdash \alpha : \textsf{nonnull exactly}\ C \rhd \Gamma''}{\Gamma \vdash e : \textsf{sdisp}(C)\ \textsf{ptr} \rhd \Gamma''} \qquad \frac{\begin{array}{l} \Gamma \vdash e \Downarrow \beta + n \rhd \Gamma' \\ \Gamma' \vdash \beta : \textsf{sdisp}(C) \rhd \Gamma'' \qquad (C \text{ has a method at offset } n) \end{array}}{\Gamma \vdash e : \textsf{smeth}(C, n)\ \textsf{ptr} \rhd \Gamma''}$$

$$\frac{\Gamma \vdash e \Downarrow \&\texttt{init\_table} + 4 \cdot \beta + 4 \rhd \Gamma' \quad \Gamma' \vdash \beta : \textsf{tag}(\alpha, N) \rhd \Gamma''}{\Gamma \vdash e : \textsf{init}(\alpha)\ \textsf{ptr} \rhd \Gamma''} \qquad \frac{\Gamma \vdash e \Downarrow \alpha \rhd \Gamma' \quad \Gamma'(\alpha) = \textsf{nonnull}\ C}{\Gamma \vdash e : \textsf{tag}(\alpha, \{n \mid n = \mathit{tagof}(C') \wedge \Gamma' \vdash C' <: C\})\ \textsf{ptr} \rhd \Gamma'}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha \rhd \Gamma' \quad \Gamma'(\alpha) = \textsf{nonnull exactly}\ C}{\Gamma \vdash e : \textsf{tag}(\alpha, \{\mathit{tagof}(C)\})\ \textsf{ptr} \rhd \Gamma'} \qquad \frac{\Gamma \vdash e \Downarrow \alpha \rhd \Gamma' \quad \Gamma'(\alpha) = \textsf{nonnull classof}(\beta) \quad \Gamma \vdash \beta : \textsf{tag}(\gamma, N)\ \textsf{ptr} \rhd \Gamma'}{\Gamma \vdash e : \textsf{tag}(\gamma, N)\ \textsf{ptr} \rhd \Gamma'}$$

Figure 7: Typing rules.