Type-Based Verification of Assembly Language

by Bor-Yuh Evan Chang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor George C. Necula
Research Advisor

(Date)

* * * * * * *

Professor Rastislav Bodik
Second Reader

(Date)

Type-Based Verification of Assembly Language

Bor-Yuh Evan Chang

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.

Contents

List of Figures

Acknowledgments

Part of the work presented here has been published in a paper co-authored with Adam Chlipala, George C. Necula, and Robert R. Schneck [CCNS05b]. Some of the text from that manuscript has been adapted for use in this report.

In addition to above mentioned co-authors, we would like to thank Kun Gao for his work on the GUI for Coolaid and other parts of the Open Verifier infrastructure.

I would like to thank Ras Bodik for the helpful comments on this manuscript, and I would especially like to thank my advisor, George Necula, for all the guidance he has generously provided.

Chapter 1

Introduction

Effective tools for automatically verifying properties of low-level code has several applications, including enforcing the safety of mobile code and debugging compilers (particularly, complicated optimizing ones). It is a widely held belief that such checking of compiler output greatly aids the debugging process. 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] can be used to check 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 type-checking 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. A 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. Native-code compilers are 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 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 Chapter 2, we present the main ideas of the design of Coolaid, our assembly-level verifier for the Java-like classroom language Cool. Section 2.1 gives an overview of the challenges in building such a verifier, while a formalization of the verification algorithm is introduced in Sections 2.2–2.3. In Chapter 3, we give further details necessary for a complete description of the verifier. We then describe our results and experience using Coolaid in the compiler class in Chapter 4. Finally, we discuss related work and conclude in Chapter 5.

Chapter 2

Concept

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, and self-type polymorphism [BCM+93]. We have also extended Cool with exceptions. 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.

In Section 2.1, we give an overview of Coolaid, along with the challenges in developing such an assembly-level verifier. Then, we formalize the abstraction employed by Coolaid in Section 2.2. In Section 2.3, we present the verification algorithm through example. Finally, we give an example that illustrates one of the novel aspects of our design in Section 2.4.

2.1 Challenges

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 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*.

Suppose we compile Cool to JVML, and consider first the bytecode verifier for JVML. 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) $\bot$ 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 $\bot$ (*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:

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

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

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

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. That is, S is a stack of types (given by $S ::= \cdot \mid \tau :: S$), and R is a finite map from registers to types. We write $bc\colon \langle S, R\rangle_p \rightarrow_{\text{BV}} \langle S', R'\rangle_{p'}$ 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`, which is the bytecode for a virtual method dispatch:

$$\frac{\tau <: C \quad \tau_0' <: \tau_0 \quad \cdots \quad \tau_{n-1}' <: \tau_{n-1}}{\begin{array}{l}\texttt{invokevirtual}\ C.m(\tau_0, \tau_1, \ldots, \tau_{n-1}) : \tau_n \\ \qquad : \langle \tau_{n-1}' :: \cdots :: \tau_1' :: \tau_0' :: \tau :: S, R\rangle \rightarrow_{\text{BV}} \langle \tau_n :: S, R\rangle\end{array}}$$

where $C.m(\tau_0, \tau_1, \ldots, \tau_{n-1}) : \tau_n$ indicates a method m of class C with argument types $\tau_0, \tau_1, \ldots, \tau_{n-1}$ 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 with an assumption about return type (just like in type-checking). This assume-guarantee style reasoning is possible because the input and output types of each method are given in advance and provide sufficient pre- and post-conditions of the method for this particular abstraction.

The verification itself proceeds by symbolically executing the bytecode of each method using the abstract interpretation transition $\rightarrow_{\text{BV}}$. An abstract state is kept for each program point, initially the bottom abstract state 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

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

Figure 2.1: Computing by abstract interpretation the abstract state $\mathcal{R}_{\text{BV}}(p)$ at program point p. $\mathit{Instr}_{\text{BV}}(p)$ 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.

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. Thus, the abstract states are computed as the least fixed point of equations in Figure 2.1. 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) shown in Figure 2.2(a), along with the compilation of the method `Main.scan` to bytecode (JVML) in (b). We show below a computation of the abstract state at line 3.

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

The first time $\mathcal{R}_{\text{BV}}(3)$ is computed, τ is `SubSeq` and then the `invokevirtual` is okay because $\texttt{SubSeq} <: \texttt{Seq}$. However, this requires that $\mathcal{R}_{\text{BV}}(3)$ 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 3.1. 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 2.2, we describe the appropriate lattice of abstract states, and in Section 2.3, 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 2.2 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 3–9 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. According to the Cool run-time conventions, the dispatch table is located at offset 8 from an object reference, and in this case, the method is located at offset 12 in the dispatch table.

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 necessarily compiled into separate instructions, which then may be re-ordered

```
class Seq {
  int data;
  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:

   1:  aload_1           // load x from s

   3:  invokevirtual  Seq.next() : Seq
                     // (call x.next())






  10:  ifnonnull 3       // x != null

  12:  return
```

(b) JVML

```
0  Main.scan:
                  ⋮
1                 $\mathbf{r}_x := \mathbf{r}_s$
2  Loop:
3                 branch $(= \mathbf{r}_x\ 0)$ $L_{\texttt{dispatch\_abort}}$
4                 $\mathbf{r}_t :=$ mem[(add $\mathbf{r}_x$ 8)]
5                 $\mathbf{r}_t :=$ mem[(add $\mathbf{r}_t$ 12)]
6                 $\mathbf{r}_{arg_0} := \mathbf{r}_x$
7                 $\mathbf{r}_{ra} := \&L_{\texttt{ret}}$
8                 jump [$\mathbf{r}_t$]
9  $L_{\text{ret}}$:
10                branch $(= \mathbf{r}_{rv}\ 0)$ $L_{\texttt{done}}$
11                $\mathbf{r}_x := \mathbf{r}_{rv}$
12                jump Loop
```

(c) SAL

Figure 2.2: An example program shown at the source, bytecode, and assembly levels.

```
1       branch (= $\mathbf{r}_{child}$ 0) L$_{\texttt{dispatch\_abort}}$
2       $\mathbf{r}_t$ := mem[(add $\mathbf{r}_{child}$ 8)]
3       $\mathbf{r}_t$ := mem[(add $\mathbf{r}_t$ 12)]
4       $\mathbf{r}_{arg_0}$ := $\mathbf{r}_{parent}$
5       $\mathbf{r}_{ra}$ := &L$_{\texttt{ret}}$
6       jump [$\mathbf{r}_t$]
7 L$_{\text{ret}}$:
```

Figure 2.3: An example type-unsafe compilation to illustrate some difficulties in verifying assembly code.

due to instruction scheduling or other optimizations. This separation requires the typing of intermediate results (*e.g.*, dispatch tables) and tracking critical dependencies. To fully illustrate this issue, consider a bad compilation that violates type safety shown in Figure 2.3. In this example, 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. An initial implementation of an assembly-level checker using a strategy analogous to bytecode verification might assign the type $\mathbf{r}_t : \texttt{meth}(\texttt{Seq}, 12)$ 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 $\mathbf{r}_{arg_0} : \texttt{Seq}$, which succeeds since $\mathbf{r}_{parent} : \texttt{Seq}$. 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 now describe a type system to handle these difficulties.

2.2 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. To be less sensitive to the particular compilation strategy, we have found it useful to assign types lazily to intermediate values. That is, we 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

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

values to types:

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

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

2.2.1 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 3.1), 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] The values are as follows:

values	$v ::= n_0 \cdot \&L + n_1 \cdot \alpha + n_2 \mid \alpha \; R \; n$
relations	$R ::= = \mid \neq \mid < \mid \leq \mid > \mid \geq$

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. Typically, the factors n_0 and n_1 are either 0 or 1. 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 their values were obtained by replacing them with fresh symbolic values. 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 $\Gamma \vdash e \Downarrow v \rhd \Gamma'$ for the normalization of expression e to value v, yielding a possibly extended Γ for new symbolic values. In most cases, the new

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

symbolic value can be typed implicitly 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. It is fairly straightforward to define this normalization. We also lift normalization to value states, writing $\Gamma \vdash \Sigma \Downarrow \Sigma' \triangleright \Gamma'$ to mean normalizing each expression in Σ to values in Σ' for each register.

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 consider values as providing a (fancy) labeling of equivalence classes of registers. We write that $\langle \Sigma \mathbin{⨾} \Gamma \rangle \models r_0 = r_1$ to mean that the abstract state $\langle \Sigma \mathbin{⨾} \Gamma \rangle$ implies that registers r_0 and r_1 are equal, and define this as follows:

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

where the equality $\Sigma'(r_0) = \Sigma''(r_1)$ is structural equality for symbolic values. Informally, this statement simply says that $r_0 = r_1$ precisely when their contents normalize to the same value.

As noted above, the normalization proceeds in a type-directed manner in order to determine when the structure of a subexpression is irrelevant. Without types, this value analysis would have to be either weaker (in which case it may not be sufficient for our purposes) or leave more complicated normalized values; neither is particularly attractive. The typing judgment, in turn, depends on the normalization judgment to handle, for example, the stack slot issue discussed in the previous paragraph. This mutual dependency prompts the integration of this value analysis with the type inference.

2.2.2 Types

We use a (simple) dependent type system extending the non-dependent types used in bytecode verification. While we could imagine merging 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 for proving 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. This distinction is particularly useful for catching bugs. One could further distinguish word into words used as machine integers versus booleans and perhaps catch even more bugs.

types	$\tau ::=$	$\top$	unknown contents
		$\mid$ 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 reference 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 as that of the object denoted by another symbolic value (classof(α)). 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 shown below, Coolaid only uses single-level pointers (*i.e.*, C ptr but not C ptr ptr).

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

Dispatch Table and Method Types. For method dispatches, we have types for the dispatch table of an object (disp(α)) and a method obtained from such a dispatch table (meth(α, n)). A similar pair is defined for the dispatch table and meth-

[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.

ods 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. See Section 3.3.3 for additional details on how this type is used to check type-cases.

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

Exceptions. To verify exceptions, we require a type of exception frames $\mathsf{excframe}$ and a type for exception handlers of particular exception frames. Additional details on how these types are used are given in Section 3.3.4.

types	$\tau ::= \ldots$	
	$\mid \mathsf{excframe}$	exception frame
	$\mid \mathsf{handler}$	exception handler

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, which maps 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 $\mathsf{exactly}$, $\mathsf{nonnull}$ 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 $\mathsf{classof}$. 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 $\mathsf{nonnull}$. Observe that the structure of abstract states allows instances of $\mathsf{classof}$ 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.

2.2.3 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.

Because we consider values (*i.e.*, v) as a (fancy) labeling of equivalence classes of registers, the lattice of value states is the lattice of finite conjunctions of equality constraints among registers (ordered by logical implication). That is, a particular abstract state $\langle \Sigma \mathbin{;} \Gamma \rangle$ induces an (empty or finite) conjunction of equality constraints among the registers given by the $\langle \Sigma \mathbin{;} \Gamma \rangle \models r_0 = r_1$ judgment. 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 normalize each expression of the states to join so that we are only working with values. Note that the value forms thus largely determine the information that is preserved across join points. Let $A_0 = \langle \Sigma_0 \mathbin{;} \Gamma_0 \rangle$ and $A_1 = \langle \Sigma_1 \mathbin{;} \Gamma_1 \rangle$

denote these states, and let $A = \langle \Sigma \fatsemi \Gamma \rangle$ be the result of 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 according to the equations given below. 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:

$$\begin{aligned}
&\ulcorner \langle \alpha_0, \alpha_1 \rangle \urcorner \stackrel{\text{def}}{=} \beta \quad \text{where } \beta \text{ fresh and } \Gamma(\beta) = \langle \Gamma_0, \Gamma_0(\alpha_0) \rangle \sqcup_{<:} \langle \Gamma_1, \Gamma_1(\alpha_1) \rangle \\
&\ulcorner \langle n_0 \cdot \&L + n_1 \cdot \alpha_0 + n_2, n_0 \cdot \&L + n_1 \cdot \alpha_1 + n_2 \rangle \urcorner \stackrel{\text{def}}{=} n_0 \cdot \&L + n_1 \cdot \ulcorner \langle \alpha_0, \alpha_1 \rangle \urcorner + n_2 \\
&\ulcorner \langle \alpha_0 \; R \; n, \alpha_1 \; R \; n \rangle \urcorner \stackrel{\text{def}}{=} \ulcorner \langle \alpha_0, \alpha_1 \rangle \urcorner \; R \; n \\
&\ulcorner \langle v_0, v_1 \rangle \urcorner \stackrel{\text{def}}{=} \beta \quad \text{where } \beta \text{ fresh and } \Gamma(\beta) = \top \qquad\qquad \text{(otherwise)}
\end{aligned}$$

Note that each distinct pair of symbolic values maps to a fresh symbolic value. We write $\sqcup_{<:}$ for the join in the types lattice. For 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 an unknown word value α, and then subtracts that same word value α; but if a join point intervenes, the fact that the register contains a value that is α more than a pointer type may be forgotten. 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).

2.3 Example Verification

As in Section 2.1, we describe the verification procedure for compiled-Cool by the abstract transition relation

$$I\colon \langle \Sigma \mathbin{⨟} \Gamma \rangle_p \rightarrow \langle \Sigma' \mathbin{⨟} \Gamma' \rangle_{p'}.$$

As before, this determines a verification procedure by the fixed-point calculation over the equations analogous to those of Figure 2.1. In this section, we sketch some interesting cases of the abstract transition relation by following the verification of an example. A more precise formalization, along with details not covered by this example are given in Section 3.3. All the abstract transition and typing rules are then collected together in Appendix A.

We first consider in detail the assembly code in Figure 2.2(c), which performs a dynamic dispatch in a loop. Suppose the abstract state before line 1 is as follows:

$$\begin{array}{l}\langle \mathbf{r}_y = \alpha^1_y, \mathbf{r}_{self} = \alpha^1_{self} \mathbin{⨟} \\ \;\alpha^1_y : \texttt{SubSeq}, \alpha^1_{self} : \mathsf{nonnull}\ \texttt{Main} \rangle\end{array} \tag{2.1}$$

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 1 is a register to register move ($\mathbf{r}_x := \mathbf{r}_s$), we simply make $\mathbf{r}_x$ map to the same value as $\mathbf{r}_y$. This changes the abstract state to

$$\langle \mathbf{r}_x = \alpha^1_y, \mathbf{r}_y = \alpha^1_y, \ldots \mathbin{⨟} \ldots \rangle$$

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

$$\frac{}{r := ae\ :\ \langle \Sigma \mathbin{⨟} \Gamma \rangle \rightarrow \langle \Sigma[r \mapsto \Sigma(ae)] \mathbin{⨟} \Gamma \rangle}\ \textsf{set}$$

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

Line 2 does not affect the state, as labels are treated as no-ops.

$$\frac{}{L\colon\ :\ \langle \Sigma \mathbin{⨟} \Gamma \rangle \rightarrow \langle \Sigma \mathbin{⨟} \Gamma \rangle}\ \textsf{label}$$

We recognize line 3 as a null-check so that the abstract state in the false branch is

$$\langle \ldots \mathbin{⨟} \alpha^1_y : \mathsf{nonnull}\ \texttt{SubSeq} \rangle$$

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 must aliases (for they map to the same symbolic value). In general, the post states of a null-check are given as follows:

$$\frac{\Gamma \vdash \Sigma(ae) \Downarrow \alpha \; R \; 0 \rhd \Gamma' \quad \Gamma'(\alpha) = b \quad R \in \{=, \neq\} \qquad \tau = \begin{cases} \mathsf{nonnull}\ b & \text{if } \neg(\alpha \; R \; 0) \equiv \alpha \neq 0 \\ \mathsf{null} & \text{if } \neg(\alpha \; R \; 0) \equiv \alpha = 0 \end{cases}}{\texttt{branch}\ ae\ L : \langle \Sigma \mathbin{\text{;}} \Gamma \rangle_p \longrightarrow \langle \Sigma \mathbin{\text{;}} \Gamma'[\alpha \mapsto \tau] \rangle_{p+1}} \; \mathsf{nullcheck}_F$$

The true case is similar.

We recognize that line 4 loads the dispatch table of object α_y^1, and the abstract state afterwards is

$$\langle \mathbf{r}_t = \alpha_t^4, \ldots \mathbin{\text{;}} \alpha_t^4 : \mathsf{disp}(\alpha_y^1), \ldots \rangle$$

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:

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

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 4 looks up a dispatch table is

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

We determine that offset 8 of an object is a pointer to the dispatch table because knowledge of the Cool object layout is built into the typing rule and therefore into the verifier. Note this rule would apply even if $\Gamma(\alpha) = \mathsf{nonnull\ exactly}\ C$ through the use of a subsumption rule (the subsump rule in Section A.3.3).

Line 5 then looks up the appropriate method in the dispatch table, so the post state is

$$\langle \mathbf{r}_t = \alpha_t^5, \ldots \mathbin{\text{;}} \alpha_t^5 : \mathsf{meth}(\alpha_y^1, 12), \ldots \rangle$$

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

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

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).

The next two lines (6 and 7) set the first argument register (which is used to pass the receiver object) and the return address. The abstract state after line 7 is as follows (given by set):

$$\langle \mathbf{r}_{arg_0} = \alpha^1_y, \mathbf{r}_{ra} = \&\mathrm{L}_{\mathrm{ret}}, \ldots \mathbin{⨾} \ldots \rangle$$

Finally, in line 8, 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

$$\langle \mathbf{r}_{rv} = \alpha^8_{rv} \mathbin{⨾} \alpha^8_{rv} : \mathtt{Seq} \rangle$$

and the method dispatch transition rule is as follows:

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

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 $\mathbf{r}_{cs}$); see Section 3.2 for details. 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 self-type, then we take τ_{rv} to be $\mathsf{classof}(\alpha^1_{self})$ (*i.e.*, to have the same dynamic type as self).

Lines 9–11 are a label, null-check, and register to register move, as we have seen before, so the abstract state before line 12 is

$$\langle \mathbf{r}_x = \alpha^8_{rv}, \ldots \mathbin{⨾} \alpha^8_{rv} : \mathsf{nonnull}\ \mathtt{Seq}, \ldots \rangle$$

The jump instruction at line 12 loops back with the abstract transition given by

$$\frac{(L \text{ is not a code label for a method})}{\texttt{jump}\ L : \langle \Sigma \mathbin{⨾} \Gamma \rangle \longrightarrow \langle \Sigma \mathbin{⨾} \Gamma \rangle_L}\ \textsf{jump}$$

that does not modify the abstract state but makes it a predecessor of L. This weakens $\mathit{Pre}(2)$ so that the type of the value in $\mathbf{r}_x$ is `Seq`, 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.

The astute reader may observe that neither the run-time stack of activation records nor calling conventions are reflected in the above rules. In particular, in the method dispatch rule, arguments are referenced as registers when they might be passed on the stack (*e.g.*, all arguments on x86, after the fourth argument on MIPS, or after the sixth argument on Sparc), and the return address is not checked to point to the next instruction. The run-time stack and call-return abstraction is common to many compilation strategies, so we would like not to build-in such reasoning nor be concerned with such details at this level. Fortunately, we have a mechanism to modularize the handling of such issues into sub-verifiers and allow the higher-level Coolaid verifier to work cooperatively with them. Details about the handling of the run-time stack and the call-return abstraction are given in Section 3.2.

2.4 Lazy Typing

The previous section illustrates the use of dependent types to track dependencies between assembly instructions, resolving the potential soundness issue presented in Figure 2.3 of Section 2.1 where the object from which the dispatch table is found and the object that is passed as the receiver object were not checked to be the same. It does not, however, show the use of lazy typing to be less sensitive to re-orderings, for example, due to optimizations.

To see this, consider the example program fragment shown in Figure 2.4, with two corresponding compilations to assembly code. In the "unoptimized" version (b), a sequence of instructions corresponds directly to a source-level statement (lines 1–3 correspond to line i, lines 4–11 to line ii, and line 12 to line iii). The "optimized" version has eliminated an extra null check (line 4), scheduled the increment of `d` (line 12) earlier, and reused the arithmetic from line 2 for looking up the dispatch table (line 6). The first two changes do not pose any particular difficulties, as we have types for intermediate results and are interpreting each instruction individually, while the last change necessitates the lazy typing of intermediate values.

```
    Seq s;
i   int d = s.data;
ii  s.next();
iii int x = d + 1;
```

(a) Cool

```
1        branch $(= \mathbf{r}_s\ 0)$ $\mathrm{L}_{\texttt{abort}}$
2        $\mathbf{r}_t$ := (add $\mathbf{r}_s$ 12)
3        $\mathbf{r}_d$ := mem[$\mathbf{r}_t$]
4        branch $(= \mathbf{r}_s\ 0)$ $\mathrm{L}_{\texttt{abort}}$
5        $\mathbf{r}_t$ := (add $\mathbf{r}_s$ 8)
6        $\mathbf{r}_t$ := mem[$\mathbf{r}_t$]

7        $\mathbf{r}_t$ := mem[(add $\mathbf{r}_t$ 12)]
8        $\mathbf{r}_{arg_0}$ := $\mathbf{r}_s$
9        $\mathbf{r}_{ra}$ := &$\mathrm{L}_{\texttt{ret}}$
10       jump [$\mathbf{r}_t$]
11  $\mathrm{L}_{\mathrm{ret}}$:
12       $\mathbf{r}_x$ := (add $\mathbf{r}_d$ 1)
```

(b) “Unoptimized”

```
1        branch $(= \mathbf{r}_s\ 0)$ $\mathrm{L}_{\texttt{abort}}$
2        $\mathbf{r}_t$ := (add $\mathbf{r}_s$ 12)
3        $\mathbf{r}_d$ := mem[$\mathbf{r}_t$]


6        $\mathbf{r}_t$ := mem[(sub $\mathbf{r}_t$ 4)]
12       $\mathbf{r}_x$ := (add $\mathbf{r}_d$ 1)
7        $\mathbf{r}_t$ := mem[(add $\mathbf{r}_t$ 12)]
8        $\mathbf{r}_{arg_0}$ := $\mathbf{r}_s$
9        $\mathbf{r}_{ra}$ := &$\mathrm{L}_{\texttt{ret}}$
10       jump [$\mathbf{r}_t$]
11  $\mathrm{L}_{\mathrm{ret}}$:
```

(c) “Optimized”

Figure 2.4: An example program fragment demonstrating the need for lazy typing of The class definition of `Seq` is given in Figure 2.2.

Now, suppose types were assigned “eagerly” in the verification of the “optimized” version, so after line 2, we have that

$$\mathbf{r}_t : \mathit{int}\ \mathsf{ptr}$$

by the following typing rule for pointers to fields:

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

In this case, the int type comes from the declared type of `data` field of class `Seq`.

Now at line 6, we cannot assign a type to the expression (sub $\mathbf{r}_t$ 4), as the only information we have is that it is the word before a pointer to an int. Of course, in general, any type of value may reside in the word before a pointer to an int. In contrast, the abstract state after line 2 that we maintain is

$$\langle \Sigma \mathbin{⨾} \Gamma \rangle = \langle \mathbf{r}_t = \alpha_s + 12, \mathbf{r}_s = \alpha_s, \ldots \mathbin{⨾} \alpha_s : \mathsf{nonnull}\ \mathsf{Seq}, \ldots \rangle\ .$$

Then, we can recognize line 6 as looking up the dispatch table using a trivial normalization

$$\Gamma \vdash (\alpha_s + 12) - 4 \Downarrow \alpha_s + 8 \rhd \Gamma \,.$$

In full, we obtain the following derivation for the read transition:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash (\alpha_s + 12) - 4 \Downarrow \alpha_s + 8 \rhd \Gamma} \qquad \dfrac{\Gamma(\alpha_s) = \mathsf{nonnull\ Seq}}{\Gamma \vdash \alpha_s : \mathsf{nonnull\ Seq} \rhd \Gamma}\;\mathsf{var}}{\Gamma \vdash (\alpha_s + 12) - 4 : \mathsf{disp}(\alpha_s)\ \mathsf{ptr} \rhd \Gamma}\;\mathsf{dispptr} \qquad (\alpha_t\ \text{fresh})}{\mathbf{r}_t := \mathtt{mem}[(\mathtt{sub}\ \mathbf{r}_t\ 4)] \,:\, \langle \Sigma \fatsemi \Gamma \rangle \longrightarrow \langle \Sigma[\mathbf{r}_t \mapsto \alpha_t] \fatsemi \Gamma[\alpha_t \mapsto \mathsf{disp}(\alpha_s)] \rangle}\;\mathsf{read}$$

Chapter 3

Details

In this chapter, we present additional details required for a complete presentation. We first fill-in the details of the assembly language and the Cool object layout that have been alluded to in the previous chapter in Section 3.1. Then, in Section 3.2, we describe the handling of the run-time stack that has been elided thus far. In Section 3.3, we describe the abstract transition and typing rules for additional language features not covered in Section 2.3. Finally, we discuss some details regarding initializing the verification procedure in Section 3.4.

3.1 Preliminaries

SAL. Coolaid is implemented on top of the Open Verifier framework for foundational verifiers [CCNS05a, Sch04], which provides an infrastructure for abstract interpretation on assembly code (among other things). 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 given in Figure 3.1.

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`	$\mathbf{r}_{ra}$:= `&retaddr0`
	`jump fun`
	`retaddr0:`

instructions	$I ::= L:$	label
	$\mid r := ae$	assignment
	$\mid r := \texttt{mem}[ae]$	memory read
	$\mid \texttt{mem}[ae_0] := ae_1$	memory write
	$\mid \texttt{jump}\ L$	jump to a label
	$\mid \texttt{jump}\ [ae]$	indirect jump
	$\mid \texttt{branch}\ ae\ L$	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	$op ::= \texttt{add} \mid \texttt{sub} \mid \texttt{sll} \mid = \mid <> \mid < \mid \cdots$	

Figure 3.1: Abstract syntax of SAL.

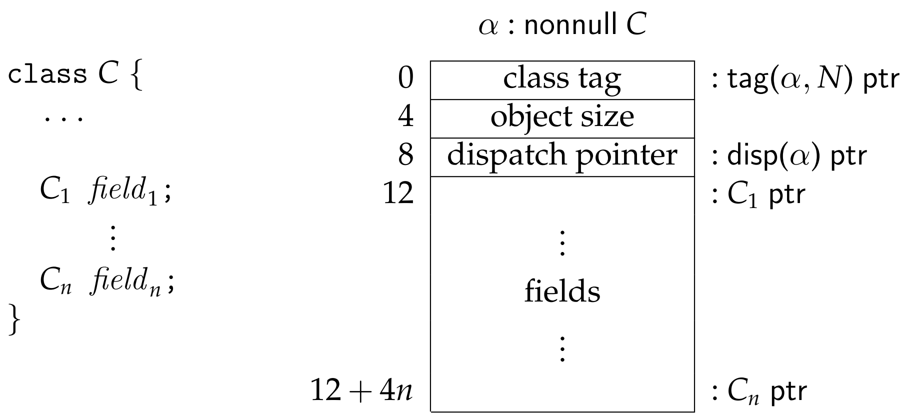


Figure 3.2: Object layout of an instance α of a class C annotated with types at appropriate offsets.

Cool Object Layout. To keep the type system used by Coolaid simple, the object layout is essentially built-in (rather than using a general record type as in traditional object encodings [AC98]). Figure 3.2 shows the layout of an object α of class C annotated with the types of the values at various offsets from the object reference. The typing rules for reading at an offset from an object reference (*e.g.*, the dispptr rule) follow directly from this diagram.

3.2 Stack and Call-Return Abstractions

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).

The verifier for compiled Cool programs described in Chapter 2 is built on top of lower-level components responsible for stack and call-return aspects. Intuitively, the stack verifier checks and identifies stack accesses and checks for stack overflow, while the call-return verifier identifies function call and returns and checks that the calling convention is obeyed (*e.g.*, callee-save registers are preserved across calls). These verifiers are arranged in a chain where the stack verifier is at the lowest-level, the call-return verifier is next, and the Coolaid verifier is at the highest-level. To interface between each pair of layers, each verifier optionally exports "higher-level assembly instructions", simplifying the job for higher-level verifiers. This process can be thought of as successively de-compiling to enable reasoning at higher levels. The result will be that the Cool-specific verifier will appear very much like the Java bytecode verifier.

Stack. From the compiler implementor's perspective, the primary purpose of the run-time stack is to preserve values across calls and provide space for spilled registers. For both of these uses, the only way the stack is modified is via indexing a constant offset from a stack pointer, and the activation record can be viewed as providing a set of additional "pseudo-registers". For basic Cool compilers, the stack handling is simplified, as all uses of the stack are one of these two forms. In particular, we do not need to handle aliasing on the stack. The stack verifier then, in essence, rewrites memory read and write operations on the stack to register operations for higher-level verifiers, which is the de-compilation it provides.

First, we describe a simple mechanism for preventing stack overflow. We assume that one segment of memory, say, of 1 MB is allocated for the stack. Each function has a some maximum size for its activation record called its *stack frame size* (either specified by a default or via compiler inserted annotations to eliminate unnecessary overflow checks). On function entry, it is assumed that the function's stack frame is valid stack space. Thus, on function call, the caller must ensure that this invariant holds for the callee. The stack frame can be extended by run-time checks using the fact that an address is a valid stack address if it is on the same segment as a known valid stack address. This check can be done with a few simple bitwise calculations if the segment size is a power of two.

We define the abstract state for the stack verifier for the above mechanism in a similar manner as for the Coolaid verifier. A value state Σ_s maps registers (and stack slots) to expressions and a type state Γ_s maps symbolic values to types. Two additional integers, n_lo and n_hi, delimit the stack frame (*i.e.*, the extent of known valid stack addresses) with respect to the value of the initial stack pointer. In other words, any address in the range $[sp_0 + n_\mathrm{lo}, sp_0 + n_\mathrm{hi}]$ is a valid stack address where sp_0 is the value of the stack pointer on entry to the function.

$$\begin{array}{ll}
\text{stack abstract state} & S ::= \langle \Sigma_\mathrm{s} \mathbin{⨾} \Gamma_\mathrm{s} \mathbin{⨾} n_\mathrm{lo} \mathbin{⨾} n_\mathrm{hi} \rangle \\
\text{stack value state} & \Sigma_\mathrm{s} ::= \cdot \mid \Sigma_\mathrm{s}, r = e \\
\text{stack type state} & \Gamma_\mathrm{s} ::= \cdot \mid \Gamma_\mathrm{s}, \alpha : \tau_\mathrm{s}
\end{array}$$

The expressions are the same as those defined in Section 2.2.1, but the values and types of interest are different. To implement the stack verifier, we need to recognize stack addresses as constant offsets from the initial stack pointer and the run-time stack overflow checks.

$$\begin{array}{rll}
\text{stack values } v_\mathrm{s} ::= & \alpha + n & \text{possible stack address} \\
\mid & (v_{\mathrm{s}0} \oplus v_{\mathrm{s}1}) \ggg n_0 \ = \ n_1 & \text{possible overflow check}
\end{array}$$

where $\oplus$ is bitwise exclusive-or and $\ggg$ is logical shift right. The value $(v_{\mathrm{s}0} \oplus v_{\mathrm{s}1}) \ggg n_0 \ = \ 0$ checks that the bits of $v_{\mathrm{s}0}$ and $v_{\mathrm{s}1}$ agree, expect possibly in the n_0 least significant bits. Analogous to the Coolaid verifier, we give a straightforward definition of a normalization judgment for expressions to stack values $\Gamma_\mathrm{s} \vdash e \Downarrow_\mathrm{s} v_\mathrm{s} \rhd \Gamma'_\mathrm{s}$.

For types, we need only have the singleton type of the value of the initial stack pointer.

$$\begin{array}{llll} \text{stack types} & \tau_{\mathrm{s}} ::= \top & \text{unknown contents} \\ & \quad\mid \mathsf{sp0} & \text{initial stack pointer} \\ & \quad\mid \bot & \text{absence of a value} \end{array}$$

Correspondingly, the subtyping relation is exceedingly simple: the reflexive-transitive closure of $\bot <:_{\mathrm{s}} \mathsf{sp0}$ and $\mathsf{sp0} <:_{\mathrm{s}} \top$. The join of two stack abstract states then follows analogously from the description for the join in the Coolaid verifier given in Section 2.2.3.

As before, the verification procedure for the stack is given by an abstract transition judgment:

$$I : \langle \Sigma_{\mathrm{s}} \,⨟\, \Gamma_{\mathrm{s}} \,⨟\, n_{\mathrm{lo}} \,⨟\, n_{\mathrm{hi}} \rangle_p \longrightarrow_{\mathrm{s}} \langle \Sigma'_{\mathrm{s}} \,⨟\, \Gamma'_{\mathrm{s}} \,⨟\, n'_{\mathrm{lo}} \,⨟\, n'_{\mathrm{hi}} \rangle_{p'}$$

However, defining this judgment directly does not indicate how to compose the stack verifier with higher-level verifiers. We obtain modular verifiers by separating this task into two parts:

1. translating the given instruction into higher-level instructions based on the current abstract state; and
2. interpreting the higher-level instruction to obtain the next abstract state.

This separation is captured by the following two auxiliary judgments:

$$\begin{array}{r} \langle \Sigma_{\mathrm{s}} \,⨟\, \Gamma_{\mathrm{s}} \,⨟\, n_{\mathrm{lo}} \,⨟\, n_{\mathrm{hi}} \rangle_p \nmid I \Rightarrow_{\mathrm{s}} I_{\mathrm{s}} \rhd \langle \Sigma'_{\mathrm{s}} \,⨟\, \Gamma'_{\mathrm{s}} \,⨟\, n'_{\mathrm{lo}} \,⨟\, n'_{\mathrm{hi}} \rangle_p \\ I_{\mathrm{s}} : \langle \Sigma_{\mathrm{s}} \,⨟\, \Gamma_{\mathrm{s}} \,⨟\, n_{\mathrm{lo}} \,⨟\, n_{\mathrm{hi}} \rangle_p \twoheadrightarrow_{\mathrm{s}} \langle \Sigma'_{\mathrm{s}} \,⨟\, \Gamma'_{\mathrm{s}} \,⨟\, n'_{\mathrm{lo}} \,⨟\, n'_{\mathrm{hi}} \rangle_{p'} \end{array}$$

Note that the auxiliary translation judgment (the top one) may give a new abstract state at the current program point allowing it to change the information captured by the instruction versus the abstract state. Contrast this to the transition judgment (the bottom one) that yields an abstract state at a succeeding program point. The abstract transition for the "top-level" verifier that only reasons about the stack is then given by the following rule:

$$\frac{\begin{array}{c} \langle \Sigma_{\mathrm{s}} \,⨟\, \Gamma_{\mathrm{s}} \,⨟\, n_{\mathrm{lo}} \,⨟\, n_{\mathrm{hi}} \rangle_p \nmid I \Rightarrow_{\mathrm{s}} I_{\mathrm{s}} \rhd \langle \Sigma'_{\mathrm{s}} \,⨟\, \Gamma'_{\mathrm{s}} \,⨟\, n'_{\mathrm{lo}} \,⨟\, n'_{\mathrm{hi}} \rangle_p \\ I_{\mathrm{s}} : \langle \Sigma'_{\mathrm{s}} \,⨟\, \Gamma'_{\mathrm{s}} \,⨟\, n'_{\mathrm{lo}} \,⨟\, n'_{\mathrm{hi}} \rangle_p \twoheadrightarrow_{\mathrm{s}} \langle \Sigma''_{\mathrm{s}} \,⨟\, \Gamma''_{\mathrm{s}} \,⨟\, n''_{\mathrm{lo}} \,⨟\, n''_{\mathrm{hi}} \rangle_{p'} \end{array}}{I : \langle \Sigma_{\mathrm{s}} \,⨟\, \Gamma_{\mathrm{s}} \,⨟\, n_{\mathrm{lo}} \,⨟\, n_{\mathrm{hi}} \rangle_p \longrightarrow_{\mathrm{s}} \langle \Sigma''_{\mathrm{s}} \,⨟\, \Gamma''_{\mathrm{s}} \,⨟\, n''_{\mathrm{lo}} \,⨟\, n''_{\mathrm{hi}} \rangle_{p'}} \ \textsf{s-step}$$

These higher-level instructions are exported to higher-level verifiers. Formally, each intermediary verifier incrementally introduces a higher-level language of instructions (*e.g.*, to extend with pseudo-registers for stack slots); we elide these details, as the changes are small and are hopefully intuitive. In this paper, the lower-level languages are sub-languages of the higher-level languages (that is, I_s includes all that instructions that I defines), though we imagine this might not be the case in general.

We give here the interesting rules for the stack verifier; all the rules are collected together in Appendix A. As mentioned above, the stack verifier's primary task is to rewrite memory accesses to register operations.

$$\frac{\Gamma_s \vdash \Sigma_s(ae) \Downarrow_s \alpha + n \triangleright \Gamma'_s \quad \Gamma'_s(\alpha) = \mathsf{sp0} \quad n_{lo} \le n \le n_{hi} \quad n \equiv 0 \pmod 4}{\langle \Sigma_s \,;\, \Gamma_s \,;\, n_{lo} \,;\, n_{hi} \rangle \vdash r := \mathtt{mem}[ae] \Rightarrow_s r := \mathbf{r}_{\mathsf{sp}_0+n} \triangleright \langle \Sigma_s \,;\, \Gamma'_s \,;\, n_{lo} \,;\, n_{hi} \rangle} \ \textsf{s-read}$$

$$\frac{\Gamma_s \vdash \Sigma_s(ae_0) \Downarrow_s \alpha + n \triangleright \Gamma'_s \quad \Gamma'_s(\alpha) = \mathsf{sp0} \quad n_{lo} \le n \le n_{hi} \quad n \equiv 0 \pmod 4}{\langle \Sigma_s \,;\, \Gamma'_s \,;\, n_{lo} \,;\, n_{hi} \rangle \vdash \mathtt{mem}[ae_0] := ae_1 \Rightarrow_s \mathbf{r}_{\mathsf{sp}_0+n} := ae_1 \triangleright \langle \Sigma_s \,;\, \Gamma'_s \,;\, n_{lo} \,;\, n_{hi} \rangle} \ \textsf{s-write}$$

The first two premises in the above rules identify the address as a stack address, while the third premise checks that the address is within the current stack frame. The last premise requires that stack accesses are word-aligned (for simplicity); this is sufficient for Coolaid. With some more details, we should be able to extend this work to handle non-word-aligned or non-word-sized accesses, but we do not consider that idea any further here. Note that both rules re-write the memory operation to use the pseudo-register for the specified stack slot $\mathbf{r}_{sp_0+n}$. For all other instruction kinds, the translation is the identity translation.

The bottom of the stack frame (n_{lo}) may be extended through a run-time stack overflow check. Recall that all valid stack addresses are on the same segment, so given a valid stack address, another address can be determined to be a stack address if it has the same higher-order bits. Recognizing this check, we extend n_{lo} on the following transition:

$$\frac{\begin{array}{l}\Gamma_s \vdash \Sigma_s(ae) \Downarrow_s ((\alpha_0 + n_0) \oplus (\alpha_0 + n_1)) \ggg 20 \ = \ 0 \triangleright \Gamma'_s \\ \Gamma'_s(\alpha_0) = \mathsf{sp0} \\ n_1 < n_{lo} \le n_0 \le n_{hi}\end{array}}{\mathtt{branch}\ ae\ L : \langle \Sigma_s \,;\, \Gamma_s \,;\, n_{lo} \,;\, n_{hi} \rangle \rightharpoonup_s \langle \Sigma_s \,;\, \Gamma'_s \,;\, n_1 \,;\, n_{hi} \rangle_L} \ \textsf{s-sp}_T$$

The above rule is for where n_0 is the currently known valid stack offset and n_1 is the offset to which we wish to extend n_{lo}. The logical shift right by 20 is for the case where the segment size is 1 MB.

Other transition rules include ones for updating a register and jumping to a label, but not for reading and writing to memory. Note that stack accesses are handled by the above translations and the register update transition. Viewing the stack verifier in isolation, no rules apply to heap read or writes, so such memory accesses would be deemed unsafe by the stack verifier alone. In other words, non-stack read and writes are left as-is for higher-level verifiers to handle appropriately.

Call-Return. The call-return verifier identifies call and return instructions and checks that the calling convention is respected. Return instructions are identified by indirect jumps to the value of the return address, and calls are identified as jumps when the return address register points to the next instruction.

Because values for implementing call-returns may be (and often are) stored in stack slots, the call-return verifier is built on top of the stack verifier. The abstract state of the call-return verifier is as follows:

$$\begin{array}{ll} \text{call-return abstract state} & F ::= \langle \Sigma_{\mathrm{f}} \mathbin{;} \Gamma_{\mathrm{f}} \mathbin{;} n_{\mathrm{pop}} \mathbin{;} S \rangle \\ \text{call-return value state} & \Sigma_{\mathrm{f}} ::= \cdot \mid \Sigma_{\mathrm{f}}, r = e \\ \text{call-return type state} & \Gamma_{\mathrm{f}} ::= \cdot \mid \Gamma_{\mathrm{f}}, \alpha : \tau_{\mathrm{f}} \end{array}$$

where n_{pop} is the amount that the callee should pop off the stack on return according to the calling convention (*e.g.*, the callee pops the function arguments on some architectures). Note that the call-return verifier contains the abstract state for the stack verifier; technically, the abstract state for Coolaid should also contain the abstract state for the call-return verifier, but is elided in Chapter 2 (see Section 3.3 for further details).

The values and types for the call-return verifier are particularly simple. Similar to the stack verifier, the types simply track singleton values.

$$\begin{array}{llll} \text{call-return values} & v_{\mathrm{f}} ::= \alpha + n & & \\ \text{call-return types} & \tau_{\mathrm{f}} ::= & \top & \text{unknown contents} \\ & & \mid \mathsf{ra} & \text{return address on entry} \\ & & \mid \mathsf{cs}(r) & \text{value of callee-save register } r \text{ on entry} \\ & & \mid \mathsf{codeaddr}(L) & \text{address of code label } L \\ & & \mid \bot & \text{absence of a value} \end{array}$$

where this lattice of types is again flat (with $\bot$ being the bottom element and $\top$ being the top element).

Like for the stack verifier, we define translation and transition judgments that define the call-return verifier:

$$\langle \Sigma_{\mathrm{f}} \mathbin{;} \Gamma_{\mathrm{f}} \mathbin{;} n_{\mathrm{pop}} \mathbin{;} S \rangle_p \vdash I \Rightarrow_{\mathrm{f}} I_{\mathrm{f}} \rhd \langle \Sigma'_{\mathrm{f}} \mathbin{;} \Gamma'_{\mathrm{f}} \mathbin{;} n'_{\mathrm{pop}} \mathbin{;} S' \rangle_p$$
$$I_{\mathrm{f}} : \langle \Sigma_{\mathrm{f}} \mathbin{;} \Gamma_{\mathrm{f}} \mathbin{;} n_{\mathrm{pop}} \mathbin{;} S \rangle_p \rightarrow_{\mathrm{f}} \langle \Sigma'_{\mathrm{f}} \mathbin{;} \Gamma'_{\mathrm{f}} \mathbin{;} n'_{\mathrm{pop}} \mathbin{;} S' \rangle_{p'}$$
$$I_{\mathrm{f}} : \langle \Sigma_{\mathrm{f}} \mathbin{;} \Gamma_{\mathrm{f}} \mathbin{;} n_{\mathrm{pop}} \mathbin{;} S \rangle_p \ \mathsf{ok}$$

The last judgment indicates that the instruction in the given state is valid end point (*i.e.*, a return). Thus far, the checking of return points has only been alluded to informally.

To emphasize that these verifiers build on each other, we define the translation using an auxiliary judgment that translates from stack instructions.

$$\langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p \mid I_{\mathrm{s}} \mathrel{\dot{\Rightarrow}_{\mathrm{f}}} I_{\mathrm{f}} \rhd \langle \Sigma'_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n'_{\mathrm{pop}} \fatsemi S' \rangle_p$$

A general rule combines the translations of the sub-verifiers.

$$\frac{S \mid I \Rightarrow_{\mathrm{s}} I_{\mathrm{s}} \rhd S' \qquad \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S' \rangle \mid I_{\mathrm{s}} \mathrel{\dot{\Rightarrow}_{\mathrm{f}}} I_{\mathrm{f}} \rhd \langle \Sigma'_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n'_{\mathrm{pop}} \fatsemi S'' \rangle}{\langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle \mid I \Rightarrow_{\mathrm{f}} I_{\mathrm{f}} \rhd \langle \Sigma'_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n'_{\mathrm{pop}} \fatsemi S'' \rangle} \ \textsf{f-decomp}$$

We give here the interesting rules for the call-return verifier; all the rules are collected together in Appendix A. Recall that the call-return verifier identifies function call and function return instructions. Indirect jumps to the return address translate into return instructions.

$$\frac{\Gamma_{\mathrm{f}} \vdash \Sigma_{\mathrm{f}}(ae) \Downarrow_{\mathrm{f}} \alpha \rhd \Gamma'_{\mathrm{f}} \qquad \Gamma'_{\mathrm{f}}(\alpha) = \mathsf{ra}}{\langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle \mid \ \mathtt{jump}\ [ae] \ \mathrel{\dot{\Rightarrow}_{\mathrm{f}}} \ \mathtt{return}\ \mathbf{r}_{rv} \ \rhd \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle} \ \textsf{f-return}$$

So that higher-level verifiers need not be concerned about the calling convention, the return instruction gives the return value (in terms of an assembly expression).

Jumps are translated to calls if the return address register points to the next instruction.

$$\frac{\begin{array}{l} \Gamma_{\mathrm{f}} \vdash \Sigma_{\mathrm{f}}(\mathbf{r}_{ra}) : \mathsf{codeaddr}(L') \rhd \Gamma'_{\mathrm{f}} \qquad (\&L' = p+1) \\ e_i = \begin{cases} \mathbf{r}_{sp} + npop(L) & \text{if } r_i \text{ is } \mathbf{r}_{sp} \\ r_i & \text{if } r_i \text{ is preserved by the callee} \\ ? & \text{otherwise} \end{cases} \end{array}}{\begin{array}{l} \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p \mid \ \mathtt{jump}\ L \\ \qquad \mathrel{\dot{\Rightarrow}_{\mathrm{f}}} \ \mathtt{call}\ L(\mathbf{r}_{arg_0}, \ldots, \mathbf{r}_{arg_\ell}) : \mathbf{r}_{rv}, \ \overrightarrow{r := e} \ \rhd \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p \end{array}} \ \textsf{f-call}$$

Similar to the return instruction, the call instruction includes the arguments and the return value. It also yields a list of register update instructions $r_0 := e_0$, $r_1 := e_1$, . . . , $r_{k-1} := e_{k-1}$ (abbreviated $\overrightarrow{r := e}$) that conservatively models the "effect" of the function call. The stack pointer register is incremented by the amount that the callee must pop on return ($npop(L)$), the callee-save registers keep their same values, and all other registers are havocked (by assigning the expression ? indicating an unknown value).

Indirect jumps that can be recognized as calls are also translated into indirect call instructions (though since the target of the call is unknown, the arguments cannot be given).

$$\frac{\begin{array}{l}\Gamma_{\mathrm{f}} \vdash \Sigma_{\mathrm{f}}(\mathbf{r}_{ra}) : \mathsf{codeaddr}(L') \rhd \Gamma'_{\mathrm{f}} \qquad (\&L' = p+1) \\ e_i = \begin{cases} r_i & \text{if } r_i \text{ is preserved by the callee} \\ ? & \text{otherwise} \end{cases}\end{array}}{\langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p \vdash \texttt{jump}\ [ae] \ \dot{\Rightarrow}_{\mathrm{f}}\ \texttt{call}\ [ae] : \mathbf{r}_{rv},\ \overrightarrow{r := e}\ \rhd \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p} \ \text{f-icall}$$

For all other instruction kinds, the translation is the identity translation.

A return instruction is deemed safe if the call-return verifier can determine that the callee-save registers have been restored and the stack pointer is pointing to the location dictated by the calling convention.

$$\frac{\begin{array}{l}\Gamma_{\mathrm{f}} \vdash \Sigma_{\mathrm{f}}(r_{cs}) \Downarrow_{\mathrm{f}} \alpha \rhd \Gamma'_{\mathrm{f}} \qquad \Gamma'_{\mathrm{f}}(\alpha) = \mathsf{cs}(r_{cs}) \qquad (\text{for all callee-save } r_{cs}) \\ S.\Gamma_{\mathrm{s}} \vdash S.\Sigma_{\mathrm{s}}(\mathbf{r}_{sp}) \Downarrow_{\mathrm{s}} \alpha + n_{\mathrm{pop}} \rhd \Gamma'_{\mathrm{s}} \qquad \Gamma'_{\mathrm{s}}(\alpha) = \mathsf{sp0}\end{array}}{\texttt{return}\ ae : \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle\ \mathsf{ok}} \ \text{f-returnok}$$

where we write $S.\Gamma_{\mathrm{s}}$ for the type-state projection of S (and so forth).

A function call is okay if the there is enough stack space for the callee and yields a transition to an abstract state where the effects of the call have been reflected.

$$\frac{\begin{array}{l}S.\Gamma_{\mathrm{s}} \vdash S.\Sigma_{\mathrm{s}}(\mathbf{r}_{sp}) \Downarrow_{\mathrm{s}} \alpha + n \rhd \Gamma^0_{\mathrm{s}} \quad \Gamma^0_{\mathrm{s}}(\alpha) = \mathsf{sp0} \quad n \equiv 0 \ (\mathrm{mod}\ 4) \\ S^0 = \langle S.\Sigma_{\mathrm{s}} \fatsemi \Gamma^0_{\mathrm{s}} \fatsemi S.n_{\mathrm{lo}} \fatsemi S.n_{\mathrm{hi}} \rangle \\ n + 4 \cdot \mathit{nargs}(L) \le S^0.n_{\mathrm{hi}} \quad n + 4 \cdot \mathit{nargs}(L) - \mathit{framesize}(L) \ge S^0.n_{\mathrm{lo}} \\ r_i := e_i : \langle \Sigma^i_{\mathrm{f}} \fatsemi \Gamma^i_{\mathrm{f}} \fatsemi n^i_{\mathrm{pop}} \fatsemi S^i \rangle \rightharpoonup_{\mathrm{f}} \langle \Sigma^{i+1}_{\mathrm{f}} \fatsemi \Gamma^{i+1}_{\mathrm{f}} \fatsemi n^{i+1}_{\mathrm{pop}} \fatsemi S^{i+1} \rangle \quad (\text{for } 0 \le i < k)\end{array}}{\texttt{call}\ L(r_{arg_0}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e} : \langle \Sigma^0_{\mathrm{f}} \fatsemi \Gamma^0_{\mathrm{f}} \fatsemi n^0_{\mathrm{pop}} \fatsemi S \rangle \rightharpoonup_{\mathrm{f}} \langle \Sigma^k_{\mathrm{f}} \fatsemi \Gamma^k_{\mathrm{f}} \fatsemi n^k_{\mathrm{pop}} \fatsemi S^k \rangle} \ \text{f-callok}$$

where $\mathit{nargs}(L)$ and $\mathit{framesize}(L)$ give the number of arguments and the initial stack size of function L, respectively. Note that the factor of 4 in the above is the word size for a 32-bit machine, as an example.

Similar to the stack verifier, other transition rules include ones for register updates and jumps to a label, but for not indirect calls. The call-return verifier identifies indirect calls, but the safety of such a call is left to higher-level verifiers.

One might wonder why the value state is even necessary for the call-return verifier. For some compilations, the value state may indeed not be necessary if the values it needs to reason about are simply moved around registers and stack slots. However, some simple compiler optimizations break this assumption; for example, a callee-save register must be updated by adding a constant offset within the function body but is preserved by the compilation by subtracting the constant before returning (*e.g.*, the stack or frame pointer). Using lazy typing simplifies the handling of such possibilities.

3.3 Abstract Transition and Typing

The complete Coolaid verifier is then built by placing the Cool-specific verifier outlined in Chapter 2 on top of the call-return verifier. Formally, the abstract state for the Cool-specific verifier includes the abstract state of the call-return verifier. Following the framework described in the previous section, we also separate the translation and transition parts.

$$\langle \Sigma \,⨟\, \Gamma \,⨟\, F \rangle_p \vdash I \Rightarrow I_c \rhd \langle \Sigma' \,⨟\, \Gamma' \,⨟\, F' \rangle_p$$
$$I_c : \langle \Sigma \,⨟\, \Gamma \,⨟\, F \rangle_p \rightarrowtriangle \langle \Sigma' \,⨟\, \Gamma' \,⨟\, F' \rangle_{p'}$$

As before, we define the translation using an auxiliary judgment that translates from the previous level

$$\langle \Sigma \,⨟\, \Gamma \,⨟\, F \rangle_p \vdash I_f \dot{\Rightarrow} I_c \rhd \langle \Sigma' \,⨟\, \Gamma' \,⨟\, F' \rangle_p$$

We then get an abstract transition relation that is very similar to that of the bytecode verifier. For example, consider dynamic dispatch.

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma(ae) : \mathsf{meth}(\alpha, n) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \alpha \qquad \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma'' \\ T(C) = \texttt{class}\ C\ \ldots\ \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \\ (m \text{ is the method at offset } n \text{ of class } C)\end{array}}{\begin{array}{l}\langle \Sigma \,⨟\, \Gamma \,⨟\, F \rangle \vdash \texttt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\ \qquad \dot{\Rightarrow}\ \texttt{invokevirtual}\ C.m(\mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e}\ \rhd \langle \Sigma \,⨟\, \Gamma'' \,⨟\, F \rangle\end{array}}\ \text{c-meth}$$

$$\frac{\begin{array}{l}T(C) = \texttt{class}\ C\ \ldots\ \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \\ \Gamma_0 \vdash \Sigma_0(r_{arg_1}) : \tau_1 \rhd \Gamma_1 \quad \cdots \quad \Gamma_{\ell-1} \vdash \Sigma_0(r_{arg_\ell}) : \tau_\ell \rhd \Gamma_\ell \\ r_i := e_i : \langle \Sigma_i \,⨟\, \Gamma_{\ell+i} \,⨟\, F_i \rangle \rightarrowtriangle \langle \Sigma_{i+1} \,⨟\, \Gamma_{\ell+i+1} \,⨟\, F_{i+1} \rangle \quad (\text{for } 0 \le i < k) \\ \texttt{call}\ m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e}\ : F_0 \rightarrowtriangle_f F' \\ (\beta \text{ fresh})\end{array}}{\begin{array}{l}\texttt{invokevirtual}\ C.m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e} \\ \qquad : \langle \Sigma_0 \,⨟\, \Gamma_0 \,⨟\, F_0 \rangle \rightarrowtriangle \langle \Sigma_k[r_{rv} \mapsto \beta] \,⨟\, \Gamma_{\ell+k}[\beta \mapsto \tau_{rv}] \,⨟\, F' \rangle\end{array}}\ \text{c-methok}$$

The translation rule captures recognizing the indirect call as a dynamic dispatch, while the transition rule is now very similar to the dynamic dispatch rule for the bytecode verifier given in Section 2.1. The dynamic dispatch rule given in Section 2.3 is essentially these two rules put together.

In the remainder of this section, we describe the verification of language features not covered by the example in Section 2.3.

```
Seq.next: smeth(Seq, 12)
      ...




Main.main:
  ... args and ra ...
  jump Seq.next
```

(a)

```
   Seq_disp: sdisp(Seq)
 0  .word  ...
 4  .word  ...
 8  .word  ...
12  .word  Seq.next

Main.main:
  r_t := mem[(add &Seq_disp 12)]
                  ⟨r_t = α ; α : smeth(Seq, 12)⟩
  ... args and ra ...
  jump [r_t]
```

(b)

```
   Seq_prot: nonnull exactly Seq
 0  .word  ...
 4  .word  ...
 8  .word  Seq_disp

Main.main:
  r_t := mem[(add &Seq_prot 8)]
                  ⟨r_t = β ; β : sdisp(Seq)⟩
  r_t := mem[(add r_t 12)]
                  ⟨r_t = α ; α : smeth(Seq, 12)⟩
  ... args and ra ...
  jump [r_t]
```

(c)

Figure 3.3: Three correct compilations of a static dispatch. Typing annotations for labels shown underlined, and abstract states shown boxed and right-justified.

3.3.1 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 analogous to non-virtual method calls in C++, but in contrast to static method calls in Java (an example is shown in Figure 3.3). The compiler can, therefore, simply emit a direct jump to the code label for the method (a). However, in many of the Cool compilers with which we experimented, we observed that static dispatch was implemented with indirect jumps based on indexing into the dispatch table for the particular class (b) or even first obtaining the dispatch table by indexing through a statically allocated, constant "prototype object" (c) (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 and treating the abstract transition rules that apply to indirect calls as also applying to direct calls, viewing `call` L as `call` $[\&L]$.

In Figure 3.3, the following labels have been annotated with the following types:

$$\texttt{Seq.next} : \mathsf{smeth}(\texttt{Seq}, 12) \qquad \texttt{Seq\_disp} : \mathsf{sdisp}(\texttt{Seq}) \qquad \texttt{Seq\_prot} : \mathsf{nonnull\ exactly\ Seq}$$

for the method, the dispatch table, and the "prototype object", respectively, so that in each case the call is to a value of type $\mathsf{smeth}(\texttt{Seq}, 12)$ (*i.e.*, the `next` method of `Seq`). The following rules to look up methods and dispatch tables for static dispatch, which are similar to the ones for dynamic dispatch, permit these deductions.

$$\frac{\Gamma \vdash e \Downarrow \beta + n \rhd \Gamma' \quad \Gamma' \vdash \beta : \mathsf{sdisp}(C) \rhd \Gamma'' \quad (C \text{ has a method at offset } n)}{\Gamma \vdash e : \mathsf{smeth}(C, n)\ \mathsf{ptr} \rhd \Gamma''}\ \mathsf{smethptr}$$

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

Note that statically-allocated objects can be typed at initialization time with the exactly qualifier by examining their class tags.

3.3.2 Object Allocation and Initialization

Cool programs allocate objects by calling a trusted run-time function that clones a given object. This idiom avoids some of the ugliness of compiling (and verifying) memory allocation, particularly partially initialized objects that are not well-typed. The object to clone can be a statically allocated "prototype" object, which is well-typed, but may not be initialized as in the source program. An initialization method is then called to initialize fields as necessary. Note that this protocol for object allocation is not imposed by Coolaid, but rather the existing run-time system.

Given this protocol for object allocation, most allocations can be handled following the previous discussion on method/function calls in a straightforward manner—by allowing calls to the run-time allocation function `Obj.copy` and the initialization methods with appropriate argument and return types. Self-type polymorphism in Cool, however, adds some bit of additional complexity (that is not present in Java).

In Cool, code can allocate an object of self-type (*i.e.*, having the same dynamic type as the self object) with the statement `new SELF_TYPE`. One compilation strategy for this statement, supported by Coolaid, is to keep an "initialization" table of prototype objects and initialization methods for each class indexed by the class tag and then generate code that looks into this table for a prototype object to clone and initialization method based on the class tag for the self object (as sketched in Figure 3.4). A first attempt might assign the following types for the code fragment in Figure 3.4 (shown boxed and right-justified). Observe that γ and δ are assigned type nonnull C after line 2 for the prototype object and prototype clone corresponding to α, respectively. Then, line 6 appears okay, as it is determined to be a call to

$$\boxed{\langle \mathbf{r}_{self} = \alpha \fatsemi \alpha : \mathsf{nonnull}\ C \rangle}$$

1 $\mathbf{r}_t := \texttt{mem}[\mathbf{r}_{self}]$

$$\boxed{\langle \mathbf{r}_t = \beta, \mathbf{r}_{self} = \alpha \fatsemi \beta : \mathsf{tag}(\alpha, \{\ldots\}), \alpha : \mathsf{nonnull}\ C \rangle}$$

2 $\mathbf{r}_{arg_0} := \texttt{mem}[\texttt{\&init\_table} + 4 \cdot \mathbf{r}_t]$

$$\boxed{\langle \mathbf{r}_{arg_0} = \gamma, \mathbf{r}_t = \beta \fatsemi \gamma : \mathsf{nonnull}\ C, \beta : \mathsf{tag}(\alpha, \{\ldots\}) \rangle}$$

3 $\texttt{jump Obj.copy}$

4 $\mathbf{r}_{arg_0} := \mathbf{r}_{rv}$

$$\boxed{\langle \mathbf{r}_{arg_0} = \delta, \mathbf{r}_t = \beta \fatsemi \delta : \mathsf{nonnull}\ C, \beta : \mathsf{tag}(\alpha, \{\ldots\}) \rangle}$$

5 $\mathbf{r}_{init} := \texttt{mem}[\texttt{\&init\_table} + 4 \cdot \mathbf{r}_t + 4]$

$$\boxed{\langle \mathbf{r}_{init} = \varepsilon, \mathbf{r}_{arg_0} = \delta \fatsemi \varepsilon : \mathsf{sinit}(C), \delta : \mathsf{nonnull}\ C \rangle}$$

6 $\texttt{jump}\ [\mathbf{r}_{init}]$

Figure 3.4: Naïve (unsound) typing of a valid compilation for allocating and initializing an object of self-type.

the initialization method for class C with an argument of type $\mathsf{nonnull}\ C$. This is, however, unsound. To see why, consider the case where $\mathbf{r}_{self}$ is dynamically an object of class B and B is a subclass of C, and suppose line 4 is changed to

4 $\mathbf{r}_{arg_0} := \mathbf{r}_x$

where the value in $\mathbf{r}_x$ is of type $\mathsf{nonnull}\ C$ but is also dynamically an object of class C. Following the above reasoning, this modified program would also type check, but this verification is unsound. The unsoundness is similar to that of the naïve typing of dynamic dispatch described in Section 2.1: the initialization method for class B is called on an object of class C, which lacks fields declared in B.

To resolve this unsoundness, we clearly must track the dependency that ε is the initialization method for object α (assigning the type $\mathsf{init}(\alpha)$ instead). However, it does not seem reasonable to check that the argument to an initialization method is the same as the object from which the method was obtained as we did for dynamic dispatch—the valid compilation given in Figure 3.4 would not pass. Rather, we require the weaker pre-condition that the argument of an initialization method of α (*i.e.*, $\mathsf{init}(\alpha)$) must have the same dynamic type as α (*i.e.*, be of type $\mathsf{nonnull}\ \mathsf{classof}(\alpha)$), introducing the need for the $\mathsf{classof}(\alpha)$ reference type. This check is reflected in the translation rule for initialization method calls:

$$\dfrac{\begin{array}{ll} \Gamma \vdash \Sigma(ae) : \mathsf{init}(\alpha) \rhd \Gamma' & \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma'' \\ \Sigma(\mathbf{r}_{arg_0}) = \beta & \Gamma'' \vdash \beta : \mathsf{nonnull}\ \mathsf{classof}(\alpha) \rhd \Gamma''' \end{array}}{\begin{array}{l} \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \vdash \texttt{call}\ [ae] : r_{rv}, \overrightarrow{r := e} \\ \qquad\qquad \overset{\cdot}{\Rightarrow}\ \texttt{invokeinit}\ C : \mathbf{r}_{arg_0}, \overrightarrow{r := e}\ \rhd \langle \Sigma \fatsemi \Gamma''' \fatsemi F \rangle \end{array}}\ \textsf{c-init}$$

Additionally, for the example in Figure 3.4 to type-check, we need to assume the stronger post-condition ensured by `Obj.copy` that it returns an object of the same dynamic type as its argument.

3.3.3 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 4.1). 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.

For example, consider a program with three classes A, B, and C such that A is a subclass of B and B is a subclass of C. Further suppose they have been assigned the following class tags:

A	B	C
3	2	1

Supposing the value in $\mathbf{r}_x$ has type $\mathsf{nonnull}\ C$, then the following fragment implements a checked down cast from $\mathsf{nonnull}\ C$ to $\mathsf{nonnull}\ A$ and is verified with the abstract states shown below.

$$\boxed{\langle \mathbf{r}_x = \alpha \mathbin{⨾} \alpha : \mathsf{nonnull}\ C \rangle}$$

1 $\mathbf{r}_t := \mathtt{mem}[\mathbf{r}_x]$

$$\boxed{\langle \mathbf{r}_t = \beta, \mathbf{r}_x = \alpha \mathbin{⨾} \beta : \mathsf{tag}(\alpha, \{3,2,1\}), \alpha : \mathsf{nonnull}\ C \rangle}$$

2 $\mathtt{branch}\ (< \mathbf{r}_t\ 2)\ L_{\mathtt{notB}}$

$$\boxed{\langle \mathbf{r}_t = \beta, \mathbf{r}_x = \alpha \mathbin{⨾} \beta : \mathsf{tag}(\alpha, \{3,2\}), \alpha : \mathsf{nonnull}\ B \rangle}$$

First, recall from 3.1, the Cool object layout has the class tag stored in the first word of the object, so the read on line 1 fetches the class tag for the object in $\mathbf{r}_x$. That value (β) is assigned the type of the class tag of object α with the possible set of class tags $\{3, 2, 1\}$ according to the following typing rule:

$$\frac{\Gamma \vdash e \Downarrow \alpha \rhd \Gamma' \qquad \Gamma'(\alpha) = \mathsf{nonnull}\ C}{\Gamma \vdash e : \mathsf{tag}(\alpha, \{n \mid n = \mathit{tagof}(C') \wedge \Gamma' \vdash C' <: C\})\ \mathsf{ptr} \rhd \Gamma'}\ \mathsf{tagptr}$$

where the function $\mathit{tagof}(C)$ gives the tag for class C. After the branch instruction on line 2, the branch condition is reflected by filtering the possible tags and assigning the type corresponding to least upper bound of the remaining tags, according to following abstract transition rule:

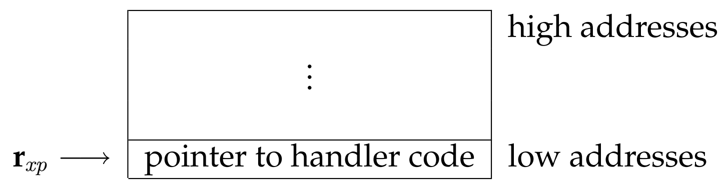
$$\frac{\begin{array}{l}\Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ k \rhd \Gamma' \qquad \Gamma'(\alpha) = \mathsf{tag}(\beta, N) \qquad \Gamma'(\beta) = \mathsf{nonnull}\ C \\ N' = \{n \in N \mid \neg(n\ R\ k)\} \neq \emptyset \\ \texttt{branch}\ ae\ L : F \mathrel{-\!\!\triangleright}_{\mathsf{f}} F'\end{array}}{\begin{array}{l}\texttt{branch}\ ae\ L \\ \quad : \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle_p \mathrel{-\!\!\triangleright} \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \mathsf{tag}(\beta, N')][\beta \mapsto \mathsf{nonnull}\ \mathit{taglub}(N')] \fatsemi F' \rangle_{p+1}\end{array}}\ \textsf{c-refinetag}_F$$

where the auxiliary function $\mathit{taglub}(N)$ yields the class that is the least upper bound in the class hierarchy given a set of class tags N.

3.3.4 Exceptions

As noted earlier, we have extended the Cool source language with exceptions (using Java-like `throw`, `try-catch`, and `try-finally` constructs). Unlike Java, any object can be thrown as the value of the exception. Furthermore, all `catch` blocks handle all exceptions (*i.e.*, there is no filtering based on type); this effect can be obtained by using a type-case in the handler and re-throwing the exception for unhandled types.

Coolaid is able to verify a compilation of these constructs following the so-called "long jump" scheme. At a high-level, this compilation strategy builds a closure of the handler code (called an *exception frame*), links it to the previous exception frame, and pushes it on top of the run-time stack at each `try` block. Then, the handler code first restores the state from the exception frame. Exceptions frames form a linked-list, and a register $\mathbf{r}_{xp}$, which must be callee-save, is used to point to the head of the list (*i.e.*, the most enclosing handler). For Coolaid, the only requirement on the layout of the exception frame is that the first word is the address for the code of the handler, as shown below:



We might, for example, save the handler code pointer, the self pointer, the frame pointer, the stack pointer, and the next exception frame pointer in an exception frame in the following manner:

$\mathbf{r}_{sp} := (\texttt{sub}\ \mathbf{r}_{sp}\ {-20})$
$\texttt{mem}[(\texttt{add}\ \mathbf{r}_{sp}\ 20)] := \mathbf{r}_{xp}$
$\texttt{mem}[(\texttt{add}\ \mathbf{r}_{sp}\ 16)] := \mathbf{r}_{sp}$
$\texttt{mem}[(\texttt{add}\ \mathbf{r}_{sp}\ 12)] := \mathbf{r}_{fp}$
$\texttt{mem}[(\texttt{add}\ \mathbf{r}_{sp}\ 8)] := \mathbf{r}_{self}$
$\texttt{mem}[(\texttt{add}\ \mathbf{r}_{sp}\ 4)] := \&\mathrm{L}_{\texttt{handler}}$
$\mathbf{r}_{xp} := (\texttt{add}\ \mathbf{r}_{sp}\ 4)$

Note that the compilation of exceptions makes several stack writes, so we will rely on the stack verifier to simplify much of this handling.

There is no explicit introduction rule for the excframe type, but rather it assumed as the type of the value in $\mathbf{r}_{xp}$ (the register that points the current exception frame) on method entry. In other words, excframe is more precisely the singleton type of the exception frame in a caller's activation record pointed to by $\mathbf{r}_{xp}$ on method entry.

Exceptions are verified using an assume-guarantee style reasoning analogous to function call-return. Handler code is verified when a method call is encountered that could jump to that code if an exception were thrown from within the method. To verify the handler code, we assume only that the exception pointer (in $\mathbf{r}_{xp}$) is preserved and the exception value (in $\mathbf{r}_{rv}$) is an object; these conditions must be then guaranteed on an "exceptional return".

Call. Thus far, the abstract transition rules for method dispatch have ignored exceptions. As usual, we have to extend the verification of a method dispatch to check both the normal return and an exceptional return. Here, we give the modified rules for dynamic dispatch with the modification boxed; the rules for other calls are similar (see Appendix A).

We first modify the dynamic dispatch instruction (invokevirtual) to (optionally) yield both the register in which the exception value is passed and the effects of an exceptional return. The form of invokevirtual is now as follows:

$$\texttt{invokevirtual}\ C.m(\mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e}\ [\!]\ \mathit{exchandler}$$

where $\mathit{exchandler} ::= r_{xv},\ \overrightarrow{r := e}\ @L \mid \mathsf{incaller}$ (that is, an indication that the dynamically nearest enclosing handler is at label L with the exception value passed in r_{xv} and effects of the call modeled by $\overrightarrow{r := e}$, or the handler is in a caller). In the case that the most enclosing handler is in a caller (*i.e.*, the current exception frame is in a caller's activation record), then an exceptional return does not return into this method, so only the normal return needs to be checked.

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma(ae) : \mathsf{meth}(\alpha, n) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \alpha \qquad \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma'' \\ T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \\ (m \text{ is the method at offset } n \text{ of class C}) \\ \boxed{\Gamma'' \vdash \Sigma(\mathbf{r}_{xp}) : \mathsf{excframe} \rhd \Gamma'''}\end{array}}{\begin{array}{l}\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \vdash \texttt{call}\ [ae] : r_{rv}, \overrightarrow{r := e} \\ \quad \dot{\Rightarrow}\ \texttt{invokevirtual}\ C.m(\mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_\ell}) : r_{rv}, \overrightarrow{r := e}\ []\ \mathsf{incaller}\ \rhd \langle \Sigma \fatsemi \Gamma''' \fatsemi F \rangle\end{array}}\ \textsf{c-meth}$$

In the case that the most enclosing handler is in the current method, then a method call is treated as branching instruction: one branch for the normal return and one branch for the exceptional return.

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma(ae) : \mathsf{meth}(\alpha, n) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \alpha \qquad \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma'' \\ T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \\ (m \text{ is the method at offset } n \text{ of class C}) \\ \boxed{\begin{array}{l}F.S.\Gamma_{\mathrm{s}} \vdash F.S.\Sigma_{\mathrm{s}}(\mathbf{r}_{xp}) \Downarrow_{\mathrm{s}} \alpha + n_{xp} \rhd \Gamma'_{\mathrm{s}} \qquad \Gamma'_{\mathrm{s}}(\alpha) = \mathsf{sp0} \\ F.\Gamma_{\mathrm{f}} \vdash F.\Sigma_{\mathrm{f}}(\mathbf{r}_{sp_0+n_{xp}}) :_{\mathrm{f}} \mathsf{codeaddr}(L) \rhd \Gamma'_{\mathrm{f}} \\ F' = \langle F.\Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi F.n_{\mathrm{pop}} \fatsemi \langle F.S.\Sigma_{\mathrm{s}} \fatsemi \Gamma'_{\mathrm{s}} \fatsemi F.S.n_{\mathrm{lo}} \fatsemi F.S.n_{\mathrm{hi}} \rangle \rangle \\ e'_i = \begin{cases} r'_i & \text{if } r'_i \text{ is preserved by the callee} \\ ? & \text{otherwise} \end{cases}\end{array}}\end{array}}{\begin{array}{l}\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \vdash \texttt{call}\ [ae] : r_{rv}, \overrightarrow{r := e} \\ \qquad \dot{\Rightarrow}\ \texttt{invokevirtual}\ C.m(\mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_\ell}) : \\ \qquad\qquad r_{rv}, \overrightarrow{r\ := e}\ []\ \mathbf{r}_{rv}, \overrightarrow{r' := e'}\ @L\ \rhd \langle \Sigma \fatsemi \Gamma'' \fatsemi F' \rangle\end{array}}\ \textsf{c-methx}$$

where only $\mathbf{r}_{xp}$ and stack slots not accessible by the callee are considered preserved by the callee.

The abstract transition rule for the normal return is essentially unchanged from before. We give it here for completeness.

$$\frac{\begin{array}{l}T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \\ \Gamma_0 \vdash \Sigma_0(r_{arg_1}) : \tau_1 \rhd \Gamma_1 \ \cdots\ \Gamma_{\ell-1} \vdash \Sigma_0(r_{arg_\ell}) : \tau_\ell \rhd \Gamma_\ell \\ r_i := e_i : \langle \Sigma_i \fatsemi \Gamma_{\ell+i} \fatsemi F_i \rangle \rightarrowtriangle \langle \Sigma_{i+1} \fatsemi \Gamma_{\ell+i+1} \fatsemi F_{i+1} \rangle \quad (\text{for } 0 \leq i < k) \\ \texttt{call}\ m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv}, \overrightarrow{r := e} : F_0 \rightarrowtriangle_{\mathrm{f}} F' \\ (\beta \text{ fresh})\end{array}}{\begin{array}{l}\texttt{invokevirtual}\ C.m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv}, \overrightarrow{r := e}\ []\ \mathit{exchandler} \\ \qquad : \langle \Sigma_0 \fatsemi \Gamma_0 \fatsemi F_0 \rangle \rightarrowtriangle \langle \Sigma_k[r_{rv} \mapsto \beta] \fatsemi \Gamma_{\ell+k}[\beta \mapsto \tau_{rv}] \fatsemi F' \rangle\end{array}}\ \textsf{c-methok}$$

For the exceptional return, note that the rule only applies when the translation determines that the handler is in the method being analyzed.

$$\frac{\begin{array}{l} T(C) = \texttt{class}\ C\ \ldots\ \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \\ \Gamma_0 \vdash \Sigma_0(r_{arg_1}) : \tau_1 \triangleright \Gamma_1 \quad \cdots \quad \Gamma_{\ell-1} \vdash \Sigma_0(r_{arg_\ell}) : \tau_\ell \triangleright \Gamma_\ell \\ \ r'_i := e'_i : \langle \Sigma_i \fatsemi \Gamma_{\ell+i} \fatsemi F_i \rangle \mathrel{-\!\triangleright} \langle \Sigma_{i+1} \fatsemi \Gamma_{\ell+i+1} \fatsemi F_{i+1} \rangle \quad (\text{for } 0 \leq i < k) \\ (\beta \text{ fresh}) \end{array}}{\begin{array}{l} \texttt{invokevirtual}\ C.m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e}\ []\ r_{xv},\ \overrightarrow{r' := e'}\ @L \\ \qquad\qquad : \langle \Sigma_0 \fatsemi \Gamma \fatsemi F_0 \rangle \mathrel{-\!\triangleright} \langle \Sigma_k[r_{xv} \mapsto \beta] \fatsemi \Gamma_{\ell+k}[\beta \mapsto \texttt{Object}] \fatsemi F_k \rangle_L \end{array}} \ \textsf{c-methxok}$$

Note that the resulting abstract state is at program point L (the exception handler) with the "effects" of a call and exceptional return reflected. Following the typing of exceptions in Cool, the only assumption that can made about the exception value is that it is an object.

Throw. A exception throw is recognized as an indirect jump to the handler of an exception (a value of type handler). A value of type handler is obtained by reading from the first word in an exception frame.

$$\frac{\Gamma \vdash \Sigma(ae) : \textsf{handler} \triangleright \Gamma'}{\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid \texttt{jump}\ [ae] \overset{\cdot}{\Rightarrow} \texttt{throw}\ \mathbf{r}_{rv} \triangleright \langle \Sigma \fatsemi \Gamma' \fatsemi F \rangle} \ \textsf{c-throw}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha \triangleright \Gamma' \quad \Gamma' \vdash \alpha : \textsf{excframe} \triangleright \Gamma''}{\Gamma \vdash e : \textsf{handler ptr} \triangleright \Gamma''} \ \textsf{throwptr}$$

Note that handler is then also a singleton type (for the value of the handler code address of the exception frame pointed to by $\mathbf{r}_{xp}$ on method entry).

A `throw` is an exceptional return, which is handled in a similar way as the normal return. As noted above, we must guarantee that the exception pointer ($\mathbf{r}_{xp}$) is preserved and the exception value is an object in this case.

$$\frac{\Gamma \vdash \Sigma(\mathbf{r}_{xp}) : \textsf{excframe} \triangleright \Gamma' \quad \Gamma' \vdash \Sigma(ae) : \texttt{Object} \triangleright \Gamma''}{\texttt{throw}\ ae : \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle\ \textsf{ok}} \ \textsf{c-throwok}$$

One may observe that the mechanism for verifying exceptions handles the case where the handler for the exception is in the caller's code. The case that the handler is in the same method as the exception throw appears as simply a `jump` within the method, which can be verified by the existing mechanisms.

3.4 Initialization

Because the abstract domain is defined with respect to a particular class hierarchy, Coolaid needs access to that information for the program being verified. In all, we need the parent class of each user-defined class, the types of the fields of each class, and the argument and return 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. To make Coolaid as user-friendly as possible, great lengths were taken to minimize the amount of annotations. In reference to our experiments discussed in Chapter 4, note that we did not need to change any of the student compilers to obtain the required annotations. These annotations can be trivially reconstructed from the source code and inserted into the assembly code independent of the compiler; however, we do have to impose the requirement that methods and fields are placed in the same order as in the source file.

At start-up, Coolaid must create initial abstract states for each method and initialization function in the code. The initial abstract states for the stack and call-return are initialized according to the calling convention specified by the Cool run-time and the abstract state for the types verifier is initialized using the method type annotations to type the arguments. In addition, we populate the initial type state Γ with types for the code label of each method (*i.e.*, with $\mathsf{smeth}(C, n)$), the code label for each initialization method (*i.e.*, with $\mathsf{sinit}(C)$), and the data label for each statically allocated object (*i.e.*, with $\mathsf{nonnull\ exactly}\ C$). Various run-time structures (*e.g.*, dispatch tables) must be checked to satisfy invariants required by the Cool run-time system (and thus Coolaid), and statically allocated objects must be type-checked.

Chapter 4

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, 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

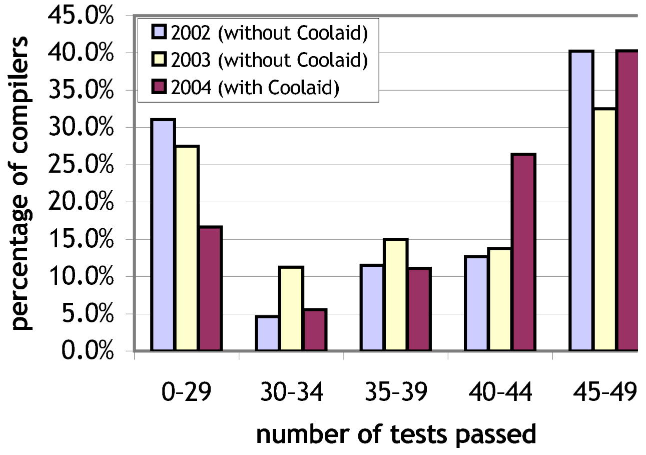


Figure 4.1: Performance of student compilers with and without Coolaid. The compilers are binned based on letter grades (for the automated testing component).

did not have Coolaid available. Each class was asked to complete the same project in the same amount of time.

4.1 Student Performance 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 4.1 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.

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 [CCNS05a, Sch04].

The breakdown of behaviors for the code produced by the student compilers is shown in Figure 4.2. 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 4.2.

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%

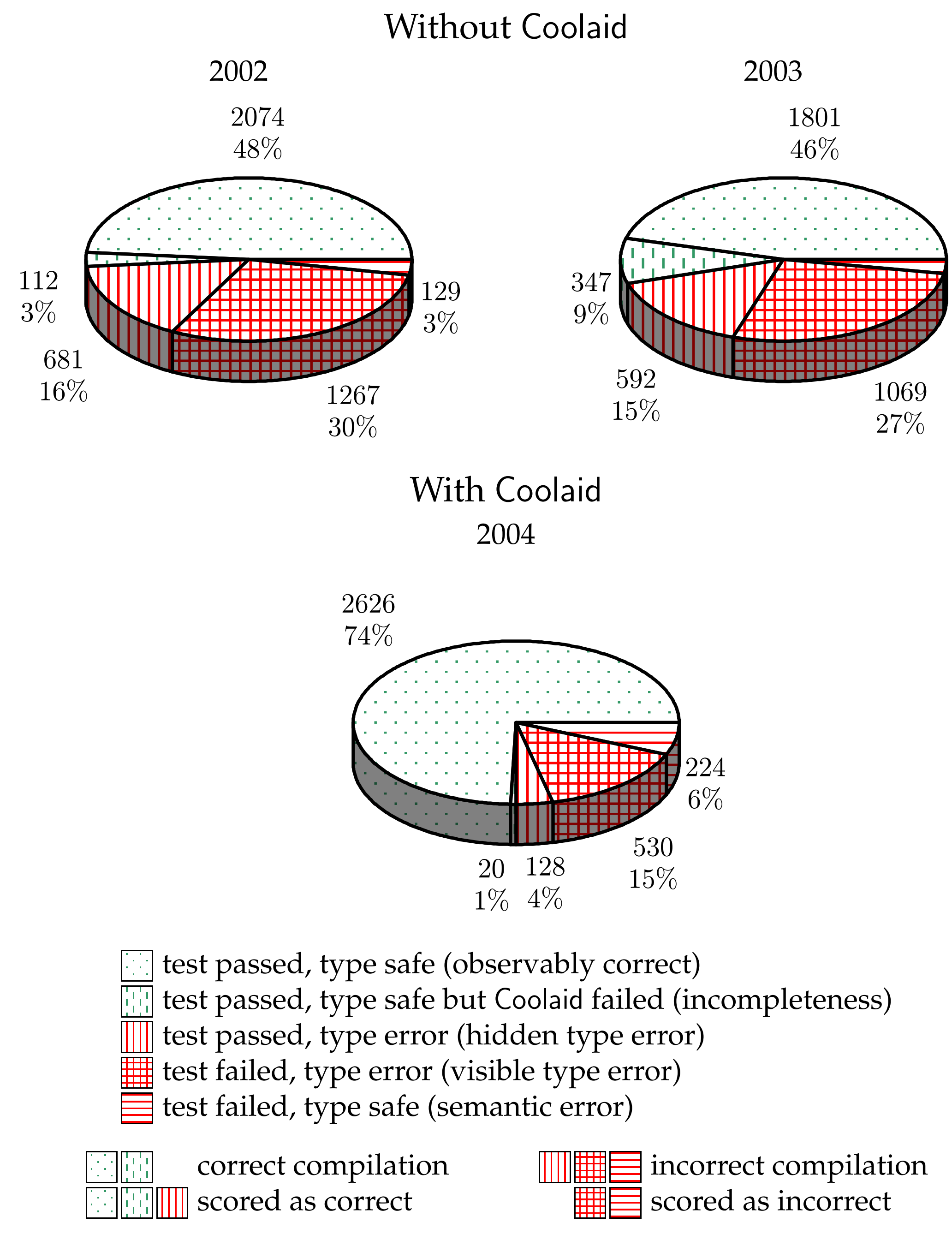


Figure 4.2: Behavior of test programs compiled by students and run through both the standard execution-based testing procedure and Coolaid. Horizontal lines indicate failing the standard testing procedure, while vertical lines (dotted or dashed) indicate failing Coolaid (and thus the grid pattern indicates failing both).

(▦/▦▤). 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 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. We do not, however, have any conclusive evidence to confirm this claim. In any case, this increase seems rather small compared to the overall benefits of reducing type errors.

4.2 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 4.3.

Common negative comments tended to involve either details about the user

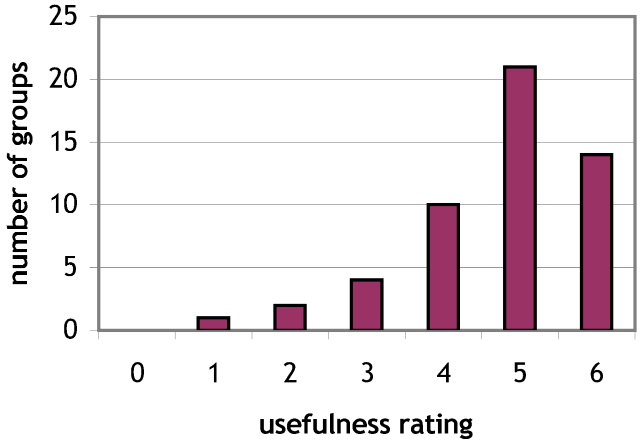


Figure 4.3: Student feedback from the 2004 class about the usefulness of Coolaid. 0 means "counterproductive" and 6 means "can't imagine developing/debugging a compiler without it."

interface, or the fact that Coolaid would not catch semantic errors that were not type errors. A favorite positive comment says,

> *"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."*

While it is difficult to measure whether the students have become better compiler writers through the use of Coolaid (as opposed to simply producing a better Cool compiler), this comment perhaps suggests positively that Coolaid helped students understand some key concepts in compiler development.

Chapter 5

Conclusion

5.1 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.

We seek generality through customizability; that is, we imagine that it will often be necessary to have customized verifiers for different languages or compilers. However, the task of building Coolaid-like verifiers should be greatly simplified by composing sub-verifiers, such as those for the stack and call-return abstraction. One might even imagine factoring the Coolaid-specific verifier into smaller pieces, such as for exception handling or dynamic dispatch.

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, which is possible because of the limits we place on our type system.

The symbolic values used in our abstract state serve a similar purpose as SSA-variables [CFR+91] for traditional compiler optimizations. Both techniques tease apart the re-use of registers into names that are more analogous to source-level variables. Others have made a similar observation (between SSA and functional programming [App98]). A difference is that we reflect these changes in the abstract state as we go rather than using a SSA-transformation pass, which has some benefit in the number of names needed. These two techniques are not necessarily incompatible though; one may also imagine a sub-verifier that uses symbolic evaluation to do a SSA-like transformation with symbolic values to simplify the work for higher-level verifiers.

Composing abstract domains has been studied by many since Cousot and Cousot defined the notion of a reduced product, which captures a notion of precise composition but requires domain-specific reduction operators [CC79]. Others have looked at obtaining more general combinations while trying to maintain precision [CCH94, LGC02, CL05]. Lerner *et al.* combine data-flow analyses for compiler optimizations using re-writing of statements to communicate between sub-analyses [LGC02], similar to the communication between our sub-verifiers; however, the re-writes in their case are all within the same language, while we consider different languages between each layer. A difference between our composition of verifiers and the above mentioned work on composing abstract domains is that we consider a composition of abstract interpreters interpreting different languages (with varying degrees of abstraction).

5.2 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 [CCNS05a].

Bibliography

[AC98] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1998.

[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.

[App98] Andrew W. Appel. Ssa is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998.

[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.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM Press.

[CCH94] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming. In *POPL*

'94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 227–239, New York, NY, USA, 1994. ACM Press.

[CCNS05a] 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.

[CCNS05b] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. Type-based verification of assembly language for compiler debugging. In *Proc. of the 2nd ACM Workshop on Types in Language Design and Implementation (TLDI'05)*, pages 91–102. ACM, January 2005.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[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.

[LGC02] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2002. ACM Press.

[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 Rules

In this chapter, we collect the rules that define the type-based abstract interpreter. Section 3.2 describe and explain the interesting rules for the stack and call-return verifier. Many of the Coolaid-specific rules are sketched through example in Section 2.3 and others are explained in Section 3.3.

A.1 Stack

$\boxed{I\colon S_p \longrightarrow_{\mathrm{s}} S'_p}$

$$\frac{S_p \mid I \Rightarrow_{\mathrm{s}} I_{\mathrm{s}} \triangleright S'_p \qquad I_{\mathrm{s}}\colon S'_p \twoheadrightarrow_{\mathrm{s}} S''_{p'}}{I\colon S_p \longrightarrow_{\mathrm{s}} S''_{p'}}\ \textsf{s-step}$$

A.1.1 Translation

$\boxed{S_p \mid I \Rightarrow_{\mathrm{s}} I_{\mathrm{s}} \triangleright S'_p}$

$$\frac{\Gamma_{\mathrm{s}} \vdash \Sigma_{\mathrm{s}}(ae) \Downarrow_{\mathrm{s}} \alpha + n \triangleright \Gamma'_{\mathrm{s}} \quad \Gamma'_{\mathrm{s}}(\alpha) = \textsf{sp0} \quad n_{\mathrm{lo}} \le n \le n_{\mathrm{hi}} \quad n \equiv 0 \pmod 4}{\langle \Sigma_{\mathrm{s}} \mathbin{⨾} \Gamma_{\mathrm{s}} \mathbin{⨾} n_{\mathrm{lo}} \mathbin{⨾} n_{\mathrm{hi}} \rangle \mid r := \mathtt{mem}[ae] \Rightarrow_{\mathrm{s}} r := \mathbf{r}_{\mathrm{sp}_0+n} \triangleright \langle \Sigma_{\mathrm{s}} \mathbin{⨾} \Gamma'_{\mathrm{s}} \mathbin{⨾} n_{\mathrm{lo}} \mathbin{⨾} n_{\mathrm{hi}} \rangle}\ \textsf{s-read}$$

$$\frac{\Gamma_{\mathrm{s}} \vdash \Sigma_{\mathrm{s}}(ae_0) \Downarrow_{\mathrm{s}} \alpha + n \triangleright \Gamma'_{\mathrm{s}} \quad \Gamma'_{\mathrm{s}}(\alpha) = \textsf{sp0} \quad n_{\mathrm{lo}} \le n \le n_{\mathrm{hi}} \quad n \equiv 0 \pmod 4}{\langle \Sigma_{\mathrm{s}} \mathbin{⨾} \Gamma'_{\mathrm{s}} \mathbin{⨾} n_{\mathrm{lo}} \mathbin{⨾} n_{\mathrm{hi}} \rangle \mid \mathtt{mem}[ae_0] := ae_1 \Rightarrow_{\mathrm{s}} \mathbf{r}_{\mathrm{sp}_0+n} := ae_1 \triangleright \langle \Sigma_{\mathrm{s}} \mathbin{⨾} \Gamma'_{\mathrm{s}} \mathbin{⨾} n_{\mathrm{lo}} \mathbin{⨾} n_{\mathrm{hi}} \rangle}\ \textsf{s-write}$$

$$\frac{}{S \mid I \Rightarrow_{\mathrm{s}} I \triangleright S}\ \textsf{s-iddecomp}$$

A.1.2 Transition

$$\boxed{I_{\mathrm{s}} \colon S_p \rightarrowtriangle_{\mathrm{s}} S'_{p'}}$$

$$\dfrac{}{L\colon\ : S \rightarrowtriangle_{\mathrm{s}} S}\ \textsf{s-label} \qquad \dfrac{}{\texttt{jump}\ L : S \rightarrowtriangle_{\mathrm{s}} S_L}\ \textsf{s-jump} \qquad \dfrac{}{r := ae : \langle \Sigma_{\mathrm{s}} \fatsemi \Gamma_{\mathrm{s}} \fatsemi n_{\mathrm{lo}} \fatsemi n_{\mathrm{hi}} \rangle \rightarrowtriangle_{\mathrm{s}} \langle \Sigma_{\mathrm{s}}[r \mapsto \Sigma_{\mathrm{s}}(ae)] \fatsemi \Gamma_{\mathrm{s}} \fatsemi n_{\mathrm{lo}} \fatsemi n_{\mathrm{hi}} \rangle}\ \textsf{s-set}$$

$$\dfrac{\begin{array}{l}\Gamma_{\mathrm{s}} \vdash \Sigma_{\mathrm{s}}(ae) \Downarrow_{\mathrm{s}} ((\alpha_0 + n_0) \oplus (\alpha_0 + n_1)) \ggg 20 \ \neq \ 0 \triangleright \Gamma'_{\mathrm{s}} \\ \Gamma'_{\mathrm{s}}(\alpha_0) = \textsf{sp0} \\ n_1 < n_{\mathrm{lo}} \leq n_0 \leq n_{\mathrm{hi}}\end{array}}{\texttt{branch}\ ae\ L : \langle \Sigma_{\mathrm{s}} \fatsemi \Gamma_{\mathrm{s}} \fatsemi n_{\mathrm{lo}} \fatsemi n_{\mathrm{hi}} \rangle_p \rightarrowtriangle_{\mathrm{s}} \langle \Sigma_{\mathrm{s}} \fatsemi \Gamma'_{\mathrm{s}} \fatsemi n_1 \fatsemi n_{\mathrm{hi}} \rangle_{p+1}}\ \textsf{s-sp}_F \qquad \dfrac{\begin{array}{l}\Gamma_{\mathrm{s}} \vdash \Sigma_{\mathrm{s}}(ae) \Downarrow_{\mathrm{s}} ((\alpha_0 + n_0) \oplus (\alpha_0 + n_1)) \ggg 20 \ = \ 0 \triangleright \Gamma'_{\mathrm{s}} \\ \Gamma'_{\mathrm{s}}(\alpha_0) = \textsf{sp0} \\ n_1 < n_{\mathrm{lo}} \leq n_0 \leq n_{\mathrm{hi}}\end{array}}{\texttt{branch}\ ae\ L : \langle \Sigma_{\mathrm{s}} \fatsemi \Gamma_{\mathrm{s}} \fatsemi n_{\mathrm{lo}} \fatsemi n_{\mathrm{hi}} \rangle \rightarrowtriangle_{\mathrm{s}} \langle \Sigma_{\mathrm{s}} \fatsemi \Gamma'_{\mathrm{s}} \fatsemi n_1 \fatsemi n_{\mathrm{hi}} \rangle_L}\ \textsf{s-sp}_T$$

$$\dfrac{}{\texttt{branch}\ ae\ L : S \rightarrowtriangle_{\mathrm{s}} S}\ \textsf{s-branch}_F \qquad \dfrac{}{\texttt{branch}\ ae\ L : S \rightarrowtriangle_{\mathrm{s}} S_L}\ \textsf{s-branch}_T$$

A.2 Call-Return

$$\boxed{\begin{array}{l} I \colon F_p \longrightarrow_{\mathrm{f}} F'_p \\ I \colon F_p\ \ \textsf{ok} \end{array}}$$

$$\dfrac{F_p \mid I \Rightarrow_{\mathrm{f}} I_{\mathrm{f}} \triangleright F'_p \qquad I_{\mathrm{f}} \colon F'_p \rightarrowtriangle_{\mathrm{f}} F''_{p'}}{I \colon F_p \longrightarrow_{\mathrm{f}} F''_{p'}}\ \textsf{f-step} \qquad \dfrac{F_p \mid I \Rightarrow_{\mathrm{f}} I_{\mathrm{f}} \triangleright F'_p \qquad I_{\mathrm{f}} \colon F'_p\ \ \textsf{ok}}{I \colon F_p\ \ \textsf{ok}}\ \textsf{f-ok}$$

A.2.1 Translation

$$\boxed{\begin{array}{l} F_p \mid I \Rightarrow_{\mathrm{f}} I_{\mathrm{f}} \triangleright F'_p \\ F_p \mid I_{\mathrm{s}} \dot{\Rightarrow}_{\mathrm{f}} I_{\mathrm{f}} \triangleright F'_p \end{array}}$$

$$\dfrac{S \mid I \Rightarrow_{\mathrm{s}} I_{\mathrm{s}} \triangleright S' \qquad \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S' \rangle \mid I_{\mathrm{s}} \dot{\Rightarrow}_{\mathrm{f}} I_{\mathrm{f}} \triangleright \langle \Sigma'_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n'_{\mathrm{pop}} \fatsemi S'' \rangle}{\langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle \mid I \Rightarrow_{\mathrm{f}} I_{\mathrm{f}} \triangleright \langle \Sigma'_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n'_{\mathrm{pop}} \fatsemi S'' \rangle}\ \textsf{f-decomp}$$

$$\dfrac{\Gamma_{\mathrm{f}} \vdash \Sigma_{\mathrm{f}}(ae) \Downarrow_{\mathrm{f}} \alpha \triangleright \Gamma'_{\mathrm{f}} \quad \Gamma'_{\mathrm{f}}(\alpha) = \textsf{ra}}{\langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle \mid \ \texttt{jump}\ [ae]\ \dot{\Rightarrow}_{\mathrm{f}}\ \texttt{return}\ \mathbf{r}_{rv}\ \triangleright \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle}\ \textsf{f-return}$$

$$\dfrac{\begin{array}{c}\Gamma_{\mathrm{f}} \vdash \Sigma_{\mathrm{f}}(\mathbf{r}_{ra}) : \textsf{codeaddr}(L') \triangleright \Gamma'_{\mathrm{f}} \qquad (\&L' = p+1) \\ e_i = \begin{cases} \mathbf{r}_{sp} + npop(L) & \text{if } r_i \text{ is } \mathbf{r}_{sp} \\ r_i & \text{if } r_i \text{ is preserved by the callee} \\ ? & \text{otherwise} \end{cases}\end{array}}{\langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p \mid \ \texttt{jump}\ L\ \dot{\Rightarrow}_{\mathrm{f}}\ \texttt{call}\ L(\mathbf{r}_{arg_0}, \ldots, \mathbf{r}_{arg_\ell}) : \mathbf{r}_{rv}, \overrightarrow{r := e}\ \triangleright \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p}\ \textsf{f-call}$$

$$\dfrac{\begin{array}{c}\Gamma_{\mathrm{f}} \vdash \Sigma_{\mathrm{f}}(\mathbf{r}_{ra}) : \textsf{codeaddr}(L') \triangleright \Gamma'_{\mathrm{f}} \qquad (\&L' = p+1) \\ e_i = \begin{cases} r_i & \text{if } r_i \text{ is preserved by the callee} \\ ? & \text{otherwise} \end{cases}\end{array}}{\langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p \mid \ \texttt{jump}\ [ae]\ \dot{\Rightarrow}_{\mathrm{f}}\ \texttt{call}\ [ae] : \mathbf{r}_{rv}, \overrightarrow{r := e}\ \triangleright \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle_p}\ \textsf{f-icall}$$

$$\frac{}{F \mid I_{\mathrm{s}} \dot{\Rightarrow}_{\mathrm{f}} I_{\mathrm{s}} \triangleright F}\;\textsf{f-iddecomp}$$

A.2.2 Transition

$$\boxed{\begin{array}{l} I_{\mathrm{f}} \colon F_p \rightarrowtriangle_{\mathrm{f}} F'_{p'} \\ I_{\mathrm{f}} \colon F_p \;\; \textsf{ok} \end{array}}$$

$$\frac{\begin{array}{l}\Gamma_{\mathrm{f}} \vdash \Sigma_{\mathrm{f}}(r_{cs}) \Downarrow_{\mathrm{f}} \alpha \triangleright \Gamma'_{\mathrm{f}} \qquad \Gamma'_{\mathrm{f}}(\alpha) = \textsf{cs}(r_{cs}) \qquad \text{(for all callee-save } r_{cs}) \\ S.\Gamma_{\mathrm{s}} \vdash S.\Sigma_{\mathrm{s}}(\mathbf{r}_{sp}) \Downarrow_{\mathrm{s}} \alpha + n_{\mathrm{pop}} \triangleright \Gamma'_{\mathrm{s}} \qquad \Gamma'_{\mathrm{s}}(\alpha) = \textsf{sp0}\end{array}}{\texttt{return}\; ae : \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle \;\; \textsf{ok}}\;\textsf{f-returnok}$$

$$\frac{\begin{array}{l} S.\Gamma_{\mathrm{s}} \vdash S.\Sigma_{\mathrm{s}}(\mathbf{r}_{sp}) \Downarrow_{\mathrm{s}} \alpha + n \triangleright \Gamma^0_{\mathrm{s}} \quad \Gamma^0_{\mathrm{s}}(\alpha) = \textsf{sp0} \quad n \equiv 0 \pmod 4 \\ S^0 = \langle S.\Sigma_{\mathrm{s}} \fatsemi \Gamma^0_{\mathrm{s}} \fatsemi S.n_{\mathrm{lo}} \fatsemi S.n_{\mathrm{hi}} \rangle \\ n + 4 \cdot \mathit{nargs}(L) \leq S^0.n_{\mathrm{hi}} \quad n + 4 \cdot \mathit{nargs}(L) - \mathit{framesize}(L) \geq S^0.n_{\mathrm{lo}} \\ r_i := e_i : \langle \Sigma^i_{\mathrm{f}} \fatsemi \Gamma^i_{\mathrm{f}} \fatsemi n^i_{\mathrm{pop}} \fatsemi S^i \rangle \rightarrowtriangle_{\mathrm{f}} \langle \Sigma^{i+1}_{\mathrm{f}} \fatsemi \Gamma^{i+1}_{\mathrm{f}} \fatsemi n^{i+1}_{\mathrm{pop}} \fatsemi S^{i+1} \rangle \quad (\text{for } 0 \leq i < k) \end{array}}{\texttt{call}\; L(r_{arg_0}, \ldots, r_{arg_\ell}) : r_{rv}, \overrightarrow{r := e} : \langle \Sigma^0_{\mathrm{f}} \fatsemi \Gamma^0_{\mathrm{f}} \fatsemi n^0_{\mathrm{pop}} \fatsemi S \rangle \rightarrowtriangle_{\mathrm{f}} \langle \Sigma^k_{\mathrm{f}} \fatsemi \Gamma^k_{\mathrm{f}} \fatsemi n^k_{\mathrm{pop}} \fatsemi S^k \rangle}\;\textsf{f-callok}$$

$$\frac{r := ae : S \rightarrowtriangle_{\mathrm{s}} S'}{r := ae : \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle \rightarrowtriangle_{\mathrm{f}} \langle \Sigma_{\mathrm{f}}[r \mapsto \Sigma_{\mathrm{f}}(ae)] \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S' \rangle}\;\textsf{f-set}$$

$$\frac{I_{\mathrm{f}} \colon S \rightarrowtriangle_{\mathrm{s}} S'}{I_{\mathrm{f}} \colon \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S \rangle \rightarrowtriangle_{\mathrm{f}} \langle \Sigma_{\mathrm{f}} \fatsemi \Gamma_{\mathrm{f}} \fatsemi n_{\mathrm{pop}} \fatsemi S' \rangle}\;\textsf{f-follow}$$

A.2.3 Typing

$$\boxed{\Gamma_{\mathrm{f}} \vdash e : \tau_{\mathrm{f}} \triangleright \Gamma'_{\mathrm{f}}}$$

$$\frac{\Gamma_{\mathrm{f}} \vdash e \Downarrow_{\mathrm{f}} \alpha \triangleright \Gamma'_{\mathrm{f}} \quad \Gamma'(\alpha) = \tau_{\mathrm{f}}}{\Gamma_{\mathrm{f}} \vdash e : \tau_{\mathrm{f}} \triangleright \Gamma'_{\mathrm{f}}}\;\textsf{f-var}$$

A.3 Coolaid

$$\boxed{\begin{array}{l} I \colon A_p \rightarrow A'_p \\ I \colon A_p \;\; \textsf{ok} \end{array}}$$

$$\frac{A_p \mid I \Rightarrow I_{\mathrm{c}} \triangleright A'_p \quad I_{\mathrm{c}} \colon A'_p \rightarrowtriangle A''_{p'}}{I \colon A_p \rightarrow A''_{p'}}\;\textsf{c-step} \qquad \frac{A_p \mid I \Rightarrow I_{\mathrm{c}} \triangleright A'_p \quad I_{\mathrm{c}} \colon A'_p \;\; \textsf{ok}}{I \colon A_p \;\; \textsf{ok}}\;\textsf{c-ok}$$

A.3.1 Translation

$$\boxed{\begin{array}{l} A_p \mid I \Rightarrow I_{\mathrm{c}} \rhd A'_p \\ A_p \mid I_{\mathrm{f}} \dot{\Rightarrow} I_{\mathrm{c}} \rhd A'_p \end{array}}$$

$$\frac{F_p \mid I \Rightarrow_{\mathrm{f}} I_{\mathrm{f}} \rhd F'_p \qquad \langle \Sigma \fatsemi \Gamma \fatsemi F' \rangle_p \mid I_{\mathrm{f}} \dot{\Rightarrow} I_{\mathrm{c}} \rhd \langle \Sigma' \fatsemi \Gamma' \fatsemi F'' \rangle_{p'}}{\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle_p \mid I \Rightarrow I_{\mathrm{c}} \rhd \langle \Sigma' \fatsemi \Gamma' \fatsemi F'' \rangle_{p'}}\ \textsf{c-decomp}$$

$$\frac{\Gamma \vdash \Sigma(ae) : \tau\ \mathsf{ptr} \rhd \Gamma'}{\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid r := \mathtt{mem}[ae] \ \dot{\Rightarrow}\ r := ?_\tau \ \rhd \langle \Sigma \fatsemi \Gamma' \fatsemi F \rangle}\ \textsf{c-read}
\qquad
\frac{\Gamma \vdash \Sigma(ae_0) : \tau\ \mathsf{ptr} \rhd \Gamma' \quad \Gamma' \vdash \Sigma(ae_1) : \tau \rhd \Gamma''}{\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid \mathtt{mem}[ae_0] := ae_1 \ \dot{\Rightarrow}\ \mathtt{noop} \ \rhd \langle \Sigma \fatsemi \Gamma'' \fatsemi F \rangle}\ \textsf{c-write}$$

$$\frac{\begin{array}{l}
\Gamma \vdash \Sigma(ae) : \mathsf{meth}(\alpha, n) \rhd \Gamma' \\
\Sigma(\mathbf{r}_{arg_0}) = \alpha \qquad \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma'' \\
T(C) = \mathtt{class}\ C\ \ldots \{ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots \} \\
(m \text{ is the method at offset } n \text{ of class } C) \\
\Gamma'' \vdash \Sigma(\mathbf{r}_{xp}) : \mathsf{excframe} \rhd \Gamma'''
\end{array}}{\begin{array}{l}
\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid \mathtt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\
\quad \dot{\Rightarrow}\ \mathtt{invokevirtual}\ C.m(\mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_\ell}) : \\
\qquad\qquad r_{rv},\ \overrightarrow{r := e}\ [\!]\ \mathsf{incaller}\ \rhd \langle \Sigma \fatsemi \Gamma''' \fatsemi F \rangle
\end{array}}\ \textsf{c-meth}$$

$$\frac{\begin{array}{l}
\Gamma \vdash \Sigma(ae) : \mathsf{meth}(\alpha, n) \rhd \Gamma' \\
\Sigma(\mathbf{r}_{arg_0}) = \alpha \qquad \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma'' \\
T(C) = \mathtt{class}\ C\ \ldots \{ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots \} \\
(m \text{ is the method at offset } n \text{ of class } C) \\
F.S.\Gamma_{\mathrm{s}} \vdash F.S.\Sigma_{\mathrm{s}}(\mathbf{r}_{xp}) \Downarrow_{\mathrm{s}} \alpha + n_{xp} \rhd \Gamma'_{\mathrm{s}} \qquad \Gamma'_{\mathrm{s}}(\alpha) = \mathsf{sp0} \\
F.\Gamma_{\mathrm{f}} \vdash F.\Sigma_{\mathrm{f}}(\mathbf{r}_{sp_0 + n_{xp}}) :_{\mathrm{f}} \mathsf{codeaddr}(L) \rhd \Gamma'_{\mathrm{f}} \\
F' = \langle F.\Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi F.n_{\mathrm{pop}} \fatsemi \langle F.S.\Sigma_{\mathrm{s}} \fatsemi \Gamma'_{\mathrm{s}} \fatsemi F.S.n_{\mathrm{lo}} \fatsemi F.S.n_{\mathrm{hi}} \rangle \rangle \\
e'_i = \begin{cases} r'_i & \text{if } r'_i \text{ is preserved by the callee} \\ ? & \text{otherwise} \end{cases}
\end{array}}{\begin{array}{l}
\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid \mathtt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\
\quad \dot{\Rightarrow}\ \mathtt{invokevirtual}\ C.m(\mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_\ell}) : \\
\qquad\qquad r_{rv},\ \overrightarrow{r\ := e}\ [\!]\ \mathbf{r}_{rv},\ \overrightarrow{r' := e'}\ @L\ \rhd \langle \Sigma \fatsemi \Gamma'' \fatsemi F' \rangle
\end{array}}\ \textsf{c-methx}$$

$$\frac{\begin{array}{l}
\Gamma \vdash \Sigma(ae) : \mathsf{smeth}(C, n) \rhd \Gamma' \\
\Gamma' \vdash \Sigma(\mathbf{r}_{arg_0}) : \mathsf{nonnull}\ C \rhd \Gamma'' \\
T(C) = \mathtt{class}\ C\ \ldots \{ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots \} \\
(m \text{ is the method at offset } n \text{ of class } C) \\
\Gamma'' \vdash \Sigma(\mathbf{r}_{xp}) : \mathsf{excframe} \rhd \Gamma'''
\end{array}}{\begin{array}{l}
\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid \mathtt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\
\quad \dot{\Rightarrow}\ \mathtt{invokesmeth}\ C.m(\mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_\ell}) : \\
\qquad\qquad r_{rv},\ \overrightarrow{r := e}\ [\!]\ \mathsf{incaller}\ \rhd \langle \Sigma \fatsemi \Gamma''' \fatsemi F \rangle
\end{array}}\ \textsf{c-smeth}$$

$$\frac{\begin{array}{l}
\Gamma \vdash \Sigma(ae) : \mathsf{smeth}(C, n) \rhd \Gamma' \\
\Gamma' \vdash \Sigma(\mathbf{r}_{arg_0}) : \mathsf{nonnull}\ C \rhd \Gamma'' \\
T(C) = \mathtt{class}\ C\ \ldots \{ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots \} \\
(m \text{ is the method at offset } n \text{ of class } C) \\
F.S.\Gamma_{\mathrm{s}} \vdash F.S.\Sigma_{\mathrm{s}}(\mathbf{r}_{xp}) \Downarrow_{\mathrm{s}} \alpha + n_{xp} \rhd \Gamma'_{\mathrm{s}} \qquad \Gamma'_{\mathrm{s}}(\alpha) = \mathsf{sp0} \\
F.\Gamma_{\mathrm{f}} \vdash F.\Sigma_{\mathrm{f}}(\mathbf{r}_{sp_0 + n_{xp}}) :_{\mathrm{f}} \mathsf{codeaddr}(L) \rhd \Gamma'_{\mathrm{f}} \\
F' = \langle F.\Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi F.n_{\mathrm{pop}} \fatsemi \langle F.S.\Sigma_{\mathrm{s}} \fatsemi \Gamma'_{\mathrm{s}} \fatsemi F.S.n_{\mathrm{lo}} \fatsemi F.S.n_{\mathrm{hi}} \rangle \rangle \\
e'_i = \begin{cases} r'_i & \text{if } r'_i \text{ is preserved by the callee} \\ ? & \text{otherwise} \end{cases}
\end{array}}{\begin{array}{l}
\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid \mathtt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\
\quad \dot{\Rightarrow}\ \mathtt{invokesmeth}\ C.m(\mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_\ell}) : \\
\qquad\qquad r_{rv},\ \overrightarrow{r\ := e}\ [\!]\ \mathbf{r}_{rv},\ \overrightarrow{r' := e'}\ @L\ \rhd \langle \Sigma \fatsemi \Gamma'' \fatsemi F' \rangle
\end{array}}\ \textsf{c-smethx}$$

$$\frac{\begin{array}{l}
\Gamma \vdash \Sigma(ae) : \mathsf{init}(\alpha) \rhd \Gamma' \\
\Sigma(\mathbf{r}_{arg_0}) = \beta \qquad \Gamma' \vdash \beta : \mathsf{nonnull\ classof}(\alpha) \rhd \Gamma'' \\
\Gamma'' \vdash \Sigma(\mathbf{r}_{xp}) : \mathsf{excframe} \rhd \Gamma'''
\end{array}}{\begin{array}{l}
\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid \mathtt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\
\quad \dot{\Rightarrow}\ \mathtt{invokeinit}\ C : \\
\qquad\qquad \mathbf{r}_{arg_0},\ \overrightarrow{r := e}\ [\!]\ \mathsf{incaller}\ \rhd \langle \Sigma \fatsemi \Gamma''' \fatsemi F \rangle
\end{array}}\ \textsf{c-init}$$

$$\frac{\begin{array}{l}
\Gamma \vdash \Sigma(ae) : \mathsf{init}(\alpha) \rhd \Gamma' \\
\Sigma(\mathbf{r}_{arg_0}) = \beta \qquad \Gamma' \vdash \beta : \mathsf{nonnull\ classof}(\alpha) \rhd \Gamma'' \\
F.S.\Gamma_{\mathrm{s}} \vdash F.S.\Sigma_{\mathrm{s}}(\mathbf{r}_{xp}) \Downarrow_{\mathrm{s}} \alpha + n_{xp} \rhd \Gamma'_{\mathrm{s}} \qquad \Gamma'_{\mathrm{s}}(\alpha) = \mathsf{sp0} \\
F.\Gamma_{\mathrm{f}} \vdash F.\Sigma_{\mathrm{f}}(\mathbf{r}_{sp_0 + n_{xp}}) :_{\mathrm{f}} \mathsf{codeaddr}(L) \rhd \Gamma'_{\mathrm{f}} \\
F' = \langle F.\Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi F.n_{\mathrm{pop}} \fatsemi \langle F.S.\Sigma_{\mathrm{s}} \fatsemi \Gamma'_{\mathrm{s}} \fatsemi F.S.n_{\mathrm{lo}} \fatsemi F.S.n_{\mathrm{hi}} \rangle \rangle \\
e'_i = \begin{cases} r'_i & \text{if } r'_i \text{ is preserved by the callee} \\ ? & \text{otherwise} \end{cases}
\end{array}}{\begin{array}{l}
\langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \mid \mathtt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\
\quad \dot{\Rightarrow}\ \mathtt{invokeinit}\ C : \\
\qquad\qquad \mathbf{r}_{arg_0},\ \overrightarrow{r\ := e}\ [\!]\ \mathbf{r}_{rv},\ \overrightarrow{r' := e'}\ @L\ \rhd \langle \Sigma \fatsemi \Gamma'' \fatsemi F' \rangle
\end{array}}\ \textsf{c-initx}$$

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma(ae) : \mathsf{sinit}(C) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \beta \qquad \Gamma' \vdash \beta : \mathsf{nonnull}\ C \rhd \Gamma'' \\ \Gamma'' \vdash \Sigma(\mathbf{r}_{xp}) : \mathsf{excframe} \rhd \Gamma''' \end{array}}{\begin{array}{l}\langle \Sigma \fatsemi \Gamma \fatsemi F\rangle \wr\ \texttt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\ \quad \dot{\Rightarrow}\ \texttt{invokeinit}\ C : \\ \qquad\qquad \mathbf{r}_{arg_0},\ \overrightarrow{r := e}\ [\!]\ \mathsf{incaller}\ \rhd \langle \Sigma \fatsemi \Gamma''' \fatsemi F\rangle\end{array}}\ \textsf{c-sinit}$$

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma(ae) : \mathsf{sinit}(C) \rhd \Gamma' \\ \Sigma(\mathbf{r}_{arg_0}) = \beta \qquad \Gamma' \vdash \beta : \mathsf{nonnull}\ C \rhd \Gamma'' \\ F.S.\Gamma_{\mathrm{s}} \vdash F.S.\Sigma_{\mathrm{s}}(\mathbf{r}_{xp}) \Downarrow_{\mathrm{s}} \alpha + n_{xp} \rhd \Gamma'_{\mathrm{s}} \qquad \Gamma'_{\mathrm{s}}(\alpha) = \mathsf{sp0} \\ F.\Gamma_{\mathrm{f}} \vdash F.\Sigma_{\mathrm{f}}(\mathbf{r}_{sp_0+n_{xp}}) :_{\mathrm{f}} \mathsf{codeaddr}(L) \rhd \Gamma'_{\mathrm{f}} \\ F' = \langle F.\Sigma_{\mathrm{f}} \fatsemi \Gamma'_{\mathrm{f}} \fatsemi F.n_{\mathrm{pop}} \fatsemi \langle F.S.\Sigma_{\mathrm{s}} \fatsemi \Gamma'_{\mathrm{s}} \fatsemi F.S.n_{\mathrm{lo}} \fatsemi F.S.n_{\mathrm{hi}}\rangle\rangle \\ e'_i = \begin{cases} r'_i & \text{if } r'_i \text{ is preserved by the callee} \\ ? & \text{otherwise}\end{cases}\end{array}}{\begin{array}{l}\langle \Sigma \fatsemi \Gamma \fatsemi F\rangle \wr\ \texttt{call}\ [ae] : r_{rv},\ \overrightarrow{r := e} \\ \quad \dot{\Rightarrow}\ \texttt{invokeinit}\ C : \\ \qquad\qquad \mathbf{r}_{arg_0},\ \overrightarrow{r\ := e}\ [\!]\ \mathbf{r}_{rv},\ \overrightarrow{r' := e'}\ @L\ \rhd \langle \Sigma \fatsemi \Gamma'' \fatsemi F'\rangle\end{array}}\ \textsf{c-sinitx}$$

$$\frac{\Gamma \vdash \Sigma(ae) : \mathsf{handler} \rhd \Gamma'}{\langle \Sigma \fatsemi \Gamma \fatsemi F\rangle \wr\ \texttt{jump}\ [ae]\ \dot{\Rightarrow}\ \texttt{throw}\ \mathbf{r}_{rv}\ \rhd \langle \Sigma \fatsemi \Gamma' \fatsemi F\rangle}\ \textsf{c-throw}$$

$$\frac{}{A \wr I_{\mathrm{f}} \dot{\Rightarrow} I_{\mathrm{f}} \rhd A}\ \textsf{c-iddecomp}$$

A.3.2 Transition

$$\boxed{\begin{array}{l} I_{\mathrm{c}} \colon A_p \rightarrowtriangle A'_{p'} \\ I_{\mathrm{c}} \colon A_p\ \ \mathsf{ok}\end{array}}$$

$$\frac{\Gamma \vdash ae : \tau_{rv} \rhd \Gamma' \qquad \texttt{return}\ ae : F\ \ \mathsf{ok}}{\texttt{return}\ ae : \langle \Sigma \fatsemi \Gamma \fatsemi F\rangle\ \ \mathsf{ok}}\ \textsf{c-returnok} \qquad \frac{\Gamma \vdash \Sigma(\mathbf{r}_{xp}) : \mathsf{excframe} \rhd \Gamma' \quad \Gamma' \vdash \Sigma(ae) : \texttt{Object} \rhd \Gamma''}{\texttt{throw}\ ae : \langle \Sigma \fatsemi \Gamma \fatsemi F\rangle\ \ \mathsf{ok}}\ \textsf{c-throwok}$$

$$\frac{\begin{array}{l} T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \qquad \Gamma_0 \vdash \Sigma_0(r_{arg_1}) : \tau_1 \rhd \Gamma_1\ \ \cdots\ \ \Gamma_{\ell-1} \vdash \Sigma_0(r_{arg_\ell}) : \tau_\ell \rhd \Gamma_\ell \\ r_i := e_i : \langle \Sigma_i \fatsemi \Gamma_{\ell+i} \fatsemi F_i\rangle \rightarrowtriangle \langle \Sigma_{i+1} \fatsemi \Gamma_{\ell+i+1} \fatsemi F_{i+1}\rangle \quad (\text{for } 0 \le i < k) \\ \texttt{call}\ m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e} : F_0 \rightarrowtriangle_{\mathrm{f}} F' \hfill (\beta \text{ fresh})\end{array}}{\begin{array}{l}\texttt{invokevirtual}\ C.m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e}\ [\!]\ \mathit{exchandler} \\ \qquad : \langle \Sigma_0 \fatsemi \Gamma_0 \fatsemi F_0\rangle \rightarrowtriangle \langle \Sigma_k[r_{rv} \mapsto \beta] \fatsemi \Gamma_{\ell+k}[\beta \mapsto \tau_{rv}] \fatsemi F'\rangle\end{array}}\ \textsf{c-methok}$$

$$\frac{\begin{array}{l} T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \qquad \Gamma_0 \vdash \Sigma_0(r_{arg_1}) : \tau_1 \rhd \Gamma_1\ \ \cdots\ \ \Gamma_{\ell-1} \vdash \Sigma_0(r_{arg_\ell}) : \tau_\ell \rhd \Gamma_\ell \\ r'_i := e'_i : \langle \Sigma_i \fatsemi \Gamma_{\ell+i} \fatsemi F_i\rangle \rightarrowtriangle \langle \Sigma_{i+1} \fatsemi \Gamma_{\ell+i+1} \fatsemi F_{i+1}\rangle \quad (\text{for } 0 \le i < k) \hfill (\beta \text{ fresh})\end{array}}{\begin{array}{l}\texttt{invokevirtual}\ C.m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r\ := e}\ [\!]\ r_{xv},\ \overrightarrow{r' := e'}\ @L \\ \qquad : \langle \Sigma_0 \fatsemi \Gamma \fatsemi F_0\rangle \rightarrowtriangle \langle \Sigma_k[r_{xv} \mapsto \beta] \fatsemi \Gamma_{\ell+k}[\beta \mapsto \texttt{Object}] \fatsemi F_k\rangle_L\end{array}}\ \textsf{c-methxok}$$

$$\frac{\begin{array}{l} T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \qquad \Gamma_0 \vdash \Sigma_0(r_{arg_1}) : \tau_1 \rhd \Gamma_1\ \ \cdots\ \ \Gamma_{\ell-1} \vdash \Sigma_0(r_{arg_\ell}) : \tau_\ell \rhd \Gamma_\ell \\ r_i := e_i : \langle \Sigma_i \fatsemi \Gamma_{\ell+i} \fatsemi F_i\rangle \rightarrowtriangle \langle \Sigma_{i+1} \fatsemi \Gamma_{\ell+i+1} \fatsemi F_{i+1}\rangle \quad (\text{for } 0 \le i < k) \\ \texttt{call}\ m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e} : F_0 \rightarrowtriangle_{\mathrm{f}} F' \hfill (\beta \text{ fresh})\end{array}}{\begin{array}{l}\texttt{invokesmeth}\ C.m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r := e}\ [\!]\ \mathit{exchandler} \\ \qquad : \langle \Sigma_0 \fatsemi \Gamma_0 \fatsemi F_0\rangle \rightarrowtriangle \langle \Sigma_k[r_{rv} \mapsto \beta] \fatsemi \Gamma_{\ell+k}[\beta \mapsto \tau_{rv}] \fatsemi F'\rangle\end{array}}\ \textsf{c-smethok}$$

$$\frac{\begin{array}{l} T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau_{rv}\ m(\tau_1, \ldots, \tau_\ell)\ \ldots\ \} \qquad \Gamma_0 \vdash \Sigma_0(r_{arg_1}) : \tau_1 \rhd \Gamma_1\ \ \cdots\ \ \Gamma_{\ell-1} \vdash \Sigma_0(r_{arg_\ell}) : \tau_\ell \rhd \Gamma_\ell \\ r'_i := e'_i : \langle \Sigma_i \fatsemi \Gamma_{\ell+i} \fatsemi F_i\rangle \rightarrowtriangle \langle \Sigma_{i+1} \fatsemi \Gamma_{\ell+i+1} \fatsemi F_{i+1}\rangle \quad (\text{for } 0 \le i < k) \hfill (\beta \text{ fresh})\end{array}}{\begin{array}{l}\texttt{invokesmeth}\ C.m(r_{arg_1}, \ldots, r_{arg_\ell}) : r_{rv},\ \overrightarrow{r\ := e}\ [\!]\ r_{xv},\ \overrightarrow{r' := e'}\ @L \\ \qquad : \langle \Sigma_0 \fatsemi \Gamma \fatsemi F_0\rangle \rightarrowtriangle \langle \Sigma_k[r_{xv} \mapsto \beta] \fatsemi \Gamma_{\ell+k}[\beta \mapsto \texttt{Object}] \fatsemi F_k\rangle_L\end{array}}\ \textsf{c-smethxok}$$

$$\frac{\begin{array}{l} r_i := e_i \,:\, \langle \Sigma_i \fatsemi \Gamma_{\ell+i} \fatsemi F_i \rangle \rightarrowtriangle \langle \Sigma_{i+1} \fatsemi \Gamma_{\ell+i+1} \fatsemi F_{i+1} \rangle \quad (\text{for } 0 \le i < k) \\ \texttt{call}\ C.\mathit{init}(r_{arg_0}) : r_{arg_0},\ \overrightarrow{r := e}\ :\ F_0 \rightarrowtriangle_{\mathrm{f}} F' \end{array}}{\begin{array}{l} \texttt{invokeinit}\ C : r_{arg_0},\ \overrightarrow{r := e}\ [\!]\ \mathit{exchandler} \\ \qquad\qquad : \langle \Sigma_0 \fatsemi \Gamma_0 \fatsemi F_0 \rangle \rightarrowtriangle \langle \Sigma_k[r_{arg_0} \mapsto \Sigma_0(r_{arg_0})] \fatsemi \Gamma_{\ell+k} \fatsemi F' \rangle \end{array}} \text{ c-initok}$$

$$\frac{r'_i := e'_i \,:\, \langle \Sigma_i \fatsemi \Gamma_{\ell+i} \fatsemi F_i \rangle \rightarrowtriangle \langle \Sigma_{i+1} \fatsemi \Gamma_{\ell+i+1} \fatsemi F_{i+1} \rangle \quad (\text{for } 0 \le i < k) \qquad (\beta \text{ fresh})}{\begin{array}{l} \texttt{invokeinit}\ C : r_{arg_0},\ \overrightarrow{r\ := e}\ [\!]\ r_{xv},\ \overrightarrow{r' := e'}\ @L \\ \qquad\qquad : \langle \Sigma_0 \fatsemi \Gamma \fatsemi F_0 \rangle \rightarrowtriangle \langle \Sigma_k[r_{xv} \mapsto \beta] \fatsemi \Gamma_{\ell+k}[\beta \mapsto \texttt{Object}] \fatsemi F_k \rangle_L \end{array}} \text{ c-initxok}$$

$$\frac{r := ?\ :\ F \rightarrowtriangle_{\mathrm{f}} F' \quad (\alpha \text{ fresh})}{r := ?_\tau\ :\ \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \rightarrowtriangle \langle \Sigma[r \mapsto \alpha] \fatsemi \Gamma[\alpha \mapsto \tau] \fatsemi F' \rangle} \text{ c-setty} \qquad \frac{r := ae\ :\ F \rightarrowtriangle_{\mathrm{f}} F'}{r := ae\ :\ \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \rightarrowtriangle \langle \Sigma[r \mapsto \Sigma(ae)] \fatsemi \Gamma \fatsemi F' \rangle} \text{ c-set}$$

$$\frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ k \rhd \Gamma' \qquad \Gamma'(\alpha) = \mathsf{tag}(\beta, N) \qquad \Gamma'(\beta) = \mathsf{nonnull}\ C \qquad N' = \{n \in N \mid \neg(n\ R\ k)\} \ne \emptyset \\ \texttt{branch}\ ae\ L\ :\ F \rightarrowtriangle_{\mathrm{f}} F' \end{array}}{\texttt{branch}\ ae\ L\ :\ \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle_p \rightarrowtriangle \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \mathsf{tag}(\beta, N')][\beta \mapsto \mathsf{nonnull}\ \mathit{taglub}(N')] \fatsemi F' \rangle_{p+1}} \text{ c-refinetag}_F$$

$$\frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ k \rhd \Gamma' \qquad \Gamma'(\alpha) = \mathsf{tag}(\beta, N) \qquad \Gamma'(\beta) = \mathsf{nonnull}\ C \qquad N' = \{n \in N \mid n\ R\ k\} \ne \emptyset \\ \texttt{branch}\ ae\ L\ :\ F \rightarrowtriangle_{\mathrm{f}} F' \end{array}}{\texttt{branch}\ ae\ L\ :\ \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \rightarrowtriangle \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \mathsf{tag}(\beta, N')][\beta \mapsto \mathsf{nonnull}\ \mathit{taglub}(N')] \fatsemi F' \rangle_L} \text{ c-refinetag}_T$$

$$\frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ 0 \rhd \Gamma' \quad \Gamma'(\alpha) = b \quad R \in \{=, \ne\} \\ \tau = \begin{cases} \mathsf{nonnull}\ b & \text{if } \neg(\alpha\ R\ 0) \equiv \alpha \ne 0 \\ \mathsf{null} & \text{if } \neg(\alpha\ R\ 0) \equiv \alpha = 0 \end{cases} \end{array}}{\texttt{branch}\ ae\ L\ :\ \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle_p \rightarrowtriangle \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \tau] \fatsemi F' \rangle_{p+1}} \text{ c-nullchk}_F \qquad \frac{\begin{array}{l} \Gamma \vdash \Sigma(ae) \Downarrow \alpha\ R\ 0 \rhd \Gamma' \quad \Gamma'(\alpha) = b \quad R \in \{=, \ne\} \\ \tau = \begin{cases} \mathsf{nonnull}\ b & \text{if } \alpha\ R\ 0 \equiv \alpha \ne 0 \\ \mathsf{null} & \text{if } \alpha\ R\ 0 \equiv \alpha = 0 \end{cases} \end{array}}{\texttt{branch}\ ae\ L\ :\ \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \rightarrowtriangle \langle \Sigma \fatsemi \Gamma'[\alpha \mapsto \tau] \fatsemi F' \rangle_L} \text{ c-nullchk}_T$$

$$\frac{I_{\mathrm{c}} \colon F \rightarrowtriangle_{\mathrm{f}} F'}{I_{\mathrm{c}} \colon \langle \Sigma \fatsemi \Gamma \fatsemi F \rangle \rightarrowtriangle \langle \Sigma \fatsemi \Gamma \fatsemi F' \rangle} \text{ c-follow}$$

A.3.3 Typing

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

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

$$\frac{\begin{array}{l} \Gamma \vdash e \Downarrow \alpha + n \rhd \Gamma' \qquad \Gamma' \vdash \alpha : \mathsf{nonnull}\ C \rhd \Gamma'' \\ T(C) = \texttt{class}\ C\ \ldots \{\ \ldots\ \tau\ f\ \ldots\ \} \qquad (f \text{ is the field at offset } n \text{ of class } C) \end{array}}{\Gamma \vdash e : \tau\ \mathsf{ptr} \rhd \Gamma''} \text{ fieldptr}$$

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

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

$$\frac{\Gamma \vdash e \Downarrow \&\texttt{init\_table} + 4 \cdot \beta \triangleright \Gamma' \quad \Gamma' \vdash \beta : \mathsf{tag}(\alpha, N) \triangleright \Gamma''}{\Gamma \vdash e : \mathsf{nonnull\ classof}(\alpha)\ \mathsf{ptr} \triangleright \Gamma''}\ \mathsf{protptr} \qquad \frac{\Gamma \vdash e \Downarrow \&\texttt{init\_table} + 4 \cdot \beta + 4 \triangleright \Gamma' \quad \Gamma' \vdash \beta : \mathsf{tag}(\alpha, N) \triangleright \Gamma''}{\Gamma \vdash e : \mathsf{init}(\alpha)\ \mathsf{ptr} \triangleright \Gamma''}\ \mathsf{initptr}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha \triangleright \Gamma' \quad \Gamma'(\alpha) = \mathsf{nonnull}\ C}{\Gamma \vdash e : \mathsf{tag}(\alpha, \{n \mid n = tagof(C') \wedge \Gamma' \vdash C' <: C\})\ \mathsf{ptr} \triangleright \Gamma'}\ \mathsf{tagptr} \qquad \frac{\Gamma \vdash e \Downarrow \alpha \triangleright \Gamma' \quad \Gamma'(\alpha) = \mathsf{nonnull\ exactly}\ C}{\Gamma \vdash e : \mathsf{tag}(\alpha, \{tagof(C)\})\ \mathsf{ptr} \triangleright \Gamma'}\ \mathsf{tagptrexactly}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha \triangleright \Gamma' \quad \Gamma'(\alpha) = \mathsf{nonnull\ classof}(\beta) \quad \Gamma \vdash \beta : \mathsf{tag}(\gamma, N)\ \mathsf{ptr} \triangleright \Gamma'}{\Gamma \vdash e : \mathsf{tag}(\gamma, N)\ \mathsf{ptr} \triangleright \Gamma'}\ \mathsf{tagptrclassof}$$

$$\frac{\Gamma \vdash e \Downarrow \alpha \triangleright \Gamma' \quad \Gamma' \vdash \alpha : \mathsf{excframe} \triangleright \Gamma''}{\Gamma \vdash e : \mathsf{handler\ ptr} \triangleright \Gamma''}\ \mathsf{throwptr}$$