The Coolaid Reference Manual

Bor-Yuh Evan Chang George Necula

November 10, 2004

1 Introduction

Coolaid is a tool to statically verify some basic correctness properties of the MIPS assembly code produced from Cool source. Coolaid will check that the assembly code is "well-typed" with respect to the Cool typing rules*, just like the Java bytecode verifier checks that the bytecode output by a Java compiler is type safe. Aside from checking the safety of the output code, this tool can greatly benefit the development process of a Cool compiler, specifically the code generation phase. Since the compiler front-end ensures that the source program is well-typed and that the type system guarantees certain safety properties, we expect that those properties should also hold on the resulting assembly code. If Coolaid is not able to verify some particular safety property, then a likely cause is a bug in the compiler itself.

Traditionally, one debugs the code generation phase of a compiler by either

1. compiling test programs, executing them on sample inputs, and then looking for the expected behavior of the test program; or

2. inspecting the assembly produced by the compiler on test programs.

The first method suffers from the problem that not only must a suite of test programs be designed to cover the functionality of the compiler, but also sample inputs must be created to cover all the paths of the test programs. It is also very difficult to create test programs and their sample inputs to cover all possible mistakes in a compiler. Finally, a bad instruction in code generated by a compiler often manifests itself long after the instruction has executed, making it very hard to diagnose the bug.

The second method is extremely tedious, if done manually. Coolaid can be viewed as a tool to automate this step. Given a compiler test case, Coolaid will check the output of the Cool compiler without requiring that we run this generated code. Coolaid will point precisely at the offending instruction, but you still have to find out why your compiler is emitting this instruction.

At the time of this writing, Coolaid is checking for 175 different correctness conditions. We tested Coolaid on 3500 programs generated by student compilers from the Spring 2003

Last modified on Wed Nov 10 04:34:45 PST 2004

*Actually, the analysis performed by Coolaid is somewhat stronger in that there are programs Coolaid can verify as safe that would not pass the Cool type-checker.

offering of CS164 at UC Berkeley. The standard testing procedure used for grading found errors in 1000 of those. Often the error messages were in the form of garbled output or output that did not match the expected output.

Coolaid finds errors in 800 of the 2500 programs that pass the testing procedure. These were errors that the testing procedure missed because the sample input for the code did not exercise the bad code portions. However, Coolaid failed to detect compilation errors in 50 of the 1000 programs that failed the testing procedure. This is because Coolaid looks only for typing errors and will not catch a compilation error in which the compiler generates type safe code that does not behave as the source code.

2 Getting Started

Coolaid is distributed as an executable called `coolaid` in the `bin` class directory. One can begin by running Coolaid on code generated by a known good Cool compiler (e.g., the CS143 reference Cool compiler), as follows:

```
cp /usr/class/cs143/examples/manual-ex1.cl .
coolc manual-ex1.cl
```

You will see that code generated by the reference compiler contains a number of annotation lines (starting with `#ANN`). These lines tell Coolaid what classes are compiled in the file and with what attributes and what methods. If you use the script `mycoolc` to run your compiler, then the annotations will be generated for you.

Now you can run Coolaid on the MIPS assembly file produced either by `coolc` or by your compiler.

```
coolaid manual-ex1.s
```

Additional command-line options that you can give to the `coolaid` command are described in Appendix A.

The above command will start the main Coolaid window as shown in Figure 1. This display is much like a debugger showing the assembly code along with buttons, for example, to ▶ (Step) forward an instruction, ◀ (Step Back) an instruction, or ▶▶ (Run) until completion. However, in contrast to SPIM, Coolaid does not actually execute instructions, but rather verifies that each instruction is safe (with respect to the safety properties for which it is checking). We can more accurately describe ▶ (Step) as check the current instruction and then go to the next instruction and ▶▶ (Run) as check all the remaining instructions.

Click on the ▶▶ (Run) button to verify the entire program. Upon completion, you should get a dialog box saying that the verification succeeded.

Coolaid can be used even if you do not read the rest of this manual. It will print error messages in the console window pointing to offending instructions, and you can try to figure out what is wrong. However, if you understand a bit of how Coolaid works, you can use it as a powerful debugger for the generated code. In the rest of this manual, we explain how Coolaid works, and how to interpret the information that is shown in the user interface.

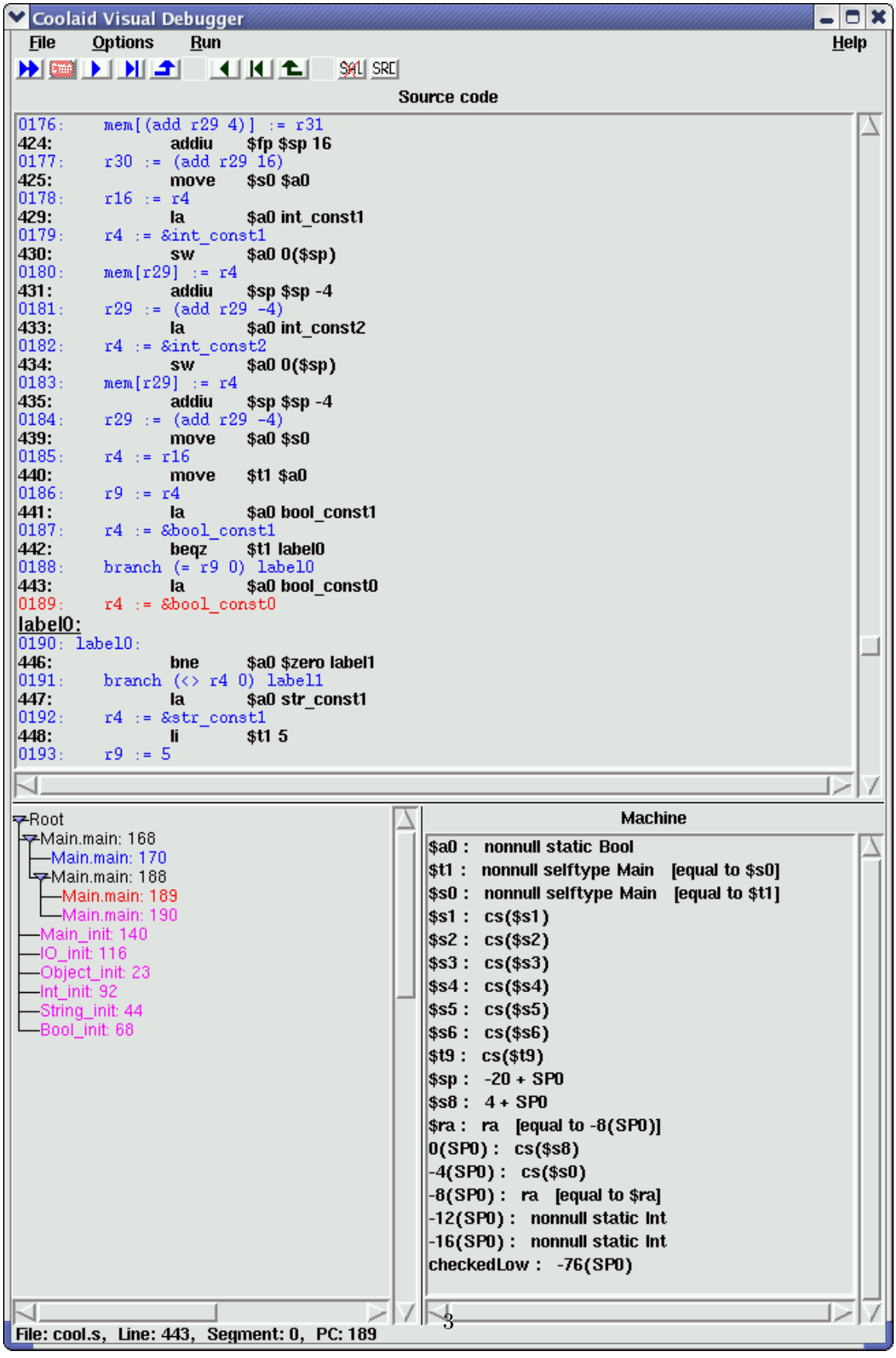

Figure 1: Coolaid.

3 The Graphical User Interface

The main Coolaid window is divided into three panes (see Figure 1). The largest pane shows the assembly code with the current instruction to be checked highlighted in red. In the lower left, a pane shows the verification path, which allows one to control the order in which instructions are checked. The pane in the lower right displays the information that we have about register contents at the current instruction, which is used to verify the safety of that instruction. Finally, the status bar at the bottom of the window indicates the assembly file being verified and the line number in the assembly file, the segment number in the assembly file, and the instruction number (PC) that Coolaid is currently checking.

3.1 Controlling the Verification

Coolaid verifies the code one method at a time and the body of the method one instruction at a time, in the execution order. The verification follows an unconditional jump, unless it is a method call. For a method call, the verification continues with the instruction after the call, once the call instruction is verified. When a conditional branch is encountered, Coolaid must verify the instruction sequences that are pointed to by each target of the branch. This can be viewed as a branching point in the verification process: first one target is explored with all instructions that are reachable from it, then the verification backtracks and checks the instructions that are reachable from the other target of the branch. The evolution of such a verification process can be depicted as a tree for each method. The root of the tree corresponds to the first instruction in the method, and the internal nodes correspond to branch instructions. The leaves of the tree are the instructions where verification stops: a return instruction or a call to one of the run-time functions that abort the execution (e.g., `dispatch_abort`). We shall see in Section 4 that there is one more case when verification stops. While the verification is in progress, the tree has been only partially explored.

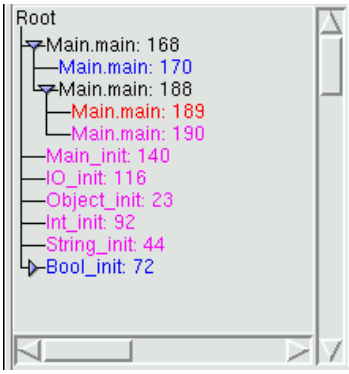

Figure 2: Verification Path.

The tree displayed in the pane in the lower left evolves as verification proceeds with the current leaves showing the instructions to be verified next (see Figure 2). Upon loading, this pane lists the labels at the start of each method in the program as initial roots from which to start verification. Each node represents an instruction, which is displayed as the pair method name and assembly line number (e.g., Main.main:168). The start of paths to be verified are shown in magenta, while the current instruction to be checked is highlighted in red. At any time, one can switch to verifying a different path by clicking on a magenta node. Blue nodes indicate entire subtrees that have already been verified. Branches in the tree arise from verifying branches in the assembly code. In the example in Figure 2, we are currently checking line 189 in `Main.main` with the subtree starting from line 170 in `Main.main` and the entire `Bool_init` method completely verified; all other methods and the rest of `Main.main` have not yet been verified.

The verification path is synchronized with the main code window, which also highlights the current instruction in red. One can start trying to verify as much as possible by clicking the ▶▶ (Run) button (and can then use the STOP (Stop) button to again halt the verification). Alternatively, one can step through the verification instruction-by-instruction using the ▶ (Step)/◀ (Step Back) buttons. One can also set breakpoints or run to a specific instruction by right-clicking in the code window at the desired instruction and clicking the appropriate menu-item (see Figure 3).

Two useful features are to step forward until we finish verifying the current subtree. Similarly, you can step backward until the current instruction is the parent of the current subtree.

In order to support ◀ (Step Back), the GUI makes periodic complete snapshots of the verification state. One can speed the verification by controlling how often these are made (see Appendix A for details).

Using the File → Reload Source one can tell Coolaid to reparse the MIPS file and start from the beginning. This is useful if changes have been made to the file. Or, one can simply restart the verification using File → Restart. In this case, the breakpoints are preserved.

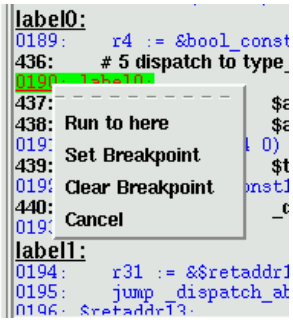

Figure 3: Breakpoints.

The layout of objects and dispatch tables that Coolaid has inferred from your assembly file can be shown, by using Options → Cool → Show Object Layout. This will add some extra entries to the type state display. One must double click on these entries to get them to show in separate windows.

Coolaid actually performs verification on a generic assembly language called SAL. On initialization, MIPS assembly is translated into SAL for verification. For the most part, one MIPS instruction corresponds to one SAL instruction, but there are a few cases where one MIPS instruction is translated into several SAL instructions (e.g., `jal`). Although you should not need to, you can toggle the display of SAL instructions in the code window by selecting Options→GUI→Show SAL or by passing the argument `-showSal` to Coolaid. See Appendix B for more details on SAL.

Similarly, Coolaid can show the lines of Cool source code to which certain assembly language blocks belong. This is possible if the Cool compiler places line number annotatation in the output assembly file. The reference Cool compiler does so with the `-L` command-line option. If there are line annotasions in the file then you will see an option Options→GUI→Show Source that you can use to toggle the display of source code.

3.2 The Type State Display

The pane in the lower right shows the current information about registers and stack slots. This information can be used to help determine why Coolaid was not able to verify some piece code. For each register or stack slot, Coolaid shows one or more types (such as that register `$a0` contains a non-null address of an object of static type `IO`). Coolaid uses numeric register names ($\mathbf{r}_0$ to $\mathbf{r}_{31}$), but one can turn on the use of symbolic names (e.g., `$a0`) with Options → Arch → Numeric Register Names.

The stack slots are represented by (possibly negative) offsets from the value of the stack pointer *method entry*. For example, 4(SP0), means the stack slot at offset $sp + 4$ on function entry. Registers or stack slots for which there is no interesting type information are not shown.

The types used by Coolaid are an extension of, but are necessarily more complicated than, the types used in the Cool source language; to understand these types one ought to read Section 4 and specifically, Section 5 for a complete description. Coolaid also keeps track of equalities between registers and display those in the type state.

Note that a register or a stack slot may have more than one type associated with it. In that case, the various types are separated by commas.

4 Verification Procedure

Coolaid starts by reading the annotations present in the assembly file. There must be one annotation for each class, declaring the parent class and the new attributes. Also, there must be annotations for each method of each class, with a list of argument types and a result type. Coolaid requires that objects be laid out in a specific way: first the attributes of the parent class, followed by the new attributes in the same order as declared in the class definition.

Coolaid then verifies the presence of the required labels (see "The Cool Runtime System"). Coolaid uses the table `class_objTab` to find the tags for each class (based on the order they appear in that table), along with the prototype objects and the initialization methods. From the prototype objects, Coolaid finds what the dispatch table is for each class. Once Coolaid has verified this information, it shows the user interface and waits for user input to start verifying the instructions you have generated. If the `-batch` option was passed to `coolaid` the verification proceeds without showing the user interface.

Coolaid verifies assembly code using a technique known as *abstract interpretation*, which is similar to how an interpreter would execute the code. The difference is that Coolaid does not maintain concrete values for registers, but maintains instead only partial information. For example, Coolaid might only record that at a certain point the register **r** holds a reference to an object of type `IO`. This is all the information it needs in order to type check that the value in register **r** is used correctly. The major advantage of an abstract interpreter over an standard interpreter is that the abstract one can interpret the program even in the absence of input data and also that it can do the verification in a finite amount of time, even when the standard interpreter runs forever. The catch is that while the standard interpreter computes the actual result of the program, the abstract one computes only whether or not the program is well-typed.

4.1 Example

To demonstrate the verification procedure, consider the following Cool program (also present in the class directory `examples/manual-ex1.cl`).

```
class Parent {                          -- A "sequence" class
  next() : Parent { ... };                      -- Iterator method
};
class Child inherits Parent {
};
class Main {
  scan(y : Child) : Object {                      -- Scan the sequence
    let x : Parent <- y in
    while not(isvoid x) loop
      x <- x.next()
    pool
  };
};
```

In the following, we show a compilation (with some optimization) of the `Main.scan` method into SAL, eliding the function prologue and epilogue. (This code is not identical to that generated by `coolc`, but a version that follows more closely is available in the class directory at `example/manual-ex2.s` so that you can run Coolaid step by step.) We show the abstract state that is computed at each program point right-justified and boxed. Note that $\mathrm{L}_{\mathtt{dispatch\_abort}}$ labels some code to issue an error message and to abort because of the attempt to dispatch on a *void* value, and $\mathrm{L}_{\mathrm{done}}$ labels the function epilogue (neither are shown). The notation $\mathbf{r}_x$ denotes a register name. For clarity, we have used subscripts on the register names according to the source variable to which they correspond (e.g., $\mathbf{r}_x$ corresponds to `x`) or to which role they play (e.g., $\mathbf{r}_{ra}$ holds the return address and $\mathbf{r}_{rv}$ holds the return value of a just-returned method).

The instructions in lines 4–10 implement the method dispatch `x.next`, consisting of a null check (line 4), fetching of the pointer to the dispatch table (line 5), fetching of the pointer to the method (line 6), setting the `self` argument (passed in register $\mathbf{r}_{arg_0}$) and the return address (lines 7–8), and finally the indirect jump in line 9. This particular compilation assumes that the pointer to the dispatch table is at offset 8 in an object and that the pointer to method `next` is at offset 12 in the tables for classes *Parent* and *Child*.

Coolaid starts with the assumption that $\mathbf{r}_y$ has type *Child*, given by the signature of method `Main.scan`. After it sees the assignment in line 2, the abstract state reflects that $\mathbf{r}_x$ also has type *Child*. When Coolaid encounters the conditional in line 4, it continues the verification with the true branch, followed by the verification of the false branch, once all the instructions reachable from the true branch have been verified. The verification of the true branch proceeds with label $\mathrm{L}_{\mathtt{dispatch\_abort}}$, and presumably goes on until it encounters a call to the `dispatch_abort` function. At that point, Coolaid backtracks and continues the verification with line 5.

In the false branch of line 4, Coolaid recognizes the preceding conditional as a void-check and can conclude that $\mathbf{r}_x$ is non-void. Then, in line 5, it determines that reading from offset 8 into an object $\mathbf{r}_x$ yields the dispatch table of that object and records this as $\mathbf{r}_t : \mathsf{dispatch}(\mathbf{r}_x)$. Note that Coolaid says this instruction is safe only because $\mathbf{r}_x$ contains a value that is non-void and points to an object. Then, the verifier recognizes that we are fetching in $\mathbf{r}_t$ the pointer to the method at offset 4 in the dispatch table of $\mathbf{r}_x$, as encoded

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

                    ⋮
```

by $\mathbf{r}_t$: method($\mathbf{r}_x$, 12). Again, this is safe only because before this instruction $\mathbf{r}_t$ points to a dispatch table of $\mathbf{r}_x$ with a method at offset 12. In the next two lines, Coolaid not only updates the types of registers, but also remembers equalities between the abstract values in the registers. All of these steps collect as part of the abstract state enough information that the indirect jump instruction on line 9 can be verified, as follows. Since $\mathbf{r}_t$: method($\mathbf{r}_x$, 12) and $\mathbf{r}_x$: *Child*, the verifier can consult the class hierarchy accompanying the compiled code to find that a method `next` is being called. Since $\mathbf{r}_{arg_0} = \mathbf{r}_x$, we can check that the `self` argument is equal to the object that was used to resolve the method. Additionally, the verifier must check that the return address is correctly set and then continue the verification of the code.

After the method dispatch at line 9, the return value (assumed to be in $\mathbf{r}_{rv}$) has type *Parent*. Say that Coolaid verifies first the true branch of the conditional in line 11. It proceeds to label $\mathrm{L}_{\texttt{done}}$, until presumably it encounters the return instruction, at which point it will be able to verify that the value in $\mathbf{r}_{rv}$ has type *Parent*, hence also type `Object`, as required by the method signature. At that point Coolaid backtracks and continues the verification with the false branch of conditional in line 11, with the assumption that $\mathbf{r}_{rv}$ is not void.

After the assignment in line 12, the abstract state of $\mathbf{r}_x$ changes to *Parent*, and Coolaid follows the jump to reach, again, the start of the loop in line 3. Coolaid realizes that it has inspected this code before, with the assumption that both $\mathbf{r}_x$ and $\mathbf{r}_y$ have type *Child*. This old assumption does not hold anymore, when line 3 is reached from line 13, since now nothing is known about register $\mathbf{r}_y$ and $\mathbf{r}_x$ has type *Parent*, which is a weaker assumption than $\mathbf{r}_x$ having type *Child*. At this point Coolaid, like any abstract interpreter, computes the least upper-bound of the types for each register (just like in the Cool typing rules at the end of a conditional). This operation is sometimes referred to as computing the join of the abstract states after lines 2 and 12. In this example, Coolaid concludes that $\mathbf{r}_x$: *Parent* after line 3 (not shown). Since the assumptions have been weakened, Coolaid cannot be sure that the previous verification still holds. Thus, it will make another pass over the code, with weaker assumptions (not shown). At the end of the second pass, Coolaid will realize that the jump back to the start of the loop does not change the assumptions after line 3. In technical terminology, we say that Coolaid has reached a fixed point, and has finished verifying this method.

A very instructive exercise is to modify manually the `manual-ex2.s` code to see how Coolaid complains. Try, for example, to "forget" the assignment in line 12.

5 The Types Used by Coolaid

In this section we discuss the types that Coolaid uses to characterize the values contained in register values and stack slots. Many of these types refer to register or stack slots names. We use the r to denote such a name. We also use n and k to denote integer constants. We use the letter τ to refer to such a type.

- unknown: the type given to registers and stack slots whose contents are undefined, perhaps because they were not initialized or because they were modified in unpredictable ways by a method call. At method entry, most registers have this type. Note

that registers with unknown type are not shown in the type-state pane in the GUI.

- *Qualifier* C*: the type of possibly-null addresses of Cool objects with static type C (a Cool class). Zero or more qualifiers can be present. The following qualifiers may appear:
  - nonnull: means that the value is not zero
  - static: means that the value is the address of a statically allocated object. For example, `& Int_protObj` has type "nonnull static `Int`".
  - selftype: means that the value has the same dynamic type as the self object for the method being verified. Initially, the first argument to a method of class C is assumed to have type "nonnull selftype C", assuming that the `self` argument is passed in register `$a0`.
  - like **(r)**: means that the value has the same dynamic type as the object whose address is stored in register or stack slot r. This is useful for ensuring that the `self` argument passed to a method has the same dynamic type as the object from which the method entry point was fetched.
- word: the type of register and stack slots that are initialized and contain an arbitrary value. The only way such a type can arise is by reading the contents of attributes in the basic classes `Int` and `Bool`.
- dispatch **(r)**: the type of the address of the dispatch table fetched from the object stored in register or stack slot r. Such a value is obtained by reading from offset 8 (according to Cool's object layout) from an object r with type nonnull C (or with additional qualifiers).
- method **(r, n)**: the type of the address of the first instruction in a method that was obtained by reading from offset n from a value of type dispatch(r) (i.e., the dispatch table of the object stored in r).
- sdispatch **(C)**: the type of the address of the dispatch table of class C. Coolaid uses the label (e.g., C`_dispTab`) specified in the prototype object to determine the label of the dispatch table for each class.
- smethod **(C, n)**: the type of the address of the first instruction in the method mentioned at offset n in the dispatch table of class C. Such a value can be obtained by reading at offset n from a value of type sdispatch(C). Additionally, smethod(C, n) is also the type of the address of the label for the nth method of class C.
- classobj: the type of the address of the `class_objTab` table.
- imethod **(C)**: the type of the address of the first instruction in the initialization method for class C. Coolaid finds out which are the initialization methods by reading the `class_objTab` table.
- tag(r, $\vec{n}$): the type of the tag word loaded from the object stored in r. At the same time, these tag values are known to be a member of the list of integers $\vec{n}$. A value of

this type can be obtained by reading the first word of the object stored in r of type nonnull C. At the time of the read, the list $\vec{n}$ is constructed to contain the tag of C and of all its subclasses based on the class hierarchy.

Values of this type might be used in equality and inequality comparisons with integer constants. Following such comparisons, Coolaid refines the list $\vec{n}$ appropriately. At the same time, Coolaid refines the type C of object r based on the current least upper-bound of the classes whose tags are still in the list $\vec{n}$. This is how Coolaid is able to handle the compilation of the Cool `case` expression.

- $\underline{n}$: the type of values equal to the constant n. For example, if a constant n is loaded into a register, then that register has type n. Coolaid will also handle arithmetic between such values correctly.

- $\underline{n + k_1 \cdot \tau_1 + \cdots + k_i \cdot \tau_i}$: the type of values that are obtained by adding the integer constant n to k_1 values of type τ_1 to k_2 values of type τ_2, and so on. We call this an arithmetic type. This type is abbreviated as $n + \tau$, when the constant $k = i = 1$. In such a type, not all possible choices for τ are valid. For example, τ cannot be a constant type n', or another arithmetic type, because in such cases the type can be simplified. Coolaid will attempt to track arithmetic operations involving such types. We list below a few common uses for the arithmetic type.

  - Offsets from object addresses are useful for accessing the tag, dispatch table, or the attributes. For example, a value of type "$8 +$ (nonnull like(r) C)" is the address where the address of a dispatch table is stored. For offsets higher than 8, we obtain addresses where the attributes are stored. From this address we read a value of type dispatch(r).
  - Similarly for dispatch tables, a value of the type "$n +$ dispatch(r)" can be dereferenced to obtain a value of type method(r, n).
  - A value of type "$n +$ classobj" can be dereferenced to read a value of type imethod(C) or nonnull static C, depending whether n refers to the initializer method or to the prototype object of class C.
  - A value of type "$4+8\cdot$tag(r, $\vec{n}$)" can be used as an index into the `class_objTab` to obtain the initializer method for objects of the same dynamic type as the object stored in r.
  - A value of type "$4 + 8 \cdot$ tag(r, $\vec{n}$) + classobj" is the address that contains the initializer method for objects of the same dynamic type as r.

- $\underline{\&\ L}$: the type of the address of the label L. This type is used only for labels that do not have a special meaning (prototype objects, dispatch tables, method entry points). Coolaid uses this label to check that the return address register is set to the next instruction after a call.

- $\underline{\texttt{SP0}}$: the type of the value of the stack pointer register on entry to the current method.

- cs (r): the type of the value that was stored in the callee-saved register r at the time the current method was called. Coolaid keeps track of these values because they must be placed back in their corresponding registers before the method returns.
- ra: the type of the value that was stored in the return address register at the time the current method was called. Coolaid keeps track of this value because it must be the one used for returning from the current method.

6 Conclusion

We have advocated Coolaid as a tool that can greatly benefit the development and debugging process of a Cool compiler by checking for certain safety properties in the emitted code. From our experience, we believe that Coolaid can both ease the debugging effort and in the end, yield better compilers. However, it is important to note that Coolaid is not a magic oracle for compiler correctness. If Coolaid succeeds, the generated code is type safe, but it does not guarantee that the generated code behaves exactly as the source code dictates according to the operational semantics for Cool. Conversely, if Coolaid fails for code generated by a Cool compiler, it almost always indicates a bug in the compiler, but in extremely rare cases, the compiler may be doing something so clever that Coolaid does not understand why it is safe. Think very carefully before deciding that this is the case for your compiler.

Acknowledgments. We would like to thank Robert Schneck-McConnell and Kun Gao for their efforts on the implementation of Coolaid, experimentation with early versions, and feedback on the documentation. Also, we thank Jeremy Condit and Sumit Gulwani for test driving Coolaid and for providing insightful comments. Finally, we acknowledge Matt Harren and Wes Weimer for useful suggestions on the documentation for Coolaid.

A Command-Line Parameters and Menu Options

In general, Coolaid is invoked from the command-line as follows:

```
coolaid [options] files
```

and takes the following command-line parameters. Some command-line parameters can also be toggled in the GUI from the menu bar at any time.

command-line	menu	description
`-help`		Displays the command-line parameters. Among the many parameters that are printed, you should only need the ones described below.
`-batch`		Verify the program in batch mode instead of using the GUI. This is much faster than using the GUI and useful if you want to run Coolaid in a script. An exit code of zero indicates success.
`-verbosecool`	Options → Cool → Verbose	Print additional debugging information in the terminal window. Use this if Coolaid reports an error before starting the user interface, or if you want additional information. In this release, this information is not optimized for readability.
`-keep-going`		Do not stop on the first error. Coolaid will attempt to continue the verification from the next method.
`-updateInterval=nn`	Options → Cool → Update Interval (ms)	Update the user interface every `nn` milliseconds. A larger value (e.g., 500) makes the verification faster but will make the display choppy while the verification is in progress. Has no effect for the batch mode.
`-snapshotInterval=nn`	Options → GUI → Save State Interval	Save a complete snapshot of the state every `nn` verification steps. A larger value (e.g. 1000) makes the verification faster but will slow down the stepping back feature. Has no effect for the batch mode.
`-showSal`	Options → GUI → Show SAL	Show SAL instructions interleaved with the MIPS instructions. See Appendix B for details on SAL.
`-showSource`	Options → GUI → Show Source	Show Cool source interleaved with the MIPS instructions. The Cool compiler must include line number information (`-L` command-line option).
`-cool_exc/` `-no-cool_exc`	Options → Cool → Check Exceptions	Turn on/off the verification of Cool exceptions.

B SAL: Simple Assembly Language

Coolaid is implemented on top of the Open Verifier infrastructure, developed at UC Berkeley. This infrastructure translates MIPS or Intel x86 assembly files into a generic assembly language, called SAL. The actual verification is performed on the SAL version of the input. If you want to see how MIPS instructions have been translated into SAL, you can pass the `-showSal` option to `coolaid`, or you can turn on/off the display of the SAL instructions using the Options menu in the GUI.

We describe below the syntax of the SAL language:

instructions	*Inst*	::=	*Label* :	a label
		|	*Reg* := *Exp*	an assignment to a register
		|	*Reg* := **mem**[*Exp*]	a memory read from address *Exp*
		|	**mem**[*Exp*] := *Exp*	a memory write
		|	**jump** *Label*	a jump to the given label
		|	**jump** [*Exp*]	a indirect jump to the given address
		|	**branch** *Exp ntLabel*	a branch if expression is not zero
registers	*Reg*	::=	**r1** | $\cdots$ | **rn**	machine registers
expressions	*Exp*	::=	**n**	integer constants
		|	*Reg*	machine registers
		|	& *Label*	address of a label
		|	(*Op Exp Exp*)	binary operations
operators	*Op*	::=	**add** | **sub** | **sll** | = | <> | $\cdots$	

The set of operators in SAL correspond closely to those in MIPS or Intel x86. Among the operators are a suite of binary operators of the form `seteq`, `setle`, ..., whose result is 1 if the the first operand is equal (or less or equal) to the second, and 0 otherwise.

For example, the MIPS instruction `jal foo` is translated into SAL as

```
        r31 := & retaddr_324
        jump foo
retaddr_324:
```

C Formalization of the Verification Procedure

C.1 Judgments

The main judgment that defines the verification procedure for Coolaid is as follows:

$$T; V \vdash^{\mathcal{S}}_{P;H;\sigma_{ret}} n \# i \text{ ok} ,$$

which says that under the assumption that the type state T and abstract value state V, instruction i (with number n) in method with return type σ_{ret} of program P with class hierarchy H and state $\mathcal{S}$ is safe. We consider an instruction i safe if and only if instruction i only accesses valid memory addresses and successor instructions of i are also safe. Program P is a mapping from instruction numbers to the corresponding instructions in the assembly code. Thus, it should be the case that $P(n) = i$. Hierarchy H is a mapping from Cool class names to class information (e.g., the parent class, method types, etc.). P and H are fixed per program and σ_{ret} is fixed per method, so we will often elide them. Type state T provides a mapping from registers to types (as shown in the Machine Registers pane in the GUI). Similarly, V provides a mapping from registers to abstract values, which are indirectly displayed in the Machine Registers pane with the equalities they imply. States $\mathcal{S}$ is the type/value state that has been computed at each program point, i.e., a mapping from instruction numbers to type/value state pairs. One can think of $\mathcal{S}$ as containing the pair $\langle T, V \rangle$ after instruction i has been checked to only access valid memory addresses (i.e., i is "locally safe"). We will also often elide references to $\mathcal{S}$ since changes to it are predictable.

We also define a typing judgment for SAL expressions and a subtyping judgment in order to check that an instruction is locally safe (i.e., only accesses valid memory addresses).

$\sigma_1 \leq_H \sigma_2$	in class hierarchy H, σ_1 is a subtype of σ_2
$T; V \vdash_H e : \sigma$	in class hierarchy H, under the type state T and value state V, expression e has type σ

Like for the verification judgment, we will often elide the H since it is fixed per program.

C.2 Rules

This section defines the inference rules for deriving the judgments in the previous section, which define the behavior of Coolaid. Understanding what Coolaid expects at each instruction will help diagnose why generated code cannot be verified and thus find a potential bug in the Cool compiler.

C.2.1 Verification Judgment

We defer the description of typing and subtyping judgments until Section C.2.2 and focus on the main verification procedure, as this is perhaps the most relevant to finding potential bugs in the Cool compiler.

Read. Recall that the basic invariant we maintain is the following:

> **Invariant 1** *An address e is safe to access if and only if e has type τ* ptr *for some address type τ.*

This translates naturally to the following rule:

$$\frac{T;V \vdash e : \tau \text{ ptr} \quad T[\tau/\mathbf{r}_k];V[v/\mathbf{r}_k] \vdash n{+}1 \# P(n{+}1) \text{ ok} \quad (v \text{ fresh})}{T;V \vdash n \# \ \mathbf{r}_k \texttt{ := mem[}e\texttt{]} \ \text{ok}} \ \textsf{read}$$

which says that first check that e is a pointer to some type τ, then we check the next instruction with the assumption register $\mathbf{r}_k$ has type τ and $\mathbf{r}_k$ corresponds to a fresh abstract value. Expression e has a pointer type only if it is an offset into an object or table. For example, consider the following class `Example`:

```
class Example {
  x : Int <- 0;
};
```

Also, suppose that register $\mathbf{r}_1$: nonnull `Example`, i.e., $\mathbf{r}_1$ contains a (non-null) reference to an object of type `Example`. According to the class hierarchy and the Cool object layout [], offset 12 in an object of type `Example` is a reference to an object of type `Int`. Thus, the expression $\mathbf{r}_1 + 12$ would have type `Int` ptr. Typing of expressions is discussed in further detail in Section C.2.2.

One detail that is not mentioned above is that a memory read from address on the stack is treated like the reading of a register (discussed below). Coolaid is able distinguish which memory reads are in the stack and which are in the heap (or in the static data segment).

Write. Similar to memory reads, we must check that writes are only addresses that can be typed as pointers. However, an additional requirement is that we must check that what is written to memory conforms to the pointer type (this is similar to assignment case for the Cool type system).

$$\frac{T;V \vdash e_1 : \tau \text{ ptr} \quad T;V \vdash e_2 : \sigma \quad \tau \text{ writeable} \quad \sigma \leq \tau \quad T;V \vdash n{+}1 \# P(n{+}1) \text{ ok}}{T;V \vdash n \# \ \texttt{mem[}e_1\texttt{] := } e_2 \ \text{ok}} \ \textsf{write}$$

We also impose an additional requirement that τ be writable (i.e., not read-only) using the auxiliary judgment τ writeable. This judgment τ writeable is defined to exclude the read-only address types (ro word, ro C, and nonnull ro C); we elide the obvious inference rules for this judgment. Finally, notice that after checking that this memory write is safe, we proceed to check the next instruction with the same type/value state. Just as for reads, writes to the stack are treated as register updates (discussed below).

Set. For a register update, we consider two cases. The first case is when we copy the contents of one register to another.

$$\frac{T[T(\mathbf{r}_{k'})/\mathbf{r}_k];V[V(\mathbf{r}_{k'})/\mathbf{r}_k] \vdash n{+}1 \# P(n{+}1) \text{ ok}}{T;V \vdash n \# \ \mathbf{r}_k \texttt{ := } \mathbf{r}_{k'} \ \text{ok}} \ \textsf{move}$$

In this case, we simply update the type state T to map $\mathbf{r}_k$ to the type of $\mathbf{r}_{k'}$ and value state V to map $\mathbf{r}_k$ to the value of $\mathbf{r}_{k'}$ to check the next instruction. This is where equalities between values in registers are introduced.

The second case is when the register is updated to an arbitrary expression.

$$\frac{T; V \vdash e : \sigma \quad T[\sigma/\mathbf{r}_k]; V[v/\mathbf{r}_k] \vdash n{+}1 \,\#\, P(n{+}1)\ \mathsf{ok} \quad (v \text{ fresh}) \quad (e \neq \mathbf{r}_{k'})}{T; V \vdash n \,\#\ \ \mathbf{r}_k\ \texttt{:=}\ e\ \ \mathsf{ok}}\ \mathsf{set}$$

In this case, we compute the type of e and update T with that type for $\mathbf{r}_k$ and choose a fresh abstract value for $\mathbf{r}_k$ in V.

Branch. For branches, in general, we simply need verify the two possible next instructions. However, we may need to refine the type state to reflect new knowledge about a register. The first case is if the branch is in fact a null-check.

$$\frac{T; V \vdash \mathbf{r}_k : \gamma \quad T[\mathsf{nonnull}\ \gamma/\mathbf{r}_k]; V \vdash l \,\#\, P(l)\ \mathsf{ok} \quad T; V \vdash n{+}1 \,\#\, P(n{+}1)\ \mathsf{ok}}{T; V \vdash n \,\#\ \ \texttt{branch}\ (\texttt{<>}\ \mathbf{r}_k\ 0)\ l\ \ \mathsf{ok}}\ \mathsf{nullcheckne}_L$$

where we let γ stand schematically for either a class (C or $\mathsf{ro}\ C$) or self type $\mathsf{selftype}\ C$. Here we check both branches but refine the type of $\mathbf{r}_k$ in the true branch to indicate that it is non-null. Coolaid also recognizes a null-check for the symmetric disequality ($\texttt{<>}\ 0\ \mathbf{r}_k$) and equality with 0. We elide these rules but note that in the latter the false branch would get the nonnull assumption instead. Note that Coolaid only recognizes a null-check if the comparison is with the integer constant 0 (not an expression that evaluates to 0).

The second case is if the branch compares the class tag of some object and enables Coolaid to determine that the dynamic type of some object is more specific (i.e., a subtype) of the assumption currently in the type state. This is necessary to be able to verify the `case` construct in Cool.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{tag}(v,\ \vec{m}) \\ T; V \vdash \mathbf{r}_{k'} : \mathsf{nonnull}\ C \\ T[\mathsf{tag}(v,\ \vec{m_t})/\mathbf{r}_k][\mathsf{nonnull}\ \mathit{taglub}(H, \vec{m_t})/\mathbf{r}_{k'}]; V \vdash l \,\#\, P(l)\ \mathsf{ok} \\ T[\mathsf{tag}(v,\ \vec{m_f})/\mathbf{r}_k][\mathsf{nonnull}\ \mathit{taglub}(H, \vec{m_f})/\mathbf{r}_{k'}]; V \vdash n{+}1 \,\#\, P(n{+}1)\ \mathsf{ok} \\ (\vec{m_t} = \langle m_j \mid m_j < m' \rangle) \\ (\vec{m_f} = \langle m_j \mid m_j \geq m' \rangle) \\ (V(\mathbf{r}_{k'}) = v) \end{array}}{T; V \vdash n \,\#\ \ \texttt{branch}\ (\texttt{<}\ \mathbf{r}_k\ m')\ l\ \ \mathsf{ok}}\ \mathsf{classrefinelt}_L$$

where $\mathit{taglub}(\vec{m})$ is the least-upper bound (in the Cool class hierarchy H) of the classes given by the list of class tags $\vec{m}$. Also, note that $\langle m_j \mid m_j < m' \rangle$ is the list of entries of $\vec{m}$ that are less than m'. By comparing a register $\mathbf{r}_k$ which contains the class tag for an object with reference v, we constrain what the dynamic type could be in the appropriate branch.

Similarly, there is a rule if $\mathbf{r}_{k'}$: nonnull selftype C.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{tag}(v,\ \vec{m}) \\ T; V \vdash \mathbf{r}_{k'} : \text{nonnull selftype } C \\ T[\mathsf{tag}(v,\ \vec{m_t})/\mathbf{r}_k][\text{nonnull selftype } taglub(H, \vec{m_t})/\mathbf{r}_{k'}]; V \vdash l \ \#\ P(l)\ \mathsf{ok} \\ T[\mathsf{tag}(v,\ \vec{m_f})/\mathbf{r}_k][\text{nonnull selftype } taglub(H, \vec{m_f})/\mathbf{r}_{k'}]; V \vdash n{+}1 \ \#\ P(n{+}1)\ \mathsf{ok} \\ (\vec{m_t} = \langle m_j \mid m_j < m' \rangle) \\ (\vec{m_f} = \langle m_j \mid m_j \geq m' \rangle) \\ (V(\mathbf{r}_{k'}) = v) \end{array}}{T; V \vdash n \ \#\ \texttt{branch}\ (\texttt{<}\ \mathbf{r}_k\ m')\ l\ \ \mathsf{ok}}\ \mathsf{selftyperefinelt}_L$$

There are also appropriate rules for different comparison operators in each case, which we elide.

In any other case, the verification just proceeds on both branches with no change in the state.

$$\frac{T; V \vdash l \ \#\ P(l)\ \mathsf{ok} \quad T; V \vdash n{+}1 \ \#\ P(n{+}1)\ \mathsf{ok}}{T; V \vdash n \ \#\ \texttt{branch}\ e\ l\ \ \mathsf{ok}}\ \mathsf{branch}$$

One detail that is not mentioned above is that Coolaid will try to do some evaluation of the branch condition. If it is able to determine statically that a particular branch cannot be taken, then it will not verify that path.

Jump. A jump is used to implement both function calls and control-flow within a method. For functions, Coolaid does not follow the jump to another function while verifying that method, for it checks each function/method independently. What it must do is ensure that arguments to the function have the right types and then can proceed to the next instruction after the call assuming the return value is of the return type specified by the function. In this discussion, we will use $\mathbf{r}_{arg_0}, \mathbf{r}_{arg_1}, \ldots, \mathbf{r}_{arg_n}$ for the registers where arguments are placed and $\mathbf{r}_{rv}$ for where the return value is placed. According to the Cool calling convention, $\mathbf{r}_{arg_0}$ and $\mathbf{r}_{rv}$ correspond to `$a0`, while $\mathbf{r}_{arg_1}, \mathbf{r}_{arg_2}, \ldots, \mathbf{r}_{arg_n}$ are on the stack (see the *Tour of Cool Support Code*).

Coolaid only allows function calls for the initialization method of a class and to the runtime functions; all other calls are done through dispatch (see **Indirect Jump** below). For initialization methods, we check that that register $\mathbf{r}_{arg_0}$ is non-null and has the proper class type. We also check that the return address register has the address of the next instruction.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_{arg_0} : \sigma \\ \sigma \leq \text{nonnull } C \\ T; V \vdash \mathbf{r}_{ra} : \&\ n{+}1 \\ cs(T)[\sigma/\mathbf{r}_{rv}]; cs(V) \vdash n{+}1 \ \#\ P(n{+}1)\ \mathsf{ok} \\ (P(l) = \ \texttt{init\_}C :) \end{array}}{T; V \vdash n \ \#\ \texttt{jump}\ l\ \ \mathsf{ok}}\ \mathsf{init}_{\mathsf{jump}}$$

where $cs(T)/cs(V)$ are the type/value state where the type/value is preserved from T/V

if the register is callee-saved; otherwise unknown/fresh, i.e.,

$$cs(T)(\mathbf{r}_k) = \begin{cases} T(\mathbf{r}_k) & \text{if } \mathbf{r}_k \text{ is callee-saved} \\ \mathsf{unknown} & \text{otherwise} \end{cases}$$

$$cs(V)(\mathbf{r}_k) = \begin{cases} V(\mathbf{r}_k) & \text{if } \mathbf{r}_k \text{ is callee-saved} \\ v & \text{otherwise, for some fresh abstract value } v \end{cases}$$

This must be done because at run-time, the callee may modify any of the registers (except the callee-saved registers). Coolaid also has similar rules for the runtime functions specified in the *Tour of Cool Support Code*).

If Coolaid does not recognize the jump as a function call, then it simply continues verifying following the target of the jump.

$$\frac{T; V \vdash l \,\#\, P(l)\ \mathsf{ok} \quad (P(l) \text{ is not the label of an init or run-time function})}{T; V \vdash n \,\#\, \texttt{jump}\ l\ \mathsf{ok}}\ \mathsf{jump}$$

Label. Jumps and branches to labels can result in loops. Recall from the example in Section 4.1 that a label may need to be checked more than once but may stop the verification once a fixpoint is reached. In the example, we described the joining of the type state but ignored the value state. We also need a notion of conformance for values as well as types; in this case, we say v_1 conforms to v_2 if $v_1 = v_2$ or v_2 is fresh. Now, we define a conformance relation between states, saying $\langle T_1, V_1\rangle \preceq_H \langle T_2, V_2\rangle$ if for each register $\mathbf{r}_k$, $T_1(\mathbf{r}_k) \leq_H T_2(\mathbf{r}_k)$ and $V_1(\mathbf{r}_k)$ conforms to $V_2(\mathbf{r}_k)$.

If we are checking a label and the current state conforms to the previous state at this label, then we have reached a fixpoint.

$$\frac{\mathcal{S}(l) \preceq_H \langle T, V\rangle}{T; V \vdash^{\mathcal{S}}_{P;H;\sigma_{ret}} n \,\#\, l :\ \mathsf{ok}}\ \mathsf{fixpoint}$$

Otherwise, we need to continue checking using the least-upper bound of the previous state and the current state.

$$\frac{T'; V' \vdash^{\mathcal{S}[\langle T', V'\rangle/l]}_{P;H;\sigma_{ret}} n{+}1 \,\#\, P(n{+}1)\ \mathsf{ok} \quad (\langle T', V'\rangle = \mathcal{S}(l) \sqcup \langle T, V\rangle)}{T; V \vdash^{\mathcal{S}}_{P;H;\sigma_{ret}} n \,\#\, l :\ \mathsf{ok}}\ \mathsf{label}$$

In these rules, we have made shown the states $\mathcal{S}$ to make explicit its use here.

Indirect Jump. Indirect jumps are used by the Cool compiler to implement both method return and dispatch. Coolaid must be able distinguish between these cases. A return is identified by an indirect jump to a register that has type ra.

$$\frac{\begin{array}{l} T; V \vdash_H \mathbf{r}_k : \mathsf{ra} \\ T; V \vdash_H \mathbf{r}_j : \mathsf{cs}(j) \quad (\text{for callee-saved registers } \mathbf{r}_j) \\ T(\mathbf{r}_{rv}) \leq_H \sigma_{ret} \end{array}}{T; V \vdash^{\mathcal{S}}_{P;H;\sigma_{ret}} n \,\#\, \texttt{jump [}\mathbf{r}_k\texttt{]}\ \mathsf{ok}}\ \mathsf{return}$$

A return is safe if we can verify that the callee-saved registers have been preserved and that the value in the return-value register is a subtype of the expected return type.

An indirect call to an initialization method is similar to a direct call to one ($\mathsf{init}_{\mathsf{jump}}$) except the call is determined by the type of the register of the jump.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{imethod}(C) \\ T; V \vdash \mathbf{r}_{arg_0} : \sigma \\ \sigma \leq \mathsf{nonnull}\ C \\ T; V \vdash \mathbf{r}_{ra} : \&\ n{+}1 \\ cs(T)[\sigma/\mathbf{r}_{rv}]; cs(V) \vdash n{+}1\ \#\ P(n{+}1)\ \mathsf{ok} \end{array}}{T; V \vdash n\ \#\ \texttt{jump [}\mathbf{r}_k\texttt{]}\ \mathsf{ok}}\ \mathsf{init}_{\mathsf{ijump}}$$

A static dispatch to a method is similar except that we must get the types of the method parameters and return type.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{smethod}(C,\ m) \\ T; V \vdash \mathbf{r}_{arg_0} : \mathsf{nonnull}\ C \\ T; V \vdash \mathbf{r}_{arg_1} : \sigma_1 \\ \quad \vdots \\ T; V \vdash \mathbf{r}_{arg_p} : \sigma_p \\ (H(C).M(m) = \langle C_1, C_2, \ldots, C_{p+1} \rangle) \\ \sigma_j \leq C_j \quad (\text{for } 1 \leq j \leq p) \\ T; V \vdash \mathbf{r}_{ra} : \&\ n{+}1 \\ cs(T)[C_{p+1}/\mathbf{r}_{rv}]; cs(V) \vdash n{+}1\ \#\ P(n{+}1)\ \mathsf{ok} \end{array}}{T; V \vdash n\ \#\ \texttt{jump [}\mathbf{r}_k\texttt{]}\ \mathsf{ok}}\ \mathsf{smethod}_{\mathsf{class}}$$

where we assume the information for class C in the hierarchy H has a component M that maps offsets to the types of the method parameters and return value. This above rule is similar to the rule for dispatch for Cool source. We also need a rule if the method returns self-type.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{smethod}(C,\ m) \\ T; V \vdash \mathbf{r}_{arg_0} : \mathsf{nonnull}\ \gamma \\ T; V \vdash \mathbf{r}_{arg_1} : \sigma_1 \\ \quad \vdots \\ T; V \vdash \mathbf{r}_{arg_p} : \sigma_p \\ (H(C).M(m) = \langle C_1, C_2, \ldots, C_p, \texttt{SELF\_TYPE} \rangle) \\ \mathsf{nonnull}\ \gamma \leq \mathsf{nonnull}\ C \\ \sigma_j \leq C_j \quad (\text{for } 1 \leq j \leq p) \\ T; V \vdash \mathbf{r}_{ra} : \&\ n{+}1 \\ cs(T)[\gamma/\mathbf{r}_{rv}]; cs(V) \vdash n{+}1\ \#\ P(n{+}1)\ \mathsf{ok} \end{array}}{T; V \vdash n\ \#\ \texttt{jump [}\mathbf{r}_k\texttt{]}\ \mathsf{ok}}\ \mathsf{smethod}_{\mathsf{selftype}}$$

Dynamic dispatch is similar to the above, but we need to look up the class from the

object on which we are dispatching.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{method}(v,\ m) \\ (V(\mathbf{r}_{arg_0}) = v) \\ T; V \vdash \mathbf{r}_{arg_0} : \mathsf{nonnull}\ C \\ T; V \vdash \mathbf{r}_{arg_1} : \sigma_1 \\ \quad \vdots \\ T; V \vdash \mathbf{r}_{arg_p} : \sigma_p \\ (H(C).M(m) = \langle C_1, C_2, \ldots, C_{p+1} \rangle) \\ \sigma_j \leq C_j \quad (\text{for } 1 \leq j \leq p) \\ T; V \vdash \mathbf{r}_{ra} : \&\ n{+}1 \\ cs(T)[C_{p+1}/\mathbf{r}_{rv}]; cs(V) \vdash n{+}1 \ \#\ P(n{+}1)\ \mathsf{ok} \end{array}}{T; V \vdash n \ \#\ \ \texttt{jump [}\mathbf{r}_k\texttt{]}\ \ \mathsf{ok}}\ \mathsf{method}_{\mathsf{class}}$$

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{method}(v,\ m) \\ (V(\mathbf{r}_{arg_0}) = v) \\ T; V \vdash \mathbf{r}_{arg_0} : \mathsf{nonnull}\ \gamma \\ T; V \vdash \mathbf{r}_{arg_1} : \sigma_1 \\ \quad \vdots \\ T; V \vdash \mathbf{r}_{arg_p} : \sigma_p \\ (H(\mathit{classof}(\gamma)).M(m) = \langle C_1, C_2, \ldots, C_p, \texttt{SELF\_TYPE} \rangle) \\ \sigma_j \leq C_j \quad (\text{for } 1 \leq j \leq p) \\ T; V \vdash \mathbf{r}_{ra} : \&\ n{+}1 \\ cs(T)[\gamma/\mathbf{r}_{rv}]; cs(V) \vdash n{+}1 \ \#\ P(n{+}1)\ \mathsf{ok} \end{array}}{T; V \vdash n \ \#\ \ \texttt{jump [}\mathbf{r}_k\texttt{]}\ \ \mathsf{ok}}\ \mathsf{method}_{\mathsf{selftype}}$$

where $\mathit{classof}(\gamma)$ gives the class of γ (recall that γ is either C, $\mathsf{ro}\ C$, $\mathsf{selftype}\ C$). An important condition that is checked that the object referenced in $\mathbf{r}_{arg_0}$ is the same as the object on which we dispatch.

C.2.2 Typing Judgment

Integers and Registers. Integers are typed as $\mathsf{constword}(n)$, which remembers the actual value. For registers, we just look up the type in the type state.

$$\frac{}{T; V \vdash n : \mathsf{constword}(n)} \qquad \frac{}{T; V \vdash \mathbf{r}_k : T(\mathbf{r}_k)}$$

Labels. There are some labels that Coolaid recognizes as special: the dispatch table of a class and the prototype object. For other labels, we just remember that it is a label.

$$\frac{}{T; V \vdash \texttt{\&}C\texttt{\_dispTab} : \mathsf{sdispatch}(C)} \qquad \frac{}{T; V \vdash \texttt{\&}C\texttt{\_protObj} : \mathsf{nonnull}\ C}$$

$$\frac{(l \neq C\texttt{\_protObj}, l \neq C\texttt{\_dispTab}, l \neq \texttt{class\_objTab})}{T; V \vdash \texttt{\&}l : \&\ l}$$

Pointers. Perhaps most important is the typing of pointers, which determines what memory reads/writes are safe. For objects, the Cool object layout determines which offsets from the object reference are valid. At offset 0, we have the class tag.

$$\frac{T; V \vdash \mathbf{r}_k : \mathsf{nonnull}\ C}{T; V \vdash \mathbf{r}_k : (\mathsf{tag}((V(\mathbf{r}_k)),\ \mathit{descendanttags}(H, C)))\ \mathsf{ptr}}$$

where $\mathit{descendanttags}(H, C)$ yields the list of class tags of the classes in the subtree rooted at C in class hierarchy H. Here we keep the abstract value $V(\mathbf{r}_k)$ that is needed to handle the Cool `case` construct as discussed in Section C.2.1.

We could also have a rule for offset 4, the object size, but this is only needed by the Cool run-time and is not normally used in user code. At offset 8 is the dispatch table and offsets 12 and higher are attributes.

$$\frac{T; V \vdash \mathbf{r}_k : \mathsf{nonnull}\ C \quad T; V \vdash e : \mathsf{constword}(m) \quad (m = 8)}{T; V \vdash \mathsf{add}\ \mathbf{r}_k\ e : (\mathsf{dispatch}(V(\mathbf{r}_k)))\ \mathsf{ptr}}$$

$$\frac{T; V \vdash e_1 : \mathsf{nonnull}\ C \quad T; V \vdash e_2 : \mathsf{constword}(m) \quad (H(C).A(m) = \gamma) \quad (m \geq 12)}{T; V \vdash \mathsf{add}\ e_1\ e_2 : \gamma\ \mathsf{ptr}}$$

We assume that the class information $H(C)$ has a component A that is a map from valid offsets to the type of the attribute at that offset. There are also rules for the symmetric case where e_1 is an integer and e_2 is an object, but we do not list them here.

We consider offsets from the dispatch table as valid methods. Though the tables and methods are the same, we have different types depending on how the dispatch table was obtained (static or dynamic dispatch).

$$\frac{T; V \vdash e_1 : \mathsf{dispatch}(v) \quad T; V \vdash e_2 : \mathsf{constword}(m)}{T; V \vdash \mathsf{add}\ e_1\ e_2 : (\mathsf{method}(v,\ m))\ \mathsf{ptr}}$$

$$\frac{T; V \vdash e_1 : \mathsf{sdispatch}(C) \quad T; V \vdash e_2 : \mathsf{constword}(m)}{T; V \vdash \mathsf{add}\ e_1\ e_2 : (\mathsf{smethod}(C,\ m))\ \mathsf{ptr}}$$

Similar to looking up in objects, there are rules for the symmetric case, which we elide.

The class object table contains prototype objects at offsets 0 (mod 8) and initialization methods at offsets 4 (mod 8).

$$\frac{T; V \vdash e : \mathsf{classobjoffset}(n,\ v,\ \vec{m}) \quad T; V \vdash \mathbf{r}_k : \mathsf{nonnull}\ \gamma \quad (V(\mathbf{r}_k) = v) \quad (0 = \vec{m}\ (\mathrm{mod}\ 8))}{T; V \vdash e : \gamma\ \mathsf{ptr}}$$

$$\frac{T; V \vdash e : \mathsf{classobjoffset}(n,\ v,\ \vec{m}) \quad T; V \vdash \mathbf{r}_k : \mathsf{nonnull}\ C \quad (V(\mathbf{r}_k) = v) \quad (4 = \vec{m}\ (\mathrm{mod}\ 8))}{T; V \vdash e : (\mathsf{imethod}(C))\ \mathsf{ptr}}$$

where we overload $=$ in the above to mean that each element in the list modulo 8 is equal to 0 (or 4, respectively). Similar to looking up in dispatch tables, there are rules for the symmetric case, which we elide.

Basic Arithmetic. There several typing rules for arithmetic for integers that evaluate the arithmetic expression that would be computed at runtime.

$$\frac{T; V \vdash e_1 : \mathsf{constword}(m_1) \quad T; V \vdash e_2 : \mathsf{constword}(m_2)}{T; V \vdash \mathtt{add}\ e_1\ e_2 : \mathsf{constword}((m_1 + m_2))}$$

$$\frac{T; V \vdash e_1 : \mathsf{constword}(m_1) \quad T; V \vdash e_2 : \mathsf{constword}(m_2)}{T; V \vdash \mathtt{sub}\ e_1\ e_2 : \mathsf{constword}((m_1 - m_2))}$$

$$\frac{T; V \vdash e_1 : \mathsf{constword}(m_1) \quad T; V \vdash e_2 : \mathsf{constword}(m_2)}{T; V \vdash \mathtt{mult}\ e_1\ e_2 : \mathsf{constword}((m_1 * m_2))}$$

$$\frac{T; V \vdash e_1 : \mathsf{constword}(m_1) \quad T; V \vdash e_2 : \mathsf{constword}(m_2)}{T; V \vdash \mathtt{div}\ e_1\ e_2 : \mathsf{constword}((m_1 / m_2))}$$

$$\frac{T; V \vdash e_1 : \mathsf{constword}(m_1) \quad T; V \vdash e_2 : \mathsf{constword}(m_2)}{T; V \vdash \mathtt{sll}\ e_1\ e_2 : \mathsf{constword}((m_1 \ll m_2))}$$

These operations also apply to words. We only the show the rule for addition and elide the rules for the other arithmetic expressions.

$$\frac{T; V \vdash e_1 : \mathsf{word} \quad T; V \vdash e_2 : \mathsf{word}}{T; V \vdash \mathtt{add}\ e_1\ e_2 : \mathsf{word}}$$

Note that since $\mathsf{constword}(m) \leq \mathsf{word}$, this rule applies to additions between something of type $\mathsf{constword}(m)$ and something of type word using subsumption (discussed below).

Tag Offsets. Offsets into the class object table are computed using the class tag to implement `new SELF_TYPE`. This requires some additional bookkeeping.

$$\frac{T; V \vdash e_1 : \mathsf{tag}(v,\ \vec{m}) \quad T; V \vdash e_2 : \mathsf{constword}(m')}{T; V \vdash \mathtt{add}\ e_1\ e_2 : \mathsf{tagoffset}(n,\ v,\ (\vec{m} + m'))}$$

where we write $\vec{m} + m'$ to mean the list that is obtained by adding m' to each element in the list $\vec{m}$. There are similar rules for the other arithmetic expressions and for the symmetric case that we do not list here.

We allow an offset into the class object table as follows:

$$\frac{T; V \vdash e_1 : \&\ \mathtt{class\_objTab} \quad T; V \vdash e_2 : \mathsf{tagoffset}(n,\ v,\ \vec{m})}{T; V \vdash \mathtt{add}\ e_1\ e_2 : \mathsf{classobjoffset}(n,\ v,\ \vec{m})}$$

With such an offset, we allow some more arithmetic with the following rule (and with similar rules for other arithmetic expressions and the symmetric case which are elided).

$$\frac{T; V \vdash e_1 : \mathsf{classobjoffset}(n,\ v,\ \vec{m}) \quad T; V \vdash e_2 : \mathsf{constword}(m')}{T; V \vdash \mathtt{add}\ e_1\ e_2 : \mathsf{classobjoffset}(n,\ v,\ (\vec{m} + m'))}$$

Subsumption. Recall that we defined the subtyping judgment to mean that if $\sigma \leq \sigma'$, then whenever something of type σ' is expected, we can provide something of type σ. This is made explicit with the following general rule, known as *subsumption*.

$$\frac{T; V \vdash e : \sigma \quad \sigma \leq \sigma'}{T; V \vdash e : \sigma'}$$

C.2.3 Subtyping Judgment

The subtyping judgment for Coolaid is a small extension of subtyping in Cool. This is necessary to check that, for example, writes to memory conform to the type of the memory location. We begin with the subtyping rules from Cool and the standard reflexivity and transitivity rules.

$$\frac{}{C \leq_H H(C).P} \qquad \frac{C \leq D}{\mathsf{selftype}\ C \leq \mathsf{selftype}\ D} \qquad \frac{C \leq D}{\mathsf{selftype}\ C \leq D}$$

$$\frac{}{\sigma \leq \sigma} \qquad \frac{\sigma_1 \leq \sigma_2 \quad \sigma_2 \leq \sigma_3}{\sigma_1 \leq \sigma_3}$$

where we say that the class hierarchy H has a component P that is the parent class, i.e., $H(C).P$ is parent class of C.

We introduced a new type unknown that is type of registers when no additional information is known, so any type conforms to unknown.

$$\frac{}{\sigma \leq \mathsf{unknown}}$$

The integer 0 or `void` value can be used whenever an object is expected. A specific integer value (include 0) can be used whenever a machine word is expected.

$$\frac{}{\mathsf{constword}(0) \leq \gamma} \qquad \frac{}{\mathsf{constword}(m) \leq \mathsf{word}}$$

We can provide something that is non-null when non-nullness is not required; similarly, for read-only.

$$\frac{\gamma_1 \leq \gamma_2}{\mathsf{nonnull}\ \gamma_1 \leq \mathsf{nonnull}\ \gamma_2} \qquad \frac{\gamma_1 \leq \gamma_2}{\mathsf{nonnull}\ \gamma_1 \leq \gamma_2}$$

$$\frac{\iota_1 \leq \iota_2}{\mathsf{ro}\ \iota_1 \leq \mathsf{ro}\ \iota_2} \qquad \frac{\iota_1 \leq \iota_2}{\mathsf{ro}\ \iota_1 \leq \iota_2}$$

where ι stands for either a class C or word.

D Exceptions

To handle exceptions, we extend our types to include the type of exception frames exc and the type of fields of an exception frame.

τ	::=	...	
	|	exc	an exception frame
	|	catch(v)	the catch block of exception frame v

There is no explicit introduction rule for the exc type, but rather will be type assumed for the register pointing to the current exception frame (denoted $\mathbf{r}_{xp}$) as part of a method's pre-condition. More precisely, exc is type of an exception frame in the activation record of a caller.

D.1 Verification Procedure

Throw. A throw is implemented by restoring the state of execution and then making an indirect jump to the nearest enclosing catch based on the current exception frame.

If the nearest enclosing catch is in a caller, then we check that the post-condition of an exceptional return has been established.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{catch}(v) \\ T; V \vdash \mathbf{r}_{rv} : C \\ V(\mathbf{r}_{xp}) = v \end{array}}{T; V \vdash n \,\#\, \texttt{jump [}\mathbf{r}_k\texttt{]}\ \mathsf{ok}}$$

When the nearest enclosing catch is the current method, we simply jump and begin verifying the catch block.

$$\frac{T; V \vdash \mathbf{r}_k : \&\, l \quad T; V \vdash l \,\#\, P(l)\ \mathsf{ok}}{T; V \vdash n \,\#\, \texttt{jump [}\mathbf{r}_k\texttt{]}\ \mathsf{ok}}$$

As an alternative, we might consider eagerly checking that the state has been restored from the exception frame, but it is not necessary.

Method Call. If the nearest enclosing catch is in a caller, then the method call is the same as before, except that the pre-condition now requires that the exception register has type exc.

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{method}(v,\ m) \\ (V(\mathbf{r}_{arg_0}) = v) \\ T; V \vdash \mathbf{r}_{arg_0} : \mathsf{nonnull}\ C \\ T; V \vdash \mathbf{r}_{arg_1} : \sigma_1 \\ \quad\vdots \\ T; V \vdash \mathbf{r}_{arg_p} : \sigma_p \\ (H(C).M(m) = \langle C_1, C_2, \ldots, C_{p+1} \rangle) \\ \sigma_j \leq C_j \quad (\text{for } 1 \leq j \leq p) \\ T; V \vdash \mathbf{r}_{xp} : \mathsf{exc} \\ T; V \vdash \mathbf{r}_{ra} : \&\ n{+}1 \\ cs(T)[C_{p+1}/\mathbf{r}_{rv}]; cs(V) \vdash n{+}1 \,\#\, P(n{+}1)\ \mathsf{ok} \end{array}}{T; V \vdash n \,\#\, \texttt{jump [}\mathbf{r}_k\texttt{]}\ \mathsf{ok}}\ \mathsf{method}_{\mathsf{class}}$$

Otherwise, when the nearest enclosing catch is within the current method, we first check that the exception pointer points to a valid exception frame and then verify the catch block assuming the method returns exceptionally.

The verification of a method call where the nearest enclosing catch block in the current method is as follows:

$$\frac{\begin{array}{l} T; V \vdash \mathbf{r}_k : \mathsf{method}(v,\ m) \\ (V(\mathbf{r}_{arg_0}) = v) \\ T; V \vdash \mathbf{r}_{arg_0} : \mathsf{nonnull}\ C \\ T; V \vdash \mathbf{r}_{arg_1} : \sigma_1 \\ \quad \vdots \\ T; V \vdash \mathbf{r}_{arg_p} : \sigma_p \\ (H(C).M(m) = \langle C_1, C_2, \ldots, C_{p+1} \rangle) \\ \sigma_j \leq C_j \quad (\text{for } 1 \leq j \leq p) \\ T; V \vdash \mathbf{r}_{ra} : \&\ n{+}1 \\ T; V \vdash \mathbf{r}_{xp} : \mathsf{SP0} + n \\ T; V \vdash \mathbf{r}_{sp} : \mathsf{SP0} + m \\ (m \leq n - 4) \\ T; V \vdash \texttt{mem[}\mathbf{r}_{xp}\texttt{]} : \&\ l \\ cs(T)[C_{p+1}/\mathbf{r}_{rv}]; cs(V) \vdash n{+}1\ \#\ P(n{+}1)\ \mathsf{ok} \\ T'[\texttt{Object}/\mathbf{r}_{rv}]; V' \vdash l\ \#\ P(l)\ \mathsf{ok} \quad (T' \text{ and } V' \text{ are "scrambled"}) \end{array}}{T; V \vdash n\ \#\ \texttt{jump [}\mathbf{r}_k\texttt{]}\ \mathsf{ok}}$$