

The Coolaid Reference Manual

Version 3.0

Bor-Yuh Evan Chang

George Necula

April 10, 2006

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.

Last modified on Mon Apr 10 15:56:56 PDT 2006

*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.

At the time of this writing, Coolaid is checking for 175 different correctness conditions. We tested Coolaid on 8200 programs generated by student compilers from the Spring 2002 and Spring 2003 offerings 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 1300 of the 5600 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 200 of the 2600 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 CS164 reference Cool compiler), as follows:

```
cp ~cs164/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.

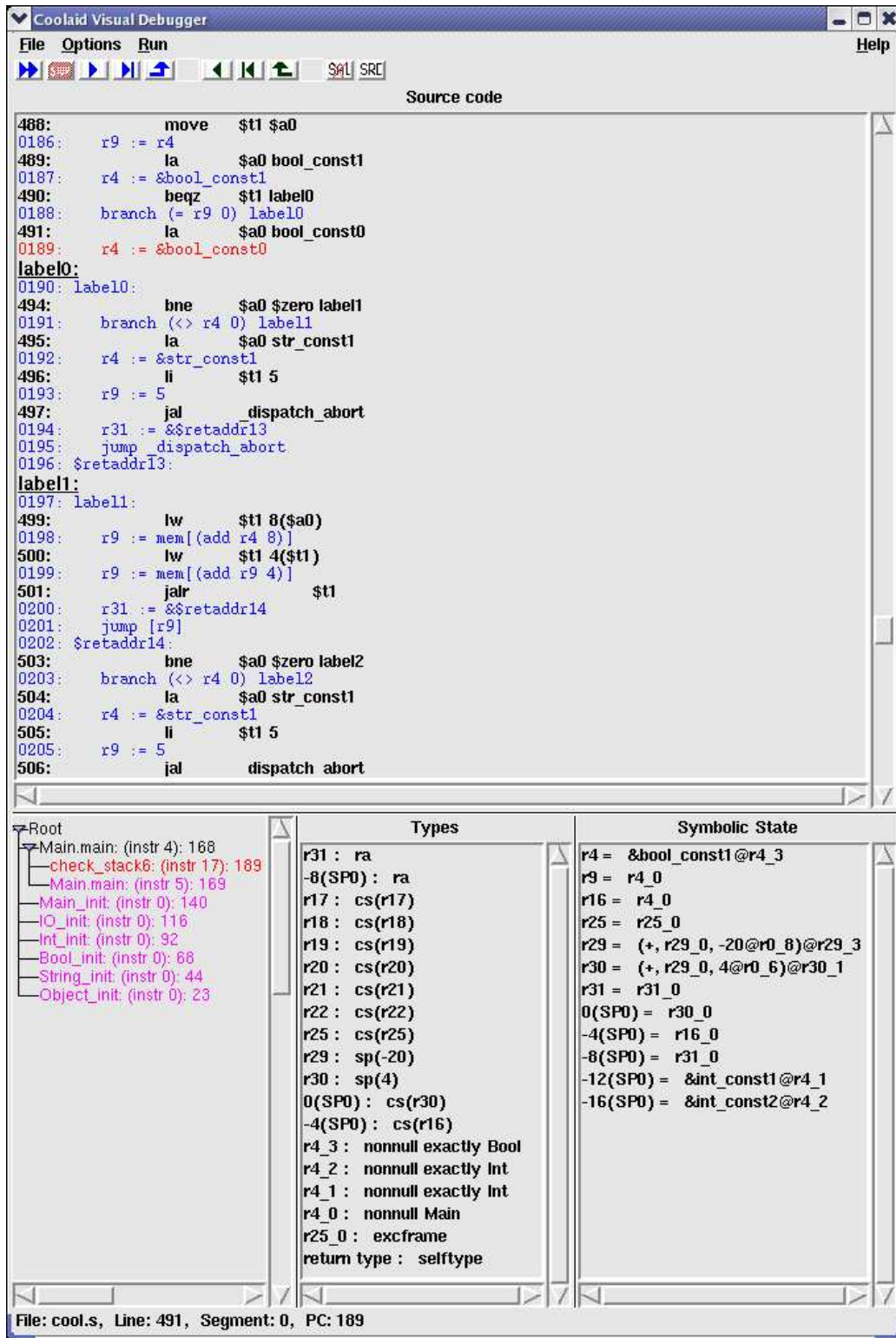


Figure 1: Coolaid.

3 The Graphical User Interface

The main Coolaid window is divided into four 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 panes in the lower right display 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.

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 basic block name (i.e., the label at the start of the basic block) 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 basic block `check_stack6` of method `Main.main` with 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

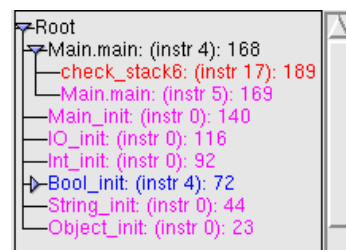



Figure 2: Verification Path.

clicking the ►► (Run) button (and can then use the  (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.

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 annotation in the output assembly file. The reference Cool compiler does so with the `-L` command-line option. If there are line annotations 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 Symbolic and Type States Display

The panes 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. In the Symbolic State display, Coolaid shows a symbolic representation of the machine state. When Coolaid cannot determine the contents of a register, it introduces a new *symbolic value* to stand for that unknown value (i.e., a new “pseudo-register” that always contains that value); for readability, these pseudo-registers are generally named by subscripting the register that originally contained that value (e.g., `$a0.0`). Thus, for each register or stack slot, Coolaid accumulates an expression showing what is known about that machine register or stack slot.

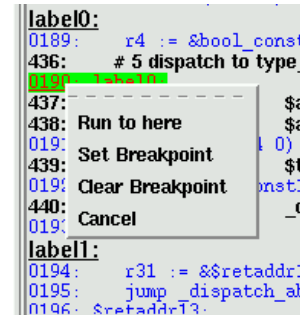


Figure 3: Breakpoints.

In the Types display, Coolaid shows types assigned to registers and/or pseudo-registers (such as `r4_0` is a non-null address of an object of static type `I0`). Coolaid uses numeric register names (`r0` to `r31`), but one can turn on the use of mnemonic names (e.g., `$a0`) with `Options → Arch → Numeric Register Names`.

To assign types to intermediate expressions (and display them), Coolaid introduces pseudo-registers for every sub-expression, which are shown in the Symbolic State display by post-fixing the expression with `@pseudo_register`. For example, `&bool_const1@r4_3` stands for the address of label `bool_const1`, which is the same as pseudo-register `r4_3`.

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.

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 `I0`. 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 a 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 {
    next() : Parent { ... };
};
class Child inherits Parent {
};
class Main {
    scan(y : Child) : Object {
        let x : Parent <- y in
        while not(isvoid x) loop
            x <- x.next()
        pool
    };
};

```

-- A "sequence" class
-- Iterator method

-- Scan the sequence

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 `Ldispatch_abort` labels some code to issue an error message and to abort because of the attempt to dispatch on a *void* value, and `Ldone` 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}_{arg0}) 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 `Ldispatch_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 (and \mathbf{r}_y) is non-void. Then, in line 5, it determines that reading

```

1 Main.scan:
    ⋮
     $\mathbf{r}_y : Child$ 
2 Lbody:     $\mathbf{r}_x := \mathbf{r}_y$ 
     $\mathbf{r}_x = \mathbf{r}_y, \mathbf{r}_y : Child$ 
3 Loop:
4    branch (=  $\mathbf{r}_x$  0) Ldispatch.abort
     $\mathbf{r}_x = \mathbf{r}_y, \mathbf{r}_y : Child$ 
5     $\mathbf{r}_t := \text{mem}[(\text{add } \mathbf{r}_x \text{ } 8)]$ 
     $\mathbf{r}_t : \text{dispatch}(\mathbf{r}_x), \mathbf{r}_x = \mathbf{r}_y, \mathbf{r}_y : \text{nonnull } Child$ 
6     $\mathbf{r}_t := \text{mem}[(\text{add } \mathbf{r}_t \text{ } 12)]$ 
     $\mathbf{r}_t : \text{method}(\mathbf{r}_x, \text{ } 12), \mathbf{r}_x = \mathbf{r}_y, \mathbf{r}_y : \text{nonnull } Child$ 
7     $\mathbf{r}_{arg_0} := \mathbf{r}_x$ 
     $\mathbf{r}_{arg_0} = \mathbf{r}_x = \mathbf{r}_y, \mathbf{r}_t : \text{method}(\mathbf{r}_x, \text{ } 12), \mathbf{r}_y : \text{nonnull } Child$ 
8     $\mathbf{r}_{ra} := \text{L}_{ret}$ 
     $\mathbf{r}_{ra} = \&\text{L}_{ret}, \mathbf{r}_{arg_0} = \mathbf{r}_x = \mathbf{r}_y, \mathbf{r}_t : \text{method}(\mathbf{r}_x, \text{ } 12), \mathbf{r}_y : \text{nonnull } Child$ 
9    jump [ $\mathbf{r}_{r_t}$ ]
     $\mathbf{r}_{rv} : Parent$ 
10 Lret:
11    branch (=  $\mathbf{r}_{rv}$  0) Ldone
     $\mathbf{r}_{rv} : \text{nonnull } Parent$ 
12     $\mathbf{r}_x := \mathbf{r}_{rv}$ 
     $\mathbf{r}_x = \mathbf{r}_{rv}, \mathbf{r}_{rv} : \text{nonnull } Parent$ 
13    jump Loop
    ⋮

```


from offset 8 into an object \mathbf{r}_x yields the dispatch table of that object and records this as $\mathbf{r}_t : \text{dispatch}(\mathbf{r}_x)$. 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 by $\mathbf{r}_t : \text{method}(\mathbf{r}_x, 12)$. 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 $\mathbf{r}_t : \text{method}(\mathbf{r}_x, 12)$ and $\mathbf{r}_x : \text{Child}$, the verifier can consult the class hierarchy accompanying the compiled code to find that a method `next` is being called. Since $\mathbf{r}_{\text{arg0}} = \mathbf{r}_x$, 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 `Ldone`, 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 $\mathbf{r}_x : \text{Parent}$ 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 registers, pseudo-registers, and stack slots. Many of these types refer to (pseudo-)register names. We use the r to denote such a name. We also use n 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 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
 - exactly: means that the dynamic type of the value is shown exactly. Without this qualifier, the type is an upper bound of the dynamic type as in Cool’s type system. For example, `& Int_protObj` has type “nonnull exactly Int”.
 - classof(*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 *n*th method of class *C*.
- imethod(*r*): the type of the address of the initialization method fetched from the object stored in register or stack slot *r*. 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.

- sp(n): the type of the value of the stack pointer register on entry to the current method + n .
- 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.
- excfame: the type of the value of the exception frame pointer at the time the current method was called.
- handler: the type of the value of the exception handler at the time the current method was called.

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	<u>O</u> ptions → <u>C</u> ool → <u>V</u> erbose	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	<u>O</u> ptions → <u>C</u> ool → <u>U</u> pdate 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	<u>O</u> ptions → <u>G</u> UI → <u>S</u> ave State <u>I</u> nterval	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	<u>O</u> ptions → <u>G</u> UI → <u>S</u> how <u>S</u> AL	Show SAL instructions interleaved with the MIPS instructions. See Appendix B for details on SAL.

command-line	menu	description
<code>-showSource</code>	<u>O</u> ptions → <u>G</u> UI → Show Source	Show Cool source interleaved with the MIPS instructions. The Cool compiler must include line number information (<code>-L</code> command-line option).
<code>-cool_exc/</code> <code>-no-cool_exc</code>	<u>O</u> ptions → <u>C</u> ool → <u>C</u> heck <u>E</u> xceptions	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	<i>Inst</i>	::=	<i>Label</i> :	a label
			<i>Reg</i> := <i>Exp</i>	an assignment to a register
			<i>Reg</i> := mem [<i>Exp</i>]	a memory read from address <i>Exp</i>
			mem [<i>Exp</i>] := <i>Exp</i>	a memory write
			jump <i>Label</i>	a jump to the given label
			jump [<i>Exp</i>]	a indirect jump to the given address
			branch <i>Exp</i> <i>ntLabel</i>	a branch if expression is not zero
registers	<i>Reg</i>	::=	r1 ... rn	machine registers
expressions	<i>Exp</i>	::=	n	integer constants
			<i>Reg</i>	machine registers
			& <i>Label</i>	address of a label
			(<i>Op</i> <i>Exp</i> <i>Exp</i>)	binary operations
operators	<i>Op</i>	::=	add sub sll = <> ...	

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 Frequently Asked Questions

If you have trouble using Coolaid, please send email to cs164@inst.eecs.berkeley.edu.