The Cool Runtime System

1 Introduction

The runtime system consists of a set of hand-coded assembly language functions that are used as subroutines by Cool programs. Under `spim` the runtime system is in the file `trap.handler`. The use of the runtime system is a concern only for code generation, which must adhere to the interface provided by the runtime system.

The runtime system contains four classes of routines:

1. startup code, which invokes the main method of the main program;
2. the code for methods of predefined classes (**Object**, **IO**, **String**);
3. a few special procedures needed by Cool programs to test objects for equality and handle runtime errors;
4. the garbage collector.

The Cool runtime system is in the file /usr/class/cs143/lib/trap.handler; it is loaded automatically whenever spim/xspim is invoked. Comments in the file explain how the pre-defined functions are called.

The following sections describe what the Cool runtime system assumes about the generated code and what the runtime system provides to the generated code. However, additional restrictions must be placed on the generated code for use with Coolaid, a debugging tool for Cool compilers based on program verification (see the Coolaid Reference Manual).

Coolaid specific restrictions are marked in this manner.

This document does not describe the expected behavior of Cool programs; refer to Section 13 of the Cool Reference Manual for a formal description of the execution semantics of Cool programs.

2 Objects

The object layout describes the location of attributes, as well as the meta-data that must be associated with each object.

Modified from Section 7 of *A Tour of the Cool Support Code* (©1995-1996 by Alex Aiken). Last modified on Wed Nov 10 04:34:45 PST 2004.

offset -4	Garbage Collector Tag
offset 0	Class tag
offset 4	Object size (in 32-bit words)
offset 8	Dispatch pointer
offset 12...	Attributes

Figure 1: Object layout.

2.1 Object Layout

The first three 32-bit words of each object are assumed to contain a class tag, the object size, and a pointer for dispatch information. In addition, the garbage collector requires that the word immediately before an object contain -1; this word is not part of the object.

Figure 1 shows the layout of a Cool object; the offsets are given in numbers of bytes. The garbage collection tag is -1. The class tag is a 32-bit integer identifying the class of the object and thus should be unique for each class. The runtime system uses the class tag in equality comparisons between objects of the basic classes and in the abort functions to index a table containing the name of each class.

The object size field and garbage collector tag are maintained by the runtime system; only the runtime system should create new objects. However, *prototype objects* (see below) must be coded directly by the code generator in the static data area, so the code generator should initialize the object size field and garbage collector tag of prototypes properly. Any statically generated objects must also initialize these fields.

The dispatch pointer points to a table used to invoke the correct method implementation on dynamic dispatches. It is never actually used by the runtime system, so the layout of dispatch information is not fixed. However, Coolaid expects pointers to the methods (i.e., the labels) of an object to reside directly in the table indicated by the dispatch pointer—not through, say, in separate table via another level of indirection.

For **Int** objects, the only attribute is the 32-bit value of the integer. For Bool objects, the only attribute is the 32-bit value 1 or 0, representing either true or false. The first attribute of String objects is an object pointer to an Int object representing the size of the string. The actual sequence of ASCII characters of the string starts at the second attribute (offset 16), terminates with a 0, and is then padded with 0's to a word boundary.

The value *void* is a null pointer and is represented by the 32-bit value 0. All uninitialized variables (except variables of type Int, Bool, and String; see the Cool Reference Manual) should be set to *void* by default.

For use with Coolaid, the layout of attributes is fixed (based on the annotations packaged with the assembly code). The attributes of the parent class should come first, then followed by the new attributes in the same order as in the source. An example of the required object layout is shown in Figure 2. For all statically allocated objects, the attributes must be statically either *void* or some other value of the declared type of that attribute.

2.2 Prototype Objects

The only way to allocate a new object in the heap is to use the `Object.copy` method. Thus, there must be an object of every class that can be copied. For each class C in the Cool program, the code generator should produce a skeleton C object in the data area; this object is the prototype of class C.

```
class Grandparent {
  first : Object;
};
class Parent inherits Grandparent {
  second : Object;
}
class Child inherits Parent {
  third : Object;
  fourth : Object;
}
```

offset -4	Garbage Collector Tag
offset 0	Class tag
offset 4	Object size (in 32-bit words)
offset 8	Dispatch pointer
offset 12	Attribute `first`
offset 16	Attribute `second`
offset 20	Attribute `third`
offset 24	Attribute `fourth`

Figure 2: Example: object layout for `Child`.

For each prototype object, the garbage collection tag, class tag, object size, and dispatch information must be set correctly. For the basic classes **Int**, **Bool**, and **String**, the attributes should be set to the defaults specified in the Cool Reference Manual. For the other classes, the attributes of the prototypes may be whatever you find convenient for your implementation. Coolaid does require that they be set to either *void* or some other value of the declared type of that attribute. This would be prudent anyway.

3 Code Generation Guidelines

While Coolaid attempts to be as flexible as possible, it must impose a few restrictions on the code generation strategy for certain Cool constructs.

3.1 Case

Recall that a case expression provides a runtime type test on objects. The class tag uniquely identifies the dynamic type of the object, so a reasonable strategy is to generate a series of conditionals comparing the class tag of the object to the tags of the types specified in the branches of the case expression. Coolaid requires that the branch of the case expression to evaluate be determined by loading the the class tag of the object on which the case is testing and comparing that value with constants (in branch instructions).

3.2 Selftype

Generating code for the expression `new SELF_TYPE` requires additional bookkeeping of which Coolaid must be aware. Since we need to create an object with the same dynamic type as the **self** object, one possibility is to use the class tag to determine the prototype object to copy. This can be done by creating table of pointers to the prototype objects of each class indexed by the class tag. Coolaid expects such a table to be present in the static data segment with the label `class_objTab` to tell Coolaid what are the labels for the prototype objects and the initialization methods (see Section 4 for a description of the table). An offset into this table for creating an object with the same dynamic type as **self** can be computed by reading the class tag for **self**, add/subtract/multiply this value by constants, and add it to the address of `class_objTab`.

Main_protObj	The prototype object of class **Main**	Data
Main_init	Code that initializes an object of class **Main** passed in **$a0**	Code
Main.main	The main method for class **Main** **$a0** contains the initial **Main** object	Code
Int_protObj	The prototype object of class **Int**	Data
Int_init	Code that initializes an object of class **Int** passed in **$a0**	Code
String_protObj	The prototype object of class **String**	Data
String_init	Code initializing an object of class **String** passed in **$a0**	Code
_int_tag	A single word containing the class tag for the **Int** class	Data
_bool_tag	A single word containing the class tag for the **Bool** class	Data
_string_tag	A single word containing the class tag for the **String** class with the class tag	Data
bool_const0	The **Bool** object representing the boolean value false	Data
class_nameTab	A table, which at index (class tag) $*$ 8 contains a pointer to a **String** object containing the name of the class associated	Data
class_objTab	A table, which at index (class tag) $*$ 8 contains a pointer to the prototype object and at index (class tag) $*$ 8 + 4 contains a pointer to the initialization method for that class.	Data
$C.m$	Method m of class C.	Code

Figure 3: Fixed labels.

4 Expected Labels

The Cool runtime system refers to the fixed labels listed in Figure 3. Each entry describes what the runtime system expects to find at a particular label and where (code/data segment) the label should appear. Note that though the runtime does not require a label for the **Bool** object representing the boolean value true, there is no need for code that initializes an object of class **Bool** if the generated code also contains a definition of the **Bool** object for true in the static data area.

Coolaid requires that labels for methods adhere to the following naming convention.

$$C\,.\,m \quad \text{for the code of method } m \text{ of class } C$$

Also, it requires an additional table in the data segment at the label `class_objTab` to indicate the addresses of the prototype objects and the initialization methods. The table should be laid out so that at offset $8 \cdot t$ is the prototype object and at offset $8 \cdot t + 4$ is the initialization method for the class with tag t. Thus, the `class_objTab` looks something like the following:

```
class_objTab:   Object_protObj
                Object_init
                IO_protObj
                IO_init
                    ⋮
```

assuming class **Object** and **IO** have tags 0 and 1, respectively, and with their prototype objects at the labels `Object_protObj` and `IO_protObj` and their initialization methods at `Object_init` and `IO_init`, respectively. A summary of the required labels for Coolaid also appears in Figure 3.

5 Register and Calling Conventions

The primitive methods in the runtime system expect arguments in register **$a0** and on the stack. Usually, **$a0** contains the **self** object of the dispatch, and the result is also returned in **$a0**. Additional arguments should be on top of the stack, first argument pushed first (an issue only for **String.substr**, which takes two arguments). Some of the primitive runtime procedures expect arguments in particular registers (see Figure 4).

Coolaid expects a similar calling convention for Cool methods. The return address should be passed in **$ra**. The standard callee-saved registers on the MIPS architecture **$s0**, **$s1**, **$s2**, **$s3**, **$s4**, **$s5**, **$s6**, and **$s7**, along with the frame pointer **$fp** are considered callee-saved by Coolaid, as well as the runtime system. (Register **$s7** is used by the garbage collector and should never be touched by the generated code.) The **self** object must be passed in **$a0**, while additional arguments should be on top of the stack, first argument pushed first. No other assumptions about the contents of the registers should be assumed on method entry. In other words, an activation record for a method with n arguments should look like the following:

⋮	low addresses
argument n	
argument $n-1$	
⋮	
argument 2	
argument 1	high addresses

One is free to design the remainder of the activation record. On method return, Coolaid expects that the callee pops the entire activation record *including* the arguments. For the initializations methods, Coolaid and the runtime system consider **$a0** to be callee-saved (in addition to the callee-saved registers for normal methods).

The following registers are used by the runtime system:

Scratch registers	**$v0**,**$v1**,**$a0**–**$a2**,**$t0**–**$t4**
Heap pointer	**$gp**
Limit pointer	**$s7**

The runtime system may modify any of the scratch registers without restoring them (unless otherwise specified for a particular routine). The heap pointer **$gp** is used to keep track of the next free word on the heap, and the limit pointer **$s7** is used to keep track of where the heap ends. These two registers should

not be modified or used by the generated code—they are maintained entirely in the runtime system. Registers **$at**,**$sp**, and **$ra** may also be modified by the runtime system.

Coolaid expects that at every label in the code segment, the value of the stack pointer (**$sp**) is the same regardless of the path of execution. For example, consider the following MIPS assembly code.

```
begin_conditional: beqz  $a0 true_branch
                   addiu $sp $sp -12
true_branch:       addiu $sp $sp -16
end_conditional:
                        ⋮
```

Such a program is not allowed, as at the label `end_conditional`, **$sp** has different values depending on the branch that is taken (assuming **$sp** is the same on all paths to `begin_conditional`). Regardless, such an invariant is useful to ensure that stack slots can always be read using constant offsets from **$sp**.

6 Runtime Interface

Figure 4 lists labels for methods defined in the runtime system that may be called by the generated code. Coolaid does not make any assumptions about the result of an **equality_test**. For example, the generated code cannot pass to **equality_test** the address of labels in **$a0** and **$a1** and then jump to the result to conditionally branch based on the equality test. This is because branches must be explicit syntactically in the generated code for Coolaid to identify them as such.

6.1 Execution Startup

On startup, the following actions are provided by the runtime system:

1. A fresh copy of the **Main** prototype object is made on the heap and then initialized by a call to **Main_init**.

   The code generator must define **Main_init**. **Main_init** should execute all initialization code of **Main**'s parent classes and finally execute the initializations of attributes in **Main** (if there are any).

2. Control is transferred to **Main.main**, passing a pointer to the newly created **Main** object in register **$a0**. Register **$ra** contains the return address.

3. If control returns from **Main.main**, execution halts with the message "COOL program successfully executed".

6.2 The Garbage Collector

The Cool runtime environment includes two different garbage collectors, a generational garbage collector and a stop-and-copy collector. The generational collector is the one used for programming assignments; the stop-and-copy collector is not currently used. The generational collector automatically scans memory for objects that may still be in use by the program and copies them into a new (and hopefully much smaller) area of memory.

Object.copy	A procedure returning a fresh copy of the object passed in **$a0** Result will be in **$a0**
Object.abort	A procedure that prints out the class name of the object in **$a0** Terminates program execution
Object.type_name	Returns the name of the class of object passed in **$a0** as a string object Uses the class tag and the table class_nameTab
IO.out_string	The value of the string object on top of the stack is printed to the terminal.
IO.out_int	The integer value of the Int object on top of the stack is printed to the terminal.
IO.in_string	Reads a string from the terminal and returns the read string object in **$a0**. (The newline that terminates the input is not part of the string)
IO.in_int	Reads an integer from the terminal and returns the read int object in **$a0**.
String.length	Returns the integer object which is the length of the string object passed in **$a0**. Result in **$a0**.
String.concat	Returns a new string, made from concatenating the string object on top of the stack to the string object in **$a0**. Return value in **$a0**
String.substr	Returns the substring of the string object passed in **$a0**, from index i with length l. The length is defined by the integer object on top of the stack, and the index by the integer object on the stack below l. Result in **$a0**.
equality_test	Tests whether the objects passed in $t1 and $t2 have the same primitive type {**Int**,**String**,**Bool**} and the same value. If they do, the value in **$a0** is returned, otherwise **$a1** is returned.
_dispatch_abort	Called when a dispatch is attempted on a void object. Prints the line number, from **$t1**, and filename, from **$a0**, at which the dispatch occurred, and aborts.
_case_abort	Should be called when a case statement has no match. The class name of the object in **$a0** is printed, and execution halts.
_case_abort2	Called when a case is attempted on a void object. Prints the line number, from **$t1**, and filename, from **$a0**, at which the dispatch occurred, and aborts.

Figure 4: Labels defined in the runtime system.

Generated code must contain definitions specifying which of several possible configurations of the garbage collector the runtime system should use. The location **_MemMgr_INITIALIZER** should contain a pointer to an initialization routine for the garbage collector and **_MemMgr_COLLECTOR** should contain a pointer to code for a collector. The options are no collection, generational collection, or stop-and-copy collection; see comments in the trap.handler for the names of the **INITIALIZER** and **COLLECTOR** routines for each case. If the location **_MemMgr_TEST** is non-zero and a garbage collector is enabled, the garbage collector is called on every memory allocation, which is useful for testing that garbage collection and code generation are working properly together.

The collectors assume every even value on the stack that is a valid heap address is a pointer to an object. Thus, a code generator must ensure that even heap addresses on the stack are in fact pointers to objects. Similarly, the collector assumes that any value in an object that is a valid heap address is a pointer to an object (the exceptions are objects of the basic classes, which are handled specially).

The collector updates registers automatically as part of a garbage collection. Which registers are updated is determined by a register mask that can be reset. The mask should have bits set for whichever registers hold heap addresses at the time a garbage collection is invoked; see the file trap.handler for details.

Generated code must notify the collector of every assignment to an attribute. The function `_GenGC_Assign` takes an updated address a in register `$a1` and records information about a that the garbage collector needs. If, for example, the attribute at offset 12 from the `$self` register is updated, then a correct code sequence is

```
sw      $x 12($self)
addiu   $a1 $self 12
jal     _GenGC_Assign
```

Calling `_GenGC_Assign` may cause a garbage collection. Note that if the garbage collector is *not* being used it is equally important *not* to call `_GenGC_Assign`.