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 MARS the runtime system is in the file trap_handler.mars. 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 trap_handler.mars; it is loaded automatically whenever MARS 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.

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.

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 initize the object size field and garbage collector tag of prototypes properly. Any statically generated objects must also initize 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.

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 0s 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.

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.

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.

3   Code Generation Guidelines

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.

3.2   Selftype

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. Let us use the label `class_objTag` to store the labels for the prototype objects and the initization 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_objTag`.

Main_protObj	The prototype object of class **Main**	Data
Main_init	Code that initizes an object of class **Main** passed in **$a0**	Code
Main.main	The main method for class **Main $a0** contains the init **Main** object	Code
Int_protObj	The prototype object of class **Int**	Data
Int_init	Code that initizes an object of class **Int** passed in **$a0**	Code
String_protObj	The prototype object of class **String**	Data
String_init	Code initizing 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) * 4 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 initization method for that class.	Data

Figure 2: Fixed labels.

4   Expected Labels

The Cool runtime system refers to the fixed labels listed in Figure 2. 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 initizes an object of class **Bool** if the generated code also contains a definition of the **Bool** object for true in the static data area.

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 3). 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.

6   Runtime Interface

Figure 3 lists labels for methods defined in the runtime system that may be called by the generated code.

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 initized by a call to **Main_init**. The code generator must define **Main_init**. **Main_init** should execute all initization code of **Main**'s parent classes and finally execute the initizations 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.

Generated code must contain definitions specifying which of several possible configurations of the garbage collector the runtime system should use. The location _MemMgr_INITIZER should contain a pointer to an initization 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.mars for the names of the INITIZER 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

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 3: Labels defined in the runtime system.

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.mars 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.