The Cool Runtime System[*]

1 Introduction

The Cool runtime consists of a library that is linked with the output of the coolc code generator. It provides many subroutines used by Cool programs as well as the startup code that creates an object of class Main and invokes its main() method. 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 library ~cs143/lib/libcool.a; it is loaded automatically by the link stage of the coolc compiler wrapper. 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.

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.

[*]Copyright ©1995-1996 by Alex Aiken. All rights reserved.

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.

```
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`.

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.

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.

Main_protObj	The prototype object of class **Main**
Main_init([self])	Code that initializes an object of class **Main**
Main.main([self])	The main method for class **Main**
Int_protObj	The prototype object of class **Int**
Int_init([self])	Code that initializes an object of class **Int**
String_protObj	The prototype object of class **String**
String_init([self])	Code initializing an object of class **String**
_int_tag	A single word containing the class tag for the **Int** class
_bool_tag	A single word containing the class tag for the **Bool** class
_string_tag	A single word containing the class tag for the **String** class with the class tag
bool_const0	The **Bool** object representing the boolean value false
class_nameTab	A table, which at index (class tag) $*$ 4 contains a pointer to a **String** object containing the name of the class associated
$C.m([self], \ldots)$	Method m of class C. (If you want them to be visible in a debugger.)

Figure 3: Fixed labels.

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 should be initialized to their default values, as defined in the Cool Reference Manual.

3 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. All exported functions should adhere to the calling convention described in Section 4. Argument order is shown, with [**self**] at the front for methods that expect it to be provided.

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.

4 Register and Calling Conventions

All functions provided by the runtime follow the **cdecl** calling convention, modified to support passing **self** objects. Similarly, the functions listed in Figure 3 that are produced during code generation must adhere to the same calling convention.

The modified **cdecl** calling convention is defined as follows:

- All arguments to a function or method are placed on the stack. The topmost value (i.e. at the lowest address on the stack) must be the **self** object (if needed), followed by the leftmost actual argument. Arguments should appear in adjacent locations on the stack, in left-to-right order. The rightmost argument should therefore be at the highest address in the stack (furthest from the top). For a call of the form:

  $$\text{target.method}(\text{arg1}, \text{arg2}, \ldots, \text{argn})$$

  the stack should look like:

⋮		high addresses
argument	n	
argument	$n-1$	
⋮		
argument	2	
argument	1	low addresses
target (if present)		← **%esp**

- If the called function returns a value, it must be placed in **%eax**.
- It is the responsibility of the caller to remove the arguments from the stack (after the callee has returned).
- The callee must preserve the following registers: **%ebx**, **%esi**, **%edi**, **%ebp**, and **%esp**.
- The callee need not preserve the following registers: **%eax**, **%ecx**, and **%edx**. The caller must save them if it wishes the values to be preserved.

This calling convention places constraints on the the high end of the activation record. The code generator is free to design the remainder of the activation record (i.e. what's pushed onto the stack after the **call** instruction is executed) in any manner that is convenient.

5 Runtime Interface

Figure 4 lists labels for methods defined in the runtime system that may be called by the generated code. All arguments must be pointers to valid Cool objects. Argument order is shown, with [**self**] at the front for methods that expect it to be provided.

5.1 Execution Startup

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

1. **Object.copy** is used to duplicate the **main_protObj** prototype object to create a fresh instance of type **Main** on the heap. That instance is 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 as the implicit **self** parameter, as described in Section 4.

Object.copy([self])
Object.abort([self])
 	
Object.type_name([self])
	A method returning a fresh copy of the object passed in **self**
	A procedure that prints out the class name of the object in **self**
	Terminates program execution
	Returns the name of the class of object passed in **self** as a string object
	Uses the class tag and the table class_nameTab

IO.out_string([self],sval)
IO.out_int([self],ival)
IO.in_string([self])
 	
IO.in_int([self])
	The value of the **sval** is printed to the terminal.
	The integer value of the Int object **ival** is printed to the terminal.
	Reads a string from the terminal, and returns the read String object.
	(The newline that terminates the input is not part of the string)
	Reads an integer from the terminal and returns the read val as
	an Int object.

String.length([self])
 	
String.concat([self],other)
 	
String.substr([self],pos,len)
	Returns the integer object which is the length of the string object
	passed in **self**.
	Returns a new string, made from concatenating the string object
	self with the string object in **other**.
	Returns the substring of the string object passed in **self**, starting at
	a position defined by the Int object **pos**. The length of the
	substring is specified by the Int object **len**.

equality_test(a,b,ifeq,ifne)
 	
_dispatch_abort(file,line)
 	
_case_abort(obj)
 	
_case_abort2(file,line)
	Tests whether the objects passed in **a** and **b** have the same
	primitive type {**Int**,**String**,**Bool**} and the same value. If they do,
	the value in **ifeq** is returned, otherwise **ifne** is returned. If the objects
	passed in **a** and **b** are not of primitive type, then the value in **ifeq**
	is returned if they are the same object, otherwise **ifne** is returned.
	Called when a dispatch is attempted on a void object. Prints the line
	number from Int object **line**, and file name from String object **file** at
	which the dispatch occurred, and aborts.
	Should be called when a case statement has no match.
	The class name of the object in **obj** is printed, and execution halts.
	Called when a case is attempted on a void object. Prints the line
	number from Int object **line**, and file name from String object **file** at
	which the case occurred, and aborts.

Figure 4: Labels defined in the runtime system.

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

5.2 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 locations **_MemMgr_Mode**, **_MemMgr_Test**, and **_MemMgr_Debug** should contain integers matching the values of the corresponding flags during code generation.

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. This is performed for the callee-saved registers listed in Section 4. Caller-saved registers are not checked, as it is presumed their values have been preserved on the stack, which the garbage collector will be scanning.

Generated code must notify the collector of every assignment to an attribute. The function `_GenGC_Assign` is provided by the runtime and takes an single argument that must be the address of the attribute that has just been updated. (Note that this is the only call into the runtime that does not pass a pointer to the base of a Cool object.) If, for example, the attribute at offset 12 from the `$self` register is updated, then a correct code sequence is

```
movl    %ecx, 12(%eax)   # modification of attribute
leal    12(%eax), %eax   # address of attribute
pushl   %eax             # pushed on stack as argument
call    _GenGC_Assign    # call into GC
addl    $4, %esp         # pop argument off stack
```

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`.