

In this assignment you will implement garbage collection for the MiniJava compiler.

1 Motivation and Background

Garbage collection is becoming quite common in modern day programming languages. By implementing a simple garbage collector, you will learn how garbage collection works and gain insight into how garbage collection can affect application performance. For this assignment you will be implementing a Mark and Sweep collector.

Currently the MiniJava compiler generates MIPS code. The run-time library is in the file `mips.rtl.s`, which is part of the `MJFull.tgz` tarball posted at the google group page. The generated MIPS program must all be within one file with the main function at the top of that file to enable simulation with the MARS simulator. MiniJava objects are allocated on the heap with the run-time library call `halloc()`, which in this project you will need to replace with a new allocation function that is capable of performing garbage collection if necessary. In the below we will refer to that new function as `halloc_gc()`.

Garbage collection should occur when there is a request for memory allocation in the mutator and there is no more memory available in the heap being managed by the garbage collection module. At that point the memory allocator will attempt to allocate the requested memory from the freelist. If there are no chunks big enough on the freelist, then garbage collection should occur.

The two main pieces of information that the MiniJava compiler needs to provide to the runtime garbage collector is a way to identify the root set for the current run-time stack and a way to determine how large objects in the heap are and which fields in those objects are pointers. You need to modify the compiler so that it generates a pointer map for each procedure to aid in identifying parameters and locals in the root set, and a descriptor for each class to aid in identifying pointers in heap objects and heap object sizes.

2 The Assignment

Your assignment is to extend the MiniJava compiler and run-time library so that the generated MIPS code implements *mark and sweep* garbage collection. For MiniJava, class instances and arrays are dynamically allocated. For this project, you need only handle class instances. Arrays can continue to be allocated in the non-GC controlled heap.

It should be possible to indicate the size of the heap in bytes available for garbage collector controlled dynamic memory allocation. For example,

```
% java -jar MJFull.jar --heap=64 file.java
```

The code generated by the MiniJava compiler (some of which is sucked in from the `mips.rtl.s`

file) should print information about dynamic memory allocations and garbage collections. For example,

```
Allocating an instance of class ### at line XX.  
Failed attempt to allocate an instance of class ### at line YY.  
***** Starting garbage collection  
    Marked ## bytes.  
    Put ## bytes on the freelist.  
Allocating an instance of class ### at line YY.  
...
```

The pound signs indicate the class key for the class being allocated. Each class and method should have its own numeric key to aid in debugging.

You will need to create at least two test input programs and run those test input programs using varying heap sizes.

- The generate MIPS program should gracefully fail when an allocation is requested and there is no more allocatable space left.
- The `halloc_gc()` function should perform garbage collection, if it is not possible to allocate the requested space.
- You should have test cases that create a lot of garbage and those that do not.

Feel free to share such test cases with other groups in the class, but each group must submit their own test cases.

Graph the number of times garbage collection occurs versus heap size for at least one test case where garbage collection occurs at least once for the largest heap size shown on your graph. Explain the results.

3 Suggested Garbage Collector Design

We recommend that the garbage collection module consist of various global variables that maintain information such as the pointer to the head of the freelist, the heap size, and a pointer to the pointer map list. The garbage collector module could also consist of a function that initializes the global GC data structures and the `halloc_gc` routine that does memory allocation and garbage collection if necessary. The following interface could be implemented in C- and compiled to MIPS using the Wisconsin C- compiler (`cmm.jar` is posted on the google group page, see <http://www.cs.wisc.edu/~lenz/compiler.html> for more info).

- **`void init_gc(struct ptrmap * last, int heapsize)`** could initialize all of the data structures needed by the GC. For example, there could be a global variable that maintains a pointer to the last `ptrmap`, and there could be a global pointer to the beginning of the GC managed heap, which should be allocated to be size `heapsize`.

- `int * halloc_gc(int * fp, struct descriptor * class_descr, int lineno)` could replace the current heap allocation routine, `halloc()`. The first parameter is the frame pointer for the function calling `halloc_gc()`. The second parameter is a pointer to the descriptor for the class being allocated. The third parameter is the line number for the allocation, which will help provide debugging information. `halloc_gc` should return a pointer to free space large enough for the requested object. The actual allocation could include three extra words that sit before the returned address. The three extra words could include a pointer to a descriptor, mark flag, and a pointer to the next item in the free list. `halloc_gc` will perform garbage collection if necessary.

For the garbage collector to do its job, you will need to add routines to the run-time library and generate new instructions and data with the MiniJava compiler. Since some of the run-time library routines will be quite involved including the manipulation of singly-linked list data structures, we strongly recommend that you use the provided C- compiler (`cmm.jar`) to write routines that can be compiled to MIPS and then included in the run-time library.

We recommend that you have the MiniJava compiler generate pointer map and class descriptor data structures for providing type information to the run-time garbage collector. The pointer map structure for this project could be as follows (also see `GC-start.c`):

```
struct ptrmap {
    struct ptrmap * prev;
    int gc_key;
    int num_params;
    int num_locals;
    int * ptrmap_data;
};
```

The `prev` field could contain a pointer to the pointer map instance that the compiler previously generated. The `gc_key` provides a way to identify what procedure with which each `ptrmap` instance is associated. Keep in mind that the GC works at runtime. You could modify the MiniJava compiler so that when a function is called, the function puts its `gc_key` value where the frame pointer is pointing. The current compiler puts the first local variable in that location so allocation for local variables should be modified to start at `-4($fp)` now. Although the address for the pointer map associated with the function at the top of the run-time stack could be passed into `halloc_gc`, to calculate the root set it is necessary to traverse the other activation records in the run-time stack via the frame pointers, therefore the `gc_key` in each activation record will help find the appropriate `ptr_map` instance. The `ptrmap_data` field points to an array of integers where there is a one or a zero for each parameter and local indicating which parameters and locals are pointer types. (CHALLENGE: There are a couple of ways to handle the pointer map data structures so that a list structure is not needed. How?)

The class descriptor the MiniJava compiler should generate for the GC is similar to a pointer map, but it indicates which fields in any instance of that class are pointers. The class descriptor for this project could be as follows (also see `GC-start.c`):

```
struct descriptor {
```

```
    int class_key;
    int num_member;
    int * ptr_data;
};
```

Here the class key is just for debug output, because it is easy to have the compiler generate a call to the GC allocation function that passes in the address for the appropriate class descriptor.

The GC could allocate each object instance enough space for its own data and three extra words for storing a descriptor pointer, a mark flag, and a free list next pointer. The allocator could return a pointer to the data segment for use in the mutator code. Only the GC code could know about the prepended fields.

4 Getting Started

1. Start with the MJFull.tgz file that is posted at google groups.
2. Play with the cmm compiler and look at the GC-start.c file (posted at google) to see example C- code. To compile to MARS-compatible MIPS that uses the calling convention from the Patt and Patel book use the following command:

```
java -jar cmm.jar --patt-mips GC-start.c GC-start.s
```

3. Modify the MiniJava compiler to generate the pointer maps, descriptors, passing of gc keys, calls to `halloc_gc()`, and calls to `init_gc`. Also add the MIPS code generated by the C- compiler for your GC.c file to the mips.rtl.s file. Use examples in the `And.java.ptrmap-desc.s` and `GCtest1.java.ptrmap-desc.s` files (posted at google group) to aid with this process.
4. Write the garbage collection routines `init_gc()` and `halloc_gc()` in C-. Instead of their full functionality, have `init_gc()` print out the pointer maps in the provided example files and `halloc_gc()` print out the relevant class descriptor and pointer maps for the functions in the runtime stack.
5. Implement the full functionality for `init_gc()` and `halloc_gc()`.

5 Your Report

The report is an essential part of your completed assignment. Use it to describe your mark and sweep algorithms, how you implemented those algorithms, your assumptions, difficulties, insights, and results. Organize and present your document as if it were the only basis for your assignment's grade. The format of your writeup is up to you, but it should minimally include the following:

- Introduce the main goals of the project and in a couple of sentences summarize what you have accomplished.
- Briefly describe the mark and sweep algorithm you chose to implement. Motivate your selection.

- Present and explain graphs that study the number of times garbage collection is called based on the heap size.
- How did you test your garbage collection implementation?
- What problems did you encounter while implementing mark and sweep garbage collection? If you knew someone who was just about to start work on this assignment, what advice would you give them?
- What, if any, outside sources did you use (e.g., articles, books, other students)? This is particularly important. It's OK to look at books and articles and speak with your professor and fellow students (although sharing code is strictly forbidden), but as with any scientific document, you should always cite your references and collaborations. You can either cite collaborations in footnotes or in a separate Acknowledgment section.

There's no exact number of pages you should write, but if you've got between four and six then you're in the right ballpark.

6 Hints for doing the assignment

You should start this assignment early!! You only have three weeks to complete this assignment. I recommend spending a couple hours a day as soon as you finish project 1. Start writing your report as you are planning the implementation. A well-written report with some missing implementation guarantees a higher grade than a poorly written report with all the implementation.

I can look at your code and help point you in the right direction, but the amount of help I can give may be inversely proportional to the amount of time until the due date. By no means should you spend several hours trying to figure out a weird bug; consult me for help. When e-mailing me about the project, send a copy of the relevant section of your code (not as an attachment; send it as text pasted into your message). Give a good description of the problem including information about the stack in a debugger. Also, indicate possible solutions you have tried.

7 What to turn in

Turn in a hard copy of the report and email a copy in pdf format to mstrout@cs.colostate.edu.

Create a jar file that includes all of the bytecode files, the source files, your test cases, and your benchmarks. Send the jar file and a README file that gives specific command-lines for running the jar file on your provided test cases and benchmarks.

8 Due date

This assignment is due Tuesday September 29th, **at 2:00pm** and is worth 10% of your final grade. Late assignments will be penalized 20% per day.