

In this assignment you will implement garbage collection for the MiniJava compiler. This assignment can be done in groups of two or three. Larger groups will be expected to show more experimental results in terms of how their garbage collection is working.

## 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.

You will be implementing a Mark and Sweep collector. For the garbage collector to do its job, it needs to identify the root set and what fields in objects are pointers. The compiler generates a pointer map for each procedure to aid in identifying parameters and locals in the root set, and a descriptor to aid in identifying pointers in heap objects.

The pointer map structure for this project is as follows (also see GC-start.s and GarbageCollection.java):

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

The `gc_key` provides a way to identify what procedure with which each `ptrmap` is associated. Keep in mind that the GC works at runtime. You will be modifying the MiniJava compiler so that when a function is called, its first parameter (even before the implicit “this” parameter) is the `gc_key` for the callee. 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.

The class descriptor the MiniJava compiler must 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 is as follows (also see GC-start.s and GarbageCollection.java):

```
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 will 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 will return a pointer to the data segment for use in the mutator code. Only the GC code will know about the prepended fields.

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.

## 2 The Assignment

Your assignment is to extend the MiniJava compiler so that it generates record descriptors and pointer maps and to implement a *mark and sweep* garbage collector in C. The following interface should be implemented in C and compiled to MIPS using the Wisconsin C- compiler (cmm.jar will be sent by email, see <http://www.cs.wisc.edu/~lenz/compiler.html> for more info).

- **void init\_gc( struct ptrmap \* last, int heapsize )** will initialize all of the data structures needed by the GC. For example, there will be a global variable that maintains a pointer to the last ptrmap, and there should 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 )** will 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 will return a pointer to free space large enough for the requested object. The actual allocation will include three extra words that sit before the returned address. halloc\_gc will perform garbage collection if necessary.

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

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

The code generated by the MiniJava compiler (some of which is sucked in from a .s file generated by cmm) 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.

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

- The program should gracefully fail when an allocation is requested and there is no more allocatable space left.
- The `alloc_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 some test cases where garbage collection occurs at least once for the largest heap size: one test case if there is one person in your group, two if there are two people, etc. Explain the results.

### 3 Getting Started

1. Start with the MiniJava-start.tar file that is posted at google groups.
2. Modify the MiniJava compiler to generate the pointer maps, descriptors, passing of gc keys, calls to `halloc_gc()`, and calls to `init_gc`. Also have the MiniJava compiler read in the GC.s file, remove its empty main definition, and append the GS.s code to the generated MIPS file. Use the GarbageCollection.java (posted as a file on cs553fall07 google group) functionality and the examples in the And.java.s and GCtest1.java.s files (also posted) to aid with this process.

3. Play with the cmm compiler and look at the GC-start.s file (posted at google) to see example C- code. The Wisconsin C- compiler <http://www.cs.wisc.edu/~lenz/compiler.html> has been compiled on the CS linux machines. Copy the jar file from the following location:

```
/s/bach/b/class/cs553/GCTesting/cmm.jar
```

To compile to Spim-compatible MIPS use the following command:

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

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()`.

## 4 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 and working together 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. More pages are needed due to the graphs.

## 5 Hints for doing the assignment

**You should start this assignment early!!** You only have two 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.

### 5.1 Suggested Extensions to the MiniJava Compiler

- Create a `GarbageCollection` instance in `CodeGenAssem.java` and pass it to the visitors and other objects that need it.
- Call `GarbageCollection.funcCallGCKey()` in the order the procedures are defined to ensure that `main` is assigned the `gc_key` zero. `BuildSymTable` or `Translate` could make the call as each method is visited.
- The `main` routine is the only one that is not called by code generated by the MiniJava compiler, but when it calls `halloc_gc` it is still expected that the `gc` key will be at `0($fp)`. Therefore, the prologue for `main` must push its own `gc` key on the stack before pushing the `ra` and `fp` registers in `MipsFrame.procEntryExit3()`. `MipsFrame.procEntryExit3()` can assume the `gcKey` for `main` is zero.
- Also modify `Mips.procEntryExit3()` so that it generates a call to `init_gc()` at the top of `main`.
- Modify `Translate.outACallExp()` to create a `Tree.Exp` for the garbage collection key and make the key the parameter that comes before the implicit “this” parameter. The `gc` key will be a constant and again can be retrieved by using `GarbageCollection.funcCallGCKey()`.
- All calls to `halloc` (except for arrays) will need to be replaced by calls to `halloc_gc`. Modify `outANewExp` in `ast_walkers/Translate.java` so that they use `GarbageCollection.heapAllocCall()`.
- Extend `SymTable/ClassSTE` and `SymTable/MethodSTE` with a method `getOrderedVarList()` that generates a list of the variables declared in that scope, in the order they were declared. Achieving this will probably mean extending the `SymTable.insert()` method so that it maintains an ordered list and has a method to access that. This information will be used to generate the class descriptors and method pointer maps.
- In `Mips/Codegen.java`, `munchExpNAME()` needs implemented so that MIPS code that loads the address of the label is generated. NOTE: Its implementation will be very similar to `munchExpCONST`.

```
la $t0, Label
```

Also, `munchExp()` should be modified to call `munchExpNAME()` when appropriate. Notice that both the `init_gc()` and `halloc_gc()` functions take a label as a parameter. Specifically, `init_gc()` will want the label for the last pointer map for its first parameter, and `halloc_gc()` requires the label for the class descriptor of what is being allocated.

- Pass in `linesToNodes` to `Translate` so that can pass line number of allocation to `halloc_gc`. Will be very helpful for debugging the GC. `Translate` will also need a `GarbageCollection` objection.
- Have `Mips/MipsFrame.programTail()` call `GarbageCollection.mips_data()` and `gc.implement()`, and append the string to the returned program tail.

## 5.2 Working with the Wisconsin C- Compiler

- Read their documentation. There isn't much, but it will help your understanding.
- For some reason their compiler allocates 16 bytes per field in a class, but accesses fields at 4 byte boundaries. This should not cause any problems, since the pointer map structure instances will be generated by the MiniJava compiler and read by functions generated by the `cmm` compiler.
- The `cmm` compiler will only compile a program that has a main. Therefore, the `GarbageCollection.java` file provides functionality to strip out the main implementation from GC.s.

## 6 What to turn in

Turn in a hard copy of the report and email a copy in pdf format to [mstrout@cs.colostate.edu](mailto: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.

## 7 Due date

This assignment is due Friday September 14th, at **2:00pm** and is worth 10% of your final grade. Late assignments will be penalized 20% per day.