A garbage-collected C-backend from scratch

Compiler Construction 2020 Final Report

Axel Lindeberg
KTH Royal Institute of Technology
alindeb@kth.se

1. Introduction

I chose to implement one of the suggested extensions, a C-backend as an alternative to the JVM backend implemented in lab 6. Since the `punkt0` relies on the garbage collected environment of the JVM and has features not present in plain C, such as classes and inheritance, this is not a trivial thing to implement. I also decided to implement the garbage collection myself, as that seemed like a fun task, instead of relying on an external library.

To implement this, an optional flag (`--cbackend`) was added to the compiler. If supplied, the compiler will output a C file instead of the JVM class-files. This file can then be compiled by a C compiler to get a native binary, equivalent to the `punkt0` program. There are several reasons why you might want to do this:

- Your compiler now supports any target that the C compiler supports, which for GCC is more or less any architecture you can think of!
- You do not have to implement sophisticated machine-code optimization and can instead rely on the decades of work done to implement that in the C compiler.
- Outputting C code is arguably a lot easier to implement than a machine-code backend. Often there is an easier mapping from your language to C since it is at a much higher level. This can get you started thinking about and implementing more high-level features of your compiler.

This was implemented in two separate parts. One was a stand-alone garbage collector written from scratch in C, and the other was a new phase of the `punkt0` compiler. The new phase has the same interface as the `CodeGeneration.scala` phase that outputs JVM bytecode and it operates a lot like the pretty-printer we implemented for lab 3. More on this in section 3.

2. Examples

This project introduces no new language feature changes that could easily be shown of here. Instead, I will show an example of what the translation to C code would look like. Consider the following `punkt0` code which uses some "advanced" features like if-expressions which are not present in C. More on this in section 3, but it requires some relatively complex translation.

```
object Main extends App {
  var x: Int = 10;
  var y: Int = 50;
  var z: Int = 0;

  z = if (x < y) y else x;
  println(z)
}
```

This gets translated into the following C code:

```
void __attribute__((noinline)) p0_main() {
  int x = 10;
  int y = 50;
  int z = 0;
  z = ({
    int p0_if_res;
    if ((x < y)) {
      p0_if_res = y;
    } else {
      p0_if_res = x;
    }
    p0_if_res;
  });
  printf("%d\n", z);
}

int main() {
  u8 dummy;
  gc_init(&dummy);
  p0_main();
}
```

Another complexity (again more on the details in section 3) is handling inheritance. The following example shows how inheritance is implemented in the C translation, using pointer casting and virtual method tables (vtables). Consider the following trivial `punkt0` example which shows of inheritance and overridden functions:

```
class A {
  var x: Int = 42;
  def fn(): Int = {x}
}

class B extends A {
  override def fn(): Int = {1337}
}

object Main extends App {
  var a: A = new A();
  var b: A = new B();
  println(a.fn());
  println(b.fn())
}
```

What follows is the translated code of the example above. It shows how much code is actually needed to get inheritance working. In C there is no such concept, so dynamic dispatch, the ability to dynamically type variables at runtime, has to be implemented from scratch. Note that some details have been removed for brevity. I would recommend the reader to compile the `QuickSort.p0` in order to see a more complex example of inheritance and function overriding in use.

```
/*------ vtables ------*/
void *p0_A__vtable[] = { NULL };
void *p0_B__vtable[] = { p0_B_fn, NULL };

/*------ struct definitions ------*/
struct p0_A {
  void **vtable;
  int x;
};
struct p0_B {
  struct p0_A parent;
};

/*------ A member functions ------*/
void p0_A__init(struct p0_A *this) {
  *((void***)this) = p0_A__vtable;
  ((struct p0_A*)this)->x = 42;
}

struct p0_A* p0_A__new() {
  struct p0_A* this = gc_malloc(sizeof(struct p0_A));
  p0_A__init(this);
  return this;
}

int p0_A_fn(struct p0_A *this) {
  void *_override_ptr = (*(void***)this)[0];
  if (_override_ptr != NULL)
    return ((int (*)())_override_ptr)(this);
  gc_collect();
  return ((struct p0_A*)this)->x;
}

/*------ B member functions ------*/
void p0_B__init(struct p0_B *this) {
  p0_A__init(this);
  *((void***)this) = p0_B__vtable;
}

struct p0_B* p0_B__new() {
  struct p0_B* this = gc_malloc(sizeof(struct p0_B));
  p0_B__init(this);
  return this;
}

int p0_B_fn(struct p0_B *this) {
  void *_override_ptr = (*(void***)this)[1];
  if (_override_ptr != NULL)
    return ((int (*)())_override_ptr)(this);
  gc_collect();
  return 1337;
}

/*------ punkt0 main function ------*/
void __attribute__((noinline)) p0_main() {
  struct p0_A* a = p0_A__new();
  struct p0_A* b = p0_B__new();
  printf("%d\n", p0_A_fn(a));
  printf("%d\n", p0_A_fn(b));
}
```

Note in particular the vtable of class B. At index zero we see the function pointer to `p0_b_fn`. In the function `p0_A_fn`, which is A's version of the `fn` function, you can see a check to look in the vtable if this entry is non-null and thus the function has been overridden. If that's the case the value is cast to the appropriate function pointer and called instead of A's version. The B instance has it's vtable pointer set to this vtable, in `p0_B__init`. This means that, while the C type is of `struct A`, the virtual table tells the program that it should actually use the B version of the `fn` function. This program correctly prints out 42 and then 1337.

3. Implementation

This project was very implementation heavy and low level. It required getting a key understanding of C, how pointer-casting works in C, and virtual method tables. This section details the implementation of this project, which had two major parts: the tracing garbage collector implemented in C from scratch, and the compiler C-backend outputting C code which uses the garbage collector.

3.1 Theoretical Background

As stated, this project was more implementation heavy and quite low level, rather than theoretical but it relies on a few established concepts that I had to research.

3.1.1 Garbage collection strategies

There are two major ways garbage collection is implemented. The first one, and perhaps most common, is a tracing garbage collector. In this strategy you have "GC pauses" where the garbage collector runs and tries to establish which part of the heap is currently used and which can be freed. This typically requires a runtime, like the JVM. In a tracing garbage collector the GC will scan the stack, global variables, and so on for references to GC-allocated memory. These are called the roots of the reference tree. Each reference may itself contain references to other instances, this is what forms the tree structure. While visiting each node in the tree the GC will mark these as used, and will then discard all memory that was not marked. This works because the memory that wasn't marked is fundamentally unreachable by the program and can thus safely be reclaimed. This strategy, as well as other much more sophisticated techniques is used in the JVM, and the V8 JS engine for example [Degenbaev et al. 2018].

The other major strategy is reference counting. In this strategy each instance of an object has a count of how many references exist to the instance. Every time such a reference is created the counter is incremented and everything it goes out of scope it is decremented. When it reaches zero you know there are no more references to the instance and can thus safely free the memory. A problem with this approach you need to solve is cyclic references which will never reach zero. This strategy is used in the Swift language for example [Apple Inc. 2020].

For this project I chose the first approach, a tracing garbage collector.

3.1.2 The x86_64-architecture

In the introduction I argued that one benefit of a C-backend is being able to target different architectures. However, my garbage collector implementation is unfortunately x86_64-specific.

As we know, a program in the x86_64 architecture is logically divided into segments: the stack, heap, code, and data segments. The `punkt0` language has no pointers or pointer arithmetic, as such all pointers to class instances can be found on the stack. To find all root nodes of the reference tree when garbage collecting you therefore only have to look at the stack. This was my initial thought, however, it turned out to not be that simple. Additionally, pointers to class instances can be temporarily held in any of the CPU's general purpose register. The x86_64 CPU has the following 12: rax, rbx, rcx, rdx, $r8$, $r9$, $r10$, $r11$, $r12$, $r13$, $r14$, $r15$. Note that the additional registers, such as rsp, rip, have specific purposes and cannot store program variables. More details on this in section 3.2.1.

3.1.3 Virtual method tables

When programming in a relatively high-level language like Java you do not have to think about a lot of the implementation details, but how does the JVM for example know what virtual function to call? If a function takes in a class A and you give it a class B which inherits from A the JVM will correctly invoke B's version of an overridden method, despite none of that information being visible in the function body.

In C there is no such thing as inheritance so this behavior had to be implemented. This is usually done through *Virtual method tables* or vtables. On a low level, a vtable is just an array of function pointers. When you dereference a class to call a method, instead of jumping to a fixed address where the function is stored, the address is looked up in the vtable. To know which vtable to use each instance of a class has to store a pointer to it's vtable [Bounov et al. 2016]. This dynamic dispatch leads to both memory and runtime overhead.

3.2 Implementation Details

What follows are two sections detailing the work required to implement the two major parts of this project: a tracing garbage collector implemented in C from scratch, and the compiler C-backend outputting C code which uses the garbage collector.

3.2.1 Garbage collector in C

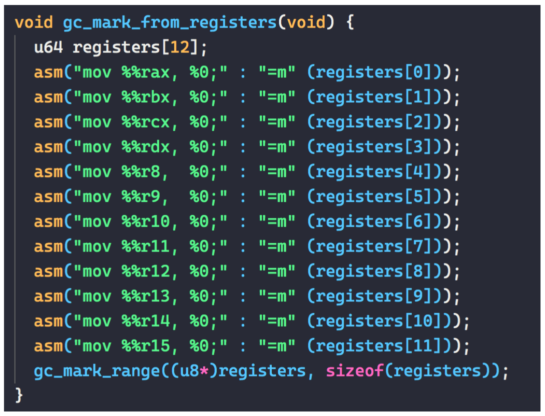
For this project I implemented a stand-alone garbage collector in C from scratch (see `./c-backend/gc.h` in my repository). It is an arena-style garbage collector. On start-up it allocates a large area of memory from malloc and via it's on allocation call hands out pointers to bits of this memory. It internally keeps track of which bytes of that memory-arena are free and which are occupied.

To keep track of which bytes are free and which are occupied the GC uses a bitmap where the n:th bit indicates whether or not the n:th *byte* of the memory-arena is taken. This requires an extra eighth of the total memory area for this bookkeeping and is not the most efficient approach (see more in section 4) but it was easy to implement for this proof of concept. To allocate a chunk of memory the GC scans this bitmap for a continuous number of zero bits that can fit the number of requested bytes. This requires some carefully thought-out bitwise operations but was overall not too difficult to implement. On allocation the GC prepends a header for it's own bookkeeping. This is required to keep track of how large the allocation is for example.

The GC I implemented is a tracing garbage collector, meaning at each collection phase it looks at all the nodes in the reference tree to see which allocated instances are still reachable by the program, and thus should not be freed. In punkt0 this is relatively straight forward. The roots of the reference tree can only be found on the stack, since `punkt0` does not allow any sort of pointer arithmetic, is type safe, and does not have any global variables. To find these references the GC scans the entire stack looking for values that ”look like” a GC allocated pointer. This is achieved by, at initialization, storing the address of a local variable to get the address of the top of the stack, and then scanning all values from the current stack pointer up to that address.

An obvious problem here is determining what values on the stack actually are GC-heap pointers versus regular values, like an `int`, that just happen to hold a value that looks like such a pointer. While this may not be a common occurrence, I chose to try and handle this edge case with a canary strategy. In the prepended header, the GC stores a random value, called `GC_CANARY`. When trying to determine whether or not a value on the stack is a GC-heap pointer, we compare the canary value in the header to this static value. If they are not equal, we can be sure that this was a false-positive. This reduces the probability of false-positives when tracing by a lot at the cost of a few extra bytes per allocation. Not only would a random value on the stack have to match a pointer to the memory area, a specific value would have to exist in the memory arena at a specific offset from this value. In the worst case, if this should happen, the GC will erroneously mark a value as allocated when it should not be, which only has the consequence of temporarily leaking the bytes. In regular execution this should have a next to zero probability of occurring. You could reduce the probability even more by introduction a larger canary, at the cost of additional wasted bytes for each allocation.

An additional complexity I did not foresee was that my initial assumption, that all references can be found on the stack, does not hold when compiling with optimizations enabled, i.e with `-O2/3`! When compiling with these flags my C programs, that initially worked, started to seg fault. After a lot of debugging, even pulling in *gdb* and stepping through the program, I realized the problem. When compiling with optimizations the compiler will, whenever it can, store a variable in a register and completely skip allocating it on the stack. This meant my tracing GC did not see the values and erroneously freed them. Fixing this problem required some inline assembly, where I read the value from all general purpose registers in the CPU and scan them for GC-heap pointers (see figure 1). This is why my GC unfortunately is not portable to other architectures, although this function could be reimplemented for another architecture.



```c
void gc_mark_from_registers(void) {
  u64 registers[12];
  asm("mov %%rax, %0;" : "=m" (registers[0]));
  asm("mov %%rbx, %0;" : "=m" (registers[1]));
  asm("mov %%rcx, %0;" : "=m" (registers[2]));
  asm("mov %%rdx, %0;" : "=m" (registers[3]));
  asm("mov %%r8,  %0;" : "=m" (registers[4]));
  asm("mov %%r9,  %0;" : "=m" (registers[5]));
  asm("mov %%r10, %0;" : "=m" (registers[6]));
  asm("mov %%r11, %0;" : "=m" (registers[7]));
  asm("mov %%r12, %0;" : "=m" (registers[8]));
  asm("mov %%r13, %0;" : "=m" (registers[9]));
  asm("mov %%r14, %0;" : "=m" (registers[10]));
  asm("mov %%r15, %0;" : "=m" (registers[11]));
  gc_mark_range((u8*)registers, sizeof(registers));
}
```

Figure 1: Collecting root references from the CPU registers

To summarize, in each garbage collection the GC does the following:

1. Completely clear the memory bitmap, i.e mark everything in the area as *free*.
2. Using inline assembly, store all values from the 12 general purpose registers of the CPU into an array. Scan that array for root references to GC-allocated objects.
3. Scan the entire stack for root references to GC-allocated objects.
4. Recursively, mark all found pointers and scan the memory of the allocation for references to additional GC-allocated instances. We know how large the allocated pointer is by looking at the prepended GC header.

The last step is very important. An instance of a `punkt0` class can have other class instances as fields, which will themselves also be GC-heap allocated. Thus they also need to be marked as allocated if their parent is still reachable. Another problem I encountered here was cyclical references. If you are not careful, the tracing algorithm will end up in an infinite loop if you blindly follow references. I handle this by checking if the memory has already been marked, before following it's references.

3.2.2 The C code generator

The C code generator (found in `CBackend.scala` in the repository) works a lot like the pretty-printer we implemented in lab 3. For each type of AST node it outputs a string with equivalent C code. I will not explain the implementation of that in detail here as that was largely the same as the lab. However, there were a few things that made translating `punkt0` to C a bit complicated considering the features of the languages and how the C language works. I will cover those now.

The major problem to solve was how to handle inheritance and overridden methods, which required some substantial thought and work. The most important thing that made this possible is the following line from the C standard:

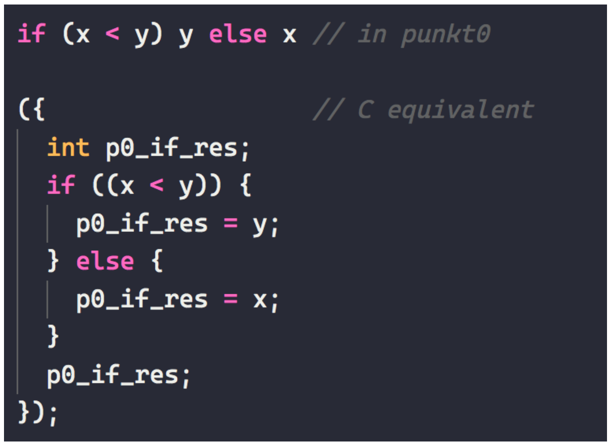
> *A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.*
>
> [ISO 2018], page 82

What this says that it is always valid to cast a pointer to *a struct* to a pointer to *the first field of that struct*. If a class B inherits from A we can then store a `struct A` as the first field, which makes casting a B pointer to an A pointer valid, guaranteed by the official ISO language specification [ISO 2018]. This is used in two places in the C backend. One is for member functions, where we simply pass in the object pointer and let C automatically downcast it to a pointer of the parent class. The other time is when accessing fields of the parent class, where we first cast to the parent type, and dereference to access the field.

Another aspect of inheritance is overriding methods. To achieve this in C I implemented a virtual method table for each class. This is essentially just a static array of `void*`, where each entry is a function pointer to the overridden function. Each member function gets assigned an index in this table and before executing it we check if this entry is non-null and if so jump to the stored function pointer instead. By storing a pointer to this table only in the uppermost parent class (i.e classes without any parent) we can again utilize the part of the C standard quoted above, to safely cast any class pointer to a vtable pointer. Also, by only storing this in the root parent class we only pay the cost of one pointer per instance, in terms of memory overhead.

If- and block-expressions are something that `punkt0` has that does not trivially map to C. To support these, I relied on a very common C language extension, which both GCC and Clang has supported for many years, called *statement expressions*. It turns a block in C into an expression, returning the last statement in the block [Free Software Foundation 2020]. For block expressions, this maps neatly which made it quite easy to implement. If-expressions on the other hand required some additional thought. I chose to implement them by turning the statement into a block expression, adding a temporary variable, letting each arm of the if assign to this variable, and return that variable from the block. See figure 2 for an example.

Furthermore, identifier-clashes is also another thing I had to consider. What happens if a `punkt0` class is called *garbage_collector* when that is already defined as a struct in the GC implementation? It would not



```
if (x < y) y else x // in punkt0

({                   // C equivalent
  int p0_if_res;
  if ((x < y)) {
    p0_if_res = y;
  } else {
    p0_if_res = x;
  }
  p0_if_res;
});
```

Figure 2: Example of how if-expressions are transformed into C

compile. To work around this I added a prefix (`p0_`) to any identifier originating from `punkt0`, and made sure not to use that prefix in my GC implementation.

Lastly, string handling and the overloaded addition operator had to be implemented. This was done by adding three helper functions for adding strings and ints. These allocate the required memory from the garbage collector and use `sprintf` to write into that memory buffer. The C code generator then inserts a call to these functions whenever the overloaded `+` is used.

4. Possible Extensions

There are so many things you could do to continue the work of this project. An obvious one I did not have time to look at is a better, more sophisticated malloc implementation in the GC. My bitmap implementation for keeping track of allocated/free bytes is simple but very inefficient. To benchmark my GC I created a simple linked list program which does 100 add, contains, and remove operations on a linked list. When compiled with `-O3` this successfully executes in 0.86 seconds, while the JVM backend does so in 0.03 (which includes the relatively slow JVM start-up time). The JVM has also had decades of work put into it's garbage collector, and entire PhD's written about how to make it performant. It is still very much an open problem of how to do concurrent garbage collection effectively. So, my approach to the tracing GC could be extended and improved a lot, for example by using multiple threads and work-stealing queues to mark the heap in parallel.

An additional performance drawback to my approach is to figure out when to actually collect garbage. For this proof of concept I simply added a call to `gc_collect()` at the end of every function call. This is needlessly aggressive. An interesting extension would be how to insert garbage collection in a more sophisticated way. One idea I had was to start a timer in a separate thread, to garbage collect at a specific interval. That however leads to questions of how to stop the main thread from allocating and so on. Another idea I had was to use a background thread which sends a signal to the program at an interval, and collecting in the signal handler. Once again, you would have to think carefully about what to do if the main thread for example is in the middle of allocating.

Lastly, one thing I did not have time to implement was increasing the garbage collectors internal memory, to support programs that require arbitrary amounts of memory. Currently, the GC allocated 2 MB on start-up and if it runs out of memory it crashes. This could quite easily be solved by allocating a new memory area whenever the first one does not fit the allocation.

References

Apple Inc. Automatic reference counting. 2020. URL https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html. Visited 2020-12-20.

D. Bounov, R. G. Kici, and S. Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *NDSS*, 2016.

U. Degenbaev, M. Lippautz, and H. Payer. Concurrent marking in v8. 2018. URL https://v8.dev/blog/concurrent-marking. Visited 2020-12-20.

Free Software Foundation. Statements and declarations in expressions. 2020. URL https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html. Visited 2020-12-20.

ISO. *ISO/IEC 9899:2017: Programming languages — C*. 2018. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.