Compiler Construction: Supervision 3

Daniel Chatfield

March 9, 2015

1. (a) Describe what calling conventions are and what they are used for. Illustrate your answer using an example in pseudo assembly code. [4]

A calling convention is a scheme for how parameters are passes to a function from the caller and how they return a result.

Different calling conventions may differ in:

- Where parameters, return values and return addresses are placed (registers, call stack etc.)
- Order in which arguments are passed
- How a return value is returned from the callee to the caller
- Whether the caller or callee is responsible for cleaning up and setting up a stack frame.

C uses the CDECL calling convention (right to left and the calling function cleans the stack).

Consider the following function:

```
int example(int a, int b) {
  return a + b;
}
```

called by:

```
x = example(2, 3)
```

These would produce the following assembly listings repsectively:

```
example:
  push ebp
  mov ebp, esp
  mov eax, [ebp + 8]
  mov edx, [ebp + 12]
  add eax, edx
  pop ebp
  ret
```

```
push 3
push 2
call example
add esp, 8
```

On entry to example, *esp* points to the return address pushed onto the stack by the call instruction.

(b) Explain the difference between calling conventions and evaluation strategies. [2]

Evaluation strategy is seen as part of the language where as calling convention is seen as implementation details. The evaluation strategy may influence or even dictate some parts of the calling convention but they are considered separate.

Evaluation strategy determines when and in what order to evaluate the arguments of a function call and what kind of value to pass.

(c) A compiler author is writing a code generator which targets the C language. His intention is to implement CPS so that each function from his source language is translated into a C function which pops its continuation off an argument stack and then calls the continuation at the end. For "convenience", he defines the following macro: [2]

```
#define JUMP(x) (*x)()
```

He then uses the macro to jump to continuations:

```
void foo() {
    void* cont = *Stack--; // pop continuation off the stack
    // code for foo
    JUMP(&cont); // cont is the continuation
}
```

Explain potential pitfalls with this implementation of CPS.

Each continutation is called from within the callee's scope and thus the stack will grow and grow as the program executes. Eventually stack overflow will occur.

(d) The compiler author reconsiders his approach in light of your answer to the previous question and changes the signatures of all C functions to e.g.: [2]

```
void* foo();
```

He also changes his macro to:

```
#define JUMP(x) return(x)
```

Finally, he implements the following:

```
int main(int argc, char** argv) {
    void* cont = &foo;
    while(1) { cont = (*cont)(); }
    return 0;
}
```

Explain why this approach is better.

> The continuations are popped off the stack and executed by a while loop and therefore since a continuation returns before the next one is executed, its stack frame is removed.

(e) If the compiler author were to choose an assembly language as the target language for his compiler, how should he implement CPS then? [2]

> Every subroutine returns the address of its continuation with a main loop that just calls the return address (as in the C example).

2. Recall that configurations in the STG-machine are 6-tuples consisting of:
   1. the code to be executed
   2. the argument stack (as), which contains values
   3. the return stack (rs), which contains continuations
   4. the update stack (us), which contains update frames
   5. the heap (h), which contains closures
   6. the global environment (σ), which gives the addresses of all closures defined at top level

   How do we map these to a C program? We begin by looking at the heap. For this purpose, we will distinguish between two kinds of objects on the heap:

   - *Head normal forms* or *values* are heap objects which can not be evaluated any further, and
   - *suspensions* or *thunks* are heap objects which have not yet been evaluated

   Collectively, we will refer to values and thunks as *closures*. Traditionally, a closure consists of a pointer to the code of a suspended computation (i.e. just a pointer to some code), and the values of all free variables. For example, think about the Further Java exercises in which you created anonymous

methods and classes. Those resulted in the creation of closures which contained copies of the variables that were captured by the anonymous methods or classes.

In the STG-machine we use this representation for both, values and thunks. This corresponds to the way we represent closures in the operational semantics. I.e. the $Eval e \rho$ code component consists of an expression e (the code) and the local environment ρ (the free variables).

(a) Show how you would represent STG closures in a low-level systems language, such as C, and explain your answer. [2]

I would represent them as a pointer to a code section and a pointer to an array of free variables.

Since free variables are themselves closures, this will be an array of pointers to closures.

When the code section is entered it sets the code pointer to a black-hole pointer that if executed causes an error (since if this happens then the value must depend on itself which is not valid).

When the code section has evaluated, it overwrites itself with the resulting data.

(b) It is possible to generate static closures for all global bindings. Generate closures and function prototypes (stubs) in C for all global bindings in the following STG program. [4]

```
nil  = {} \n {}  -> Nil {}
succ = {} \n {x} -> +# {x, 1#}
list = {} \n {}  -> Cons {5#, nil}
map  = {} \n {f,xs} ->
          case xs {} of
            Nil {}      -> Nil {}
            Cons {y,ys} -> let  fy = {f,y}  \n {} -> f {y}
                                mfy = {f,ys} \n {} -> map {f,ys}
                           in Cons {fy, mfy}
main = {} \n {} -> map {succ, list}
```

For example, the C function corresponding to `nil` may look like this:

```
void* nil_code() {
  // do nothing yet
  JUMP(NULL); // to make the compiler happy for now
}
```

```
void* succ_code() {
  // code
  JUMP(*stack--);
}
```

```
void* list_code() {
  // code
  JUMP(*stack--);
}

void* map_code() {
  // code
  JUMP(*stack--);
}

void* main_code() {
  //code
  JUMP(*stack--);
}

Closure succ_closure = {
  &succ_code,
  {}
};

Closure list_closure = {
  &list_code,
  {}
};

Closure map_closure = {
  &map_code,
  {}
};

Closure main_closure = {
  &main_code,
  {}
};
```

(c) Explain why there is no need to implement the global environment σ when mapping the STG machine to C. [2]

When mapping to C you can simply use the C global environment and let C handle it.

(d) Closures for local bindings must be allocated dynamically at runtime because we cannot predict what their free variables will be. For this purpose, we require a heap that we can allocate memory on. Do not worry about freeing memory. Show how you would implement this in C. [1]

I'm not comfortable enough with C for a full working example.

Use malloc to allocate space on the heap. In the previous question I used a pointer to an array as I wasn't sure if you could do it otherwise since the struct needs to have a fixed size, I saw something about a flexible array member but didn't really get it.

So you would create a new struct then use malloc to allocate space for the pointer array.

(e) How would you implement a test for whether there is free space on the heap? [1]

Malloc will return a `NULL` if there is insufficient space.

(f) Suppose that a local binding fy has a corresponding C function named fy_code which captures two free variables. Show how you would allocate space on the heap for a corresponding closure and how you would initialise it, assuming that pointers to the closures for the two free variables are available in local variables named f and y. [3]

```
Closure* closure;
closure->code = &fy_code;

// needs to be on heap - not sure

Closure *closures[2];

closures[0] = f;
closures[1] = y;

return closure;
```

3. (a) It is possible to implement all three stacks (*as*, *rs*, and *us*) using one concrete stack and, for simplicity, that's what we will do for now. Traditionally, stacks grow from higher memory addresses to lower memory addresses. Why?

The stack and the heap need to be able to grow dynamically and therefore the most flexible layout is for one to grow down from a high address and the other to grow up from the bottom.

As to why it is heap up, stack down I can't think of any solid reasons for this - it seems to just be convention.

(b) Show how you would implement a stack in C, including sample code to push and pop items.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
typedef struct Stack {
    int arr[100];
    int pos;
} Stack;


void push(Stack* stack, int val) {
    stack->arr[stack->pos] = val;
    (stack->pos)++;
}

int pop(Stack* stack) {
    if(stack->pos == 0) {
        fprintf(stderr,"Invalid input\n");
        exit(1);
    }
    return stack->arr[--(stack->pos)];
}
```

(c) The process of evaluating ("entering") a closure involves jumping to the memory address indicated by the code pointer of a closure and pushing any arguments onto the stack. How would you ensure that the code component of the closure can find its free variables? How would you address the argument?

When you enter a closure you first push a pointer to it into a register (lets call it the closure$_{r}egister$).$Thenthecodecomponentcanjustlookatthisregister$.

(d) Explain what we mean by "boxed" and "unboxed" values. Give an example of both in the STG language.

An unboxed value is a primitive where as a boxed value is a minimal wrapper around a primitive type.

Closures are an example of boxed, I'm not sure about unboxed.

(e)

```c
void* zero() {
  int_res = 0;
  JUMP(*stack--);
}

void* not() {
  if(int_res == 0) {
    int_res = 1;
  } else {
    int_res = 0;
  }

  JUMP(*stack--);
```

```
}

void main() {
  PUSH(not);
  zero();
}
```

(f) *Not sure*

4. (a) STG (as described in the original paper) uses the Copying Collection technique for garbage collection. Briefly explain how this works in general. [4]

With copy collection you have two heaps, when the garbage collector runs it copies traverses the heap from the roots and copies across everything that is accessible into the other heap.

(b) Copying Collection requires two heaps. How would you implement them space-efficiently? [2]

I would use a two way heap (one that goes bottom up and one that goes top down), that way when you copy the objects across you can fill in the gaps left.

(c) Copying memory from one heap to another is a slow process. Explain one garbage collection technique which could reduce the amount of memory that has to be copied. [2]

You can have different heaps with different GC policies. An object that has survived several GC runs could be moved to a heap with a less aggressive GC policy.