1 2018 Paper 4 Question 4

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2018p4q4.pdf

Suppose that we are to implement a compiler for the following simple, strongly-typed language with types t, expressions e, and programs p.

$$
\begin{array}{llr}
t ::= & \textbf{int} & \\
 & \mid t * t & \text{(product type)} \\
 & & \\
e ::= & n & \text{(integer)} \\
 & \mid ? & \text{(read integer input by user)} \\
 & \mid e + e & \text{(addition)} \\
 & \mid e - e & \text{(subtraction)} \\
 & \mid (e, e) & \text{(pair)} \\
 & \mid \textbf{fst}\ e & \text{(first projection)} \\
 & \mid \textbf{snd}\ e & \text{(second projection)} \\
 & \mid f(e) & \text{(function binding)} \\
 & \mid \textbf{let}\ x : t = e\ \textbf{in}\ e\ \textbf{end} & \text{(let binding)} \\
 & & \\
p ::= & e & \\
 & \mid \textbf{fun}\ f(x : t) : t = e; p & \text{(function definition, recursion allowed)}
\end{array}
$$

In the above x and f range over identifiers. For example, here is a simple program:

```
fun swap (p: int * int) : int * int = (snd p, fst p);

fun swizzle (p: int * (int * int)) : (int * int) * int = (swap (snd p), fst p) ;

swizzle (?, (?, ?))
```

You are asked to implement this language on a stack machine **that has no heap**. All stack entries are simple words (integers or pointers). Hint: consider using type information.

(a) Describe how your compiler will use the stack to implement function calls and returns. Describe any auxiliary pointers that you might need. Is there anything about the language above that makes this especially easy?

Since elements on the stack are integers or pointers, I assume the language has linear instructions.

We need three additional pointers: a stack pointer, a frame pointer and a code pointer.

The stack pointer points to the next free space on the stack. The frame pointer points to the bottom of the frame – so we can access data relative to the frame pointer and the code pointer points to the instruction currently being executed.

Since this language is strongly-typed where all types can be determined at compile-time, we can perform static type checking and remove type information before runtime. So we compile code using type information and produce machine code which uses only binary information with offsets – no type information (I also assume we are not generating an object file).

To avoid verbosity, let "push onto the stack" also mean "increment the stack pointer".

On a function call, push the frame pointer onto the stack. Next, increment the current code pointer and push it onto the stack. This is the return address. Then set the code pointer to the address of the code for the function. Set the frame pointer to the value of the stack pointer. Next evaluate each of the arguments to the function.

The function must also got to take a closure, such that all free variables take the correct value! A simple way to do this would be using static analysis and add all free variables as arguments before the function is called.

This language is especially easy since types are decidable at compile time – for a more complicated language like OCaml, types can be made at runtime and so we must keep RTTI.

This langauge also has no exceptions and so we do not need an exception pointer.

(b) Describe how you allocate space on the stack for a value of type t.

Firstly, find the size of an instance of type t. Let this be s bytes. Copy the value of type t onto the stack, starting at the stack pointer (which points to the next free location in memory) and working upwards. Then increase the stack pointer by s.

(c) Describe how your compiler will implement expressions of the form (e_1, e_2). Explain how the order of evaluation (left-to-right or right-to-left) impacts your choices.

The code to create the expression is as follows:

```
<code for e1>
<code for e2>
```

This firstly evaluates e_1, then e_2. When e_1 finishes executing, the value it reduces to v_1 is left on the stack. When e_2 finishes executing, the value it reduces to v_2 is left on the stack. After this code is executed, the stack is of the form `[...; v1; v2]`. Since we have statically determined type information, we don't need a `MAKE_PAIR` operation – the next code will reinterpret the bytes as a pair.

The method above performs left-to-right evaluation. If we wanted right-to-left evaluation, we would swap the order in which e_1 and e_2 occur in the code.

(d) Describe how your compiler will implement expressions of the form **fst** e and **snd** e.

Sample code for **fst** e is given below:

```
<code for e>
pop(sizeof(e[1]))
// note sizeof(e[1]) is a statically known integer
```

Sample code for **snd** e is given below:

```
<code for e[0]>
pop(sizeof(e[0]))
// note sizeof(e[0]) is a statically known integer
<code for e[1]>
```

2 2000 Paper 3 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2000p3q3.pdf

With reference to a strictly-typed block-structured programming language, write brief notes on the following topics.

I made an assumption: We didn't learn about compilers in relation to any "strictly-typed block-structured programming language". I will therefore discuss topics in relation to *an arbitrary* block-structured programming language.

(a) The allocation and recovery of records stored in a heap

Allocating onto the heap is easy. The program will make a syscall to the OS for memory of the required size. The OS will then allocate memory and return a pointer to this memory.

Recovery of records stored on the heap is more problematic – it's not always obvious what can and cannot be freed. We want to free an object once it will never be used again. However, this is undecidable so instead we free objects once it's not *possible* for them to be used again.

Consider memory on the heap to be a graph. The root set is the set of variables which are pointed to by variables on the stack or in registers. Memory which can be reached by some path from the root set is known as reachable. We will garbage collect memory which is unreachable.

Note that we must be able to distinguish between pointers and integers. There are many solutions, the easiest is to use 63 (or 31) bit integers, with the first bit used to mark whether or not the data is a pointer.

The simplest garbage collection method is "mark and sweep" – perform a depth-first traversal of the object graph and mark all nodes which are reachable. Then iterate through all allocated memory, free objects which were not marked and un-mark all marked objects. This guarantees all unreachable objects are freed and all reachable objects are kept. However, this means the size of each object must be increased such that it's possible to mark them. Additionally, iterating through the heap is expensive leading to multi-second pauses in some cases.

Copy Collection resolves these issues by focusing instead on the live objects. Copy Collection splits the heap into two – the from-heap and the to-heap. Initially, the from-heap contains all objects and the to-heap is empty. Next, traverse the object graph starting at the root and copy every object encountered into the to-heap (overwriting it with a pointer to its new location) and ensuring to keep all pointers consistent during the copying process. This ensures that all live objects are in the to-heap. The entire from-heap can now be deallocated. This is faster than Mark-and-Sweep in general; but has the downside of using twice as much memory as the program strictly needs.

The final strategy is Generational Memory Management; a strategy which takes advantage of the statistical properties of memory allocation. Most objects are deallocated when they are very new (typically 98% of data collected in a garbage collection was allocated since the last garbage collection). Generational Memory Management splits the heap in two: a major heap and a minor heap. New data is allocated onto the minor heap. Whenever the program needs to free memory, it performs Copy-Collection on the minor heap (copying into the major heap). Note that the root set for recently-allocated objects should include objects which are pointed to by old objects. The major heap is collected using either copy-collection or mark-and-sweep intermittently.

We can use other heuristics (such as reference counting) to reduce the frequency which garbage collection must be carried out. Reference counting cannot (should not) be used in isolation since it never collects cyclic data structures. Reference counting also means pointer copies require pointer dereferences – which are slow.

(b) The implementation of variables of union type

A Union type is a variable which is either an instance of one variable *or* an instance of a second variable. In a strongly-typed language, the language must know the runtime type of the variable at each use. Therefore, it must carry Runtime Type Information (RTTI). The compiler will then typecheck before each use that the variable is indeed of the correct type.

Tests for which type an instance of a variable is are idempotent – meaning that at most one check per block is required.

The size of these objects is known at compile time and so they can either be allocated on the stack or the heap.

(c) The allocation of arrays with non-manifest bounds

An array with non-manifest bounds is an array where the size is not known at compile time. This means the array cannot be hardcoded into machine code. Instead, we must allocate it on the heap at runtime and the array must carry around information about its size and type. Additionally, the program must perform bounds checks before any array accesses.

Syscalls are made to allocate data onto the heap. On the stack, the arrays will be represented by objects containing the size of the array, the type of elements in the array and a pointer to the 0^{th} element in the array.

(d) The implementation of labels and `GOTO` commands

The language will compile to machine code. Machine code supports relative branches. So keep labels and `GOTO` in the machine code until the final stage of compilation. After the rest of the program has been compiled, iterate through the program and determine the number of instructions between the `GOTO` and the label. Then remove the label and replace `GOTO k` with a jump to the instruction immediately after where `k` was.