Extra problems, week 4

Charilaos Skiadas COMMUNITY TA · 17 days ago

I don't really know how many people work on and/or enjoy these, but I enjoy writing them so I'm going to keep doing it.

As always, these are completely extra, please only work on them if you have already completed all the expected material/homework/midterm, and have nothing else left to do for the week.

More fun with references

1. (This was sort of already done in week 3's notes) Write a function `makeCounter: int -> unit -> int`, with the following behavior: `makeCounter c` returns a function, say `f`, that when called returns successively increasing numbers starting from `c`. So the first `f()` returns `c`, the second `f()` returns `c+1` and so on.
2. Write a function `makeMultiCounter: unit -> string -> int`, with the following behavior: `makeMultiCounter()` returns a function that takes as inputs strings and returns integers. This function, call it `f`, should behave as follows: For any string `s`, each time `f s` is called it will return an ever increasing number, starting from `1`. So the first call to `f "x"` produces `1`, the second call to `f "x"` produces `2` and so on. A call to `f "y"` would return 1, but some later call to `f "y"` would return `2`. In other words, `f` keeps a separate count for each string input. Each call of `makeMultiCounter()` should be creating a distinct such counting function. No other top-level bindings should be created.
3. Write a version of `makeMultiCounter`, where multiple calls to `makeMultiCounter ()` all return the same counting function. Again, no other top-level bindings should be created.
4. Write a function `gen: ('a -> 'a) -> 'a -> (unit -> 'a)` that given a function `f` and an initial "seed" value `s`, returns a function `g` so that when `g` is called, the result `f s` will be computed and returned, and that return value will be used a the input to `f` the next time `g` is called. For example if `f = fn x => x * x` and `g = gen f 2`, then `g() = 4` the first time, `g() = 16` the second time and so on. Effectively, `g` will be generating the numbers `f(s)`, `f(f(s))`, `f(f(f(s)))` and so on.
5. Use `gen` to create the counter in the first problem.
6. Write a function `once: (unit -> 'a) -> 'unit -> 'a` that is given a function `f`, and behaves as follows: If `g` is equal to `once f`, then calling `g ()` will only result in calling `f` once, the first time. Every subsequent call should simply return the resulting value, without running `f` again. `f`

must not be called until `g` is called for the first time. No new top-level bindings or datatypes should be introduced.

7. Write similarly a function `only: int -> (unit -> 'a) -> 'unit -> 'a` that is given an integer `n` and a function `f`, and returns a function that for the first `n` times it is called it will call `f`, and from that point on it will recycle the `n` results computed thus far. For example, if `f` is the counter function from the earlier problems, and `g = only 3 f`, then calling `g` 5 times would result in the sequence of results `1,2,3,1,2`.
8. Write a function `cache: ('a -> 'b) -> ('a -> 'b)` that given a function `f` returns a function `g` that "caches" `f`'s computations. On a call to `g` with some input, if it is the first time that `g` has been call with that specific input, `f` should be called on that input and the result cached. Subsequent calls to `g` with the same input should not recompute `f`, and instead use the cached value. You can use a simple list of pairs to hold the cached values (in the absense of some ordering structure on `'a`, not much more is possible).
9. (Harder) Write a version of `cache: int -> ('a -> 'b) -> ('a -> 'b)` that takes as a first argument a cache size `n`. It will then cache up to `n` values. On subsequent cache misses you will need to remove a "stale" value from the cache. How you define "stale" is up to you, you can try to keep track of frequency of calls, "recency" using a counter etc. Finding a way to do this efficiently, so that you don't always end up removing the last element on a list and requiring a reconstruction of the list, may be challenging.
10. Write a function `throttle: int -> ('a list -> unit) -> ('a -> unit)` that is given an integer `n` and a function `f`, and returns a function `g` that "throttles" `f` in the following sense: As `g` is called, its inputs are "accumulated" into a list. On the `n` call, this list of `n` elements is passed as an argument to `f` and the process starts afresh. So `f` will be called on every `n`-th call of `g`, and it will be given as argument the list of the arguments that `g` had in the last `n` calls. You must preserve the order of those arguments.
11. Write a function `throttle2: int -> ('a -> unit) -> ('a -> unit)` that acts as follows: The first time `g` is called, its input is stored, and nothing else happens until `g` is called for the `n`-th time. When `g` is called for the `n`-th time, it calls `f` with the stored argument, and the process starts anew (so the `n+1`-st argument will be stored, but `f` won't be called on it until the `2n`-th call to `g`, and so on.
12. Write a function `delayed: ('a -> 'b) -> 'a -> ('a -> 'b)` that acts as follows: If `g = delayed f x0` then the call `g x1` will return `f x0`, the call `g x2` will return `f x1` and so on. Namely each time `g` is called, it in turn calls `f` with the argument that it was given in its *previous* call, using `x0` for the first time it's called.
13. There is a built in operator `before` which behaves as follows: `e1 before e2` evaluates `e1`, then evaluates `e2` but returns the value of `e1`. The built in version has type `'a * unit -> 'a`. Define a similar infix operator `befor: 'a * 'b -> 'a`. (This is very very easy once you think about it).

A simple hash table

We will implement a simple hash table with linked list chaining. This is probably not the best way to arrange it, but it will be good for practice.

We start with the datatypes

```
type 'a hashVector = 'a list ref vector
```

```
type ('a, 'b) hashTable = { hash: 'a -> int, eq: 'a * 'a -> bool, size: int, vec: ('a
* 'b) hashVector }
```

So a hashTable consists of: a hashing function, a function to compare two values for equality, and a hashVector. A hashVector is a vector (immutable fixed width array (http://sml-family.org/Basis/vector.html)) whose entries are references to linked lists. Upon creation we populate the vector with empty list references. Then each time a new value needs to be hashed, its proper assigned location in the vector is found via the hash function, then the list stored in that reference is found, the element is appended to that list, or replaced if the key already exists, then the new list is stored in the reference.

Here are some methods you should implement:

```
makeEmpty: ('a -> int, 'a * 'a -> bool, int) -> ('a, 'b) hashTable
```

This accepts a hash function, and equality function, and a size integer, and creates an empty hashTable of that size. The function `Vector.tabulate` will come in handy for creating the necessary vector. This is the only function that creates a new hashTable, all other functions will update an existing hashTable.

Next we will need some methods to retrieve/update key-value pairs from a list. These are needed once you have drilled down to the correct hash location and now need to update the list that is there.

```
lookup_list: ('a * 'a -> bool) * ('a * 'b) list -> 'a -> 'b option
insert_list: ('a * 'a -> bool) * ('a * 'b) list -> ('a, 'b) -> ('a * 'b) list
remove_list: ('a * 'a -> bool) * ('a * 'b) list -> 'a -> ('a * 'b) list
```

The first function looks up for a particular key in the list, using the provided function (probably eq) as argument, and if it finds it then returns `SOME v`, where `v` is the corresponding value.

The second function inserts the new key-value pair at the end of the list if that key doesn't already exist. If there is a key-value pair with the same key, then it gets replaced.

The third function, predictably, removes a key if it is present.

Now we can write the corresponding function at the level of the hashTable:

```
lookup: ('a, 'b) hashTable -> 'a -> 'b option
insert: ('a, 'b) hashTable -> 'a * 'b -> unit
remove: ('a, 'b) hashTable -> 'a -> unit
```

The first function looks up a key in the hashTable, the second inserts a new key-value pair into the hashTable. Your implementations would have to use the hash function to locate the `list ref` in which this pair should be looked in/inserted, then the corresponding list function would be used. In the case of insert/remove, the contents of the ref need to be updated with the new list.

```
load: ('a, 'b) hashTable -> int
```

Computes how many key-value pairs are currently stored in the hashTable. With our current implementation of hashTables this will not be particularly efficient. You may change the hashTable definition to maintain the load information in an `int ref` field, and adjust the methods accordingly (note that insert may replace/add depending on whether the key is present, and remove many not actually remove).

A simple calculator

This example illustrates mutual recursion in both types and functions. We will create a basic datatype that supports arithmetic expressions as well as statements for storing and printing variables. For that we need one datatype for expressions and one for statements, and they depend on each other. So here is one attempt:

```
datatype Exp = Add of Exp * Exp
             | Sub of Exp * Exp
             | Mult of Exp * Exp
             | Const of int
             | Var of string
             | Compound of Stm list * Exp
and      Stm = Assign of string * Exp
             | Print of Exp
```

For simplicity we will assume that any assignments that happen within the statements in a compound expression are local to that expression. So think of a compound statement as a `let-in-end` block.

In order to manage the assignments, we will maintain a "calculator memory" in the form of a `(string * int) list`. Each assignment adds a "binding" in this memory, each `Var` expression accesses the memory. To manage this we will need some minor definitions, that you should be able to come up with:

```
exception UnboundVar of string
type memory = (string * int) list
save: memory -> string * int -> memory
load: memory -> string -> int
```

Now we want to implement an "evaluation" strategy. It will need to be a pair of functions:

```
evalExp: memory -> Exp -> int
evalStm: memory -> Stm -> memory
```

Each of these functions requires as input the current state of the memory, as well as an expression/statement. Expressions need to evaluate to an integer. Statements return the altered state of the memory.

1. A `Var` expression needs to look the variable up in the current memory context, and retrieve its value.
2. A `Assign` statement needs to evaluate the expression in the current memory context, assign the

returned value to the variable string, and return the resulting memory. This may shadow an existing value, and that is OK (or you may choose to remove any previous values, up to you).
3. A `Compound` expression computes each statement in turn, updating the memory as it goes along (a fold would do nicely for that), and finally computes the expression in the resulting memory context. Note that these changes to the memory only affect the computation of this expression.
4. A `Print` statement should evaluate the expression and use SML's `print` method to print the resulting value in its own line. If the expression is a `Var`, then it should print something like `x = ...`, otherwise it should print simply the resulting value.

Here is a skeleton for how the implementation might look like. It will be your task to fill in, most are rather easy.

```
fun evalExp mem e =
   let val ev = evalExp mem
   in
      case e of
        Add (e1, e2)       => ...
      | Sub (e1, e2)       => ...
      | Mult (e1, e2)      => ...
      | Const i            => ...
      | Var s              => ...
      | Compound (stms, e) => ...
   end
and evalStm mem stm =
   case stm of
     Assign (s, e)      => ...
   | Print (Var s)      => ...
   | Print e            => ...
val eval = evalExp []    (* Shortcut for evaluation with empty memory. No need to chan
ge. *)
```

Here's one test of a successful implementation:

```
(* Should return 6 *)
val exp1 = eval (Compound ([Assign ("y", Add(Const 2, Const 3)), Print (Var "y")], Add
 (Var "y", Const 1)))

(* Should return 5 *)
val exp2 = eval (Compound ([Assign ("y", Const 2)], Add (Compound ([Assign ("y", Const
 3), Print (Var "y")], Var "y"), Compound ([Print (Var "y")], Var "y"))))
```

Some extensions to consider:

1. Add a `PrintMem` statement, that prints the entire memory contents (but not shadowed variables).
2. Add a "value" datatype that holds the possible values of evaluation. A possible value would be either `IntV i` or `ErrorV s` where `s` is a string. Memory should be adjusted to hold `string *`

`value` pairs, `evalExp` should be adjusted to return a `value`, and unsuccessful variable lookups would result in an `Error` value. Add division, where division by 0 would result in an appropriate `Error` value, as would non-perfect division (e.g. 5 / 3. Try to create appropriate error messages for each instance).

3. Add boolean values, ways to compare/create boolean values, and if-then-else expressions. (As an easier start, you can implement `ifZero e1 then e2 else e3`)

Simple stack-based calculator

This one is for more practice with pattern matching. We will build a simple stack-based calculator, where most operations pop elements at the top of the stack, perform a computation on them, and insert their results back into the stack.

A stack will simply be a list of integers:

```
exception Empty
type stack = int list
empty: stack
push: stack -> int -> stack
pop: stack -> (i, stack)
```

Popping returns a pair of the value at the top of the stack and the remaining of the stack.

The datatype `Oper` holds the operations we can perform.

```
datatype oper = Push of int | Pop | Add | Sub | Mult | Neg | Print
```

`Push i` adds the integer `i` onto the stack. `Pop` simply removes the top of the stack. `Add` takes out the two elements at the top of the stack, adds them and pushes the result back onto the stack. Similarly for `Sub` and `Mult` (where for `Sub` the number at the top is the negative one). `Neg` negates the value at the top of the stack. `Print` prints out in a new line the value at the top of the stack.

All of these operations should raise an `Empty` exception if there are not enough elements on the stack for them to operate.

The goal is of course to write a functions:

```
evalOp: stack -> oper -> stack
eval: oper list -> unit
```

Here `evalOp` evaluates a single operator with a given stack state, and returns the updated stack state, while `eval` takes a series of operations, and performs them on an empty stack, discarding the final result.

You can approach the problem in two ways. One is to rely solely on the `push/pop` functions defined earlier, and have those raise the appropriate exceptions when needed. The other is to utilize the implementation of the stack as a list, and to have a nested pattern match on the pair of the operation and the stack, like so:

```
evalOp stk o =
   case (o, stk) of
     (Push i, _)          => ...
   | (Pop, _::stk')       => ...
   | (Add, i1::i2::stk')  => ...
   | (Sub, i1::i2::stk')  => ...
   | (Mult, i1::i2::stk') => ...
   | (Neg, i1::stk')      => ...
   | (Print, i1::_)       => ...
   | _                    => ...
```

9 · flag

Charilaos Skiadas COMMUNITY TA · 13 days ago

Here is one more project.

The following project is quite long with many components and around 5 different structures/signatures. It will be a considerable time investment to go through it, but I hope some of you will and will discuss it right here.

It is a considerably larger problem that the other extra problems.

Mini CPU emulator

In this section, which serves as practice on signatures, structures as well as other earlier concepts, we will build a model a mini cpu emulator, that will capture the use of registers, memory, instructions etc. You should probably be familiar with the basics of Computer Organization course before attempting this problem.

This mini-project will also use some more advanced features that have not been introduced in the class. I will try to explain a little when that occurs.

Various simplifications will be made in the treatment.

Words

We start small, with a structure to abstract away our notion of "word".

`memWord` will represent the standard "size" of our memory blocks, as well as register size, and we are going with 32 bit words here. The signature on this will be fairly simple:

```
signature MEMWORD =
sig
   type memWord

   val fromInt: int     -> memWord
   val toInt  : memWord -> int
```

```
end
```

The implementation is equally simple:

```
structure MemWord :> MEMWORD =
struct
   type memWord = Word32.word
   val fromInt = Word32.fromInt
   val toInt   = Word32.toInt
end
```

All we care about at this point is having an easy way to turn these words into integers to perform integer arithmetic and to be able to initialize them.

Registers

We continue with a signature for the register table:

```
signature REGISTER =
sig
   type register
   structure W : MEMWORD
   type memWord = W.memWord


   val available_regs : register list (* Registers available for normal use *)
   val esp: register                  (* Stack pointer *)
   val ip: register                   (* Program counter *)
   val toString: register -> string   (* string representation of a register *)
   val get: register -> memWord
   val set: register -> memWord -> unit

end
```

Notice that we are implementing the register type as an abstract data type. This way we could easily increase the number of registers later on, without exposing it to anyone else. The only way that the rest of our program can know what registers we have is via the `available_regs` list.

The `available_regs` variable should be a list of the registers available for use. For our example we will use 4 "normal" registers, plus a stack pointer register and a program counter register. Only the first 4 would be considered "available". The `toString` function returns a textual representation of the register, e.g. `%eax` for the `EAX` register.

For each register, we can get or set its value via the appropriate methods.

We are doing here something you have not seen before. As part of this structure, we are going to refer to another structure, `W`, which we require must follow the `MEMWORD` signature. We then declare that the type memWord will be the same as the type `W.memWord` declared inside the structure `W`. Later on in our structure implementation, we will identify this structure `W` with our `MemWord` structure. This is one of the ways to link structures together in SML (the other being functors).

The first thing you will need to do is implement the corresponding structure, which I have started for you:

```
structure Register :> REGISTER =
struct
   datatype register = EAX | EBX | ECX | EDX | ESP | IP
   structure W = MemWord
   type memWord = W.memWord

   val available_regs = ...
   val esp            = ...
   val ip             = ...

   fun toString reg =
      case reg of
         ...

   fun toInt reg =
      case reg of
        IP  => 0
      | ESP => 1
      | EAX => 2
      | EBX => 3
      | ECX => 4
      | EDX => 5

   val regTable = Vector.tabulate (6, fn _ => ref (W.fromInt 0))

   fun getRef reg = ...     (* Retrieves the appropriate reference from the table *)

   fun get reg   = ...      (* Get the value stored in the appropriate reference *)
   fun set reg v = ...      (* Set the value in the appropriate reference *)
end
```

The register table is kept as a Vector of size 6 that holds references to memWords. We have a function `toInt` that maps each register to an index. You will need to implement the method `getRef` that takes a register and retrieves the reference from `regTable` corresponding to it, and the methods `get` and `set` that manipulate the value stored in that reference.

Memory

We will describe memory simply as a vector of references, but that implementation detail will be hidden. We will however work with whole words rather than bytes to simplify things a bit. So to move to the next "location" in memory, you would increment by 1 rather than say 4.

We will use the same trick to incorporate the `MemWord` structure.

Here is the signature:

```
signature MEMORY =
sig
   eqtype location
   structure W : MEMWORD
   type memWord = W.memWord
   exception OutOfBounds

   val maxLocation: memWord
   val getLocation: memWord -> location

   val get: location -> memWord
   val set: location -> memWord -> unit
end
```

`location`s are really just `memWord` values that represent indexing into the vector that represents the memory. We do not specify the type however, to discourage blinding using numbers (`memWord`s) where memory locations are concerned.

Restrictions on the memory come in various ways. First off, a constant `maxLocation` indicates the `memWord` value, beyond which you will be trying to access out of memory. `getLocation` is the only way to obtain a memory location, and it will raise an `OutOfBounds` exception if you give it a negative or too large value. This way the `get` and `set` functions are guaranteed to work. One is retrieving the value stored in a specific location, the other is storing the value.

Here is a start on the structure that implements this signature:

```
structure Memory :> MEMORY =
struct
   structure W = MemWord
   type memWord = W.memWord

   type location = memWord
   exception OutOfBounds

   val maxLocation = W.fromInt 20000      (* Just an arbitrary number. *)

   (* Initialize memory with 0 *)
   val mem = Vector.tabulate (toInt maxLocation) (fn _ => ref (W.fromInt 0))

   fun getLocation i = ...        (* Need to make sure i is valid memory location *)

   fun getRef loc    = ...
   fun get loc       = ...
```

```
    fun set loc value = ...
end
```

You will need to implement `getLocation`, `get` and `set`. They will need to access the `mem` vector at the appropriate location, and retrieve the reference that is there. A `getRef` helper method can help with that (the Vector structure has some useful methods for accessing its entries). Then `get` retrieves the value in that reference, while `set` assigns a new value in that reference.

Notice how the signature only contains a small part (the "public" part) of the definitions in the structure.

Instructions

We proceed with modeling the subset of instructions we will consider. As part of the instruction stucture we will also need to talk about the various types of operands, so there will be an `operand` type for that. We will also need to device an encoding scheme for turning `memWord`s into instructions and vice versa. Our Instruction signature/structure will need to access parts of the Memory structure and the Register structure. To that end, the Instruction signature will "include" those other structures.

Unlike the earlier structures that used abstract data types, Instruction exposes a lot of its datatypes to allow later structures, like the main CPU program to operate on them. A long pattern match on those types is the most effective way to achieve that.

For simplicity, we have allowed only operations on the full `memWord`s and only a limited number of arithmetic and logical operations.

So without further ado, here is the relevant signature:

```
signature INSTRUCTION =
sig

   exception InvalidInstruction

   structure R : REGISTER
   structure M : MEMORY

   type memWord  = M.memWord
   type location = M.location
   type register = R.register

   datatype operand = Reg of register
                    | Imm of memWord
                    | Mem of { D : memWord,
                               Rb: register,
                               Ri: register option }
   datatype cond  = EQ | NEQ | LE | LEQ
   datatype instr = HALT
                  | MOV of { src: operand, dest: operand }
```

```
                | ADD of { src: operand, dest: operand }
                | SUB of { src: operand, dest: operand }
                | CMP of { src: operand, dest: operand }
                | TST of { src: operand, dest: operand }
                | OR  of { src: operand, dest: operand }
                | AND of { src: operand, dest: operand }
                | NOT of operand
                | JMP of cond option * location
                | PUSH of operand
                | POP of operand
                | SHL of operand * int
                | SHR of operand * int

   val encode: instr   -> memWord
   val decode: memWord -> instr
end
```

We essentially define a lot of datatypes, and export only two functions. One to encode an instruction into "machine code", one to decode the instruction. You can implement these however you like really, as long as you ensure that they are inverses of each other. We have an exception to throw if the decoding process fails to produce a viable instruction.

The operand datatype allows 3 types of operands: Registers, Immediate values and Memory addressing. As typical we require that immediate values only appear as sources, and that memory addressing cannot be used at both source and destination (We could try to enforce these conditions on a datatype level, worth thinking about how to do it). `HALT` indicates the end of the program. `JMP` indicates an unconditional jump if `cond option` is set to `NONE`. `PUSH` and `POP` are meant to add/remove from the "stack".

The structure would look very similar, but with those two functions implemented. Here's how it would start:

```
structure Instruction :> INSTRUCTION =
struct
   exception InvalidInstruction

   structure R = Register
   structure M = Memory

   type memWord  = M.memWord
   type location = M.location
   type register = R.register
   ...
end
```

You will need to repeat all the datatype definitions. You can implement the two functions in a dummy way for now. You will be able to follow most of the rest without them.

CPU

We now proceed to the main structure, that models a CPU. It really offers very little external interface, essentially just a `run` function that is meant to load and execute a list of instructions and return a "result". As it exposes some other structures, you are also able to call their methods (e.g. read register values).

```
signature CPU =
sig
   structure R : REGISTER
   structure M : MEMORY
   structure I : INSTRUCTION
   structure W : MEMWORD

   type memWord  = M.memWord
   type register = R.register

   type program = I.instr list

   val run: program -> int
end
```

Here is how an implementation of this might look like:

```
structure Cpu :> CPU =
struct

   structure R = Register
   structure M = Memory
   structure I = Instruction
   structure W = M.W

   type memWord  = W.memWord
   type location = M.location
   type register = R.register

   type program = I.instr list

   val EAX = hd R.available_regs    (* bit of a hack *)

   (* flags *)
   datatype flags = { ZF: bool, CF: bool, SF: bool, OF: bool }

   val flagsRef = ref { ZG: false, CF: false, SF: false, OF: false }
```

```
   fun setFlags fs = flagsRef := fs
   fun getZF () = #ZF (!flagsRef)
   fun getCF () = #CF (!flagsRef)
   fun getSF () = #SF (!flagsRef)
   fun getOF () = #OF (!flagsRef)


   fun load prog       = ... (* program -> unit --- Loads program in memory *)
   fun fetch ()        = ... (* unit -> I.instr --- Loads current instr and increments
 IP *)
   fun exec instr      = ... (* I.instr -> unit --- Executes instr *)
   fun read_exec_loop  = ... (* unit -> unit    --- Read/execute loop. Stops at HALT *
)
   fun reset           = ... (* unit -> unit    --- Reset ip/esp *)

   fun run prog = (
      reset ();
      load prog;
      read_exec_loop ();
      W.toInt (R.get EAX) )
end
```

We have implemented a rudimentary flags system, with a datatype and functions to get and set flags. Since each instruction will have to set all the flags, while it might want to know values of individual flags, we have only 1 joint set method but individual get methods.

There are 5 functions that you will need to implement, and we will talk about them briefly in a moment. Then the main function that "runs" the program is fairly imperative, and uses the `(e1; e2; e3; ...; en)` form, which executes each expresion in order, discarding the results until the last one. So it starts by calling `reset`, which resets the counters. It then loads the program into memory. Finally it calls the looping function `read_exec_loop` which runs the "machine" till encounters a `HALT` instruction. Finally the last expression computes a "return value" as the value currently in the EAX register.

The bulk of the work will be done in `exec`, which executes a single instruction. But before we get to it we will discuss the other functions.

We will start with `load` and `fetch`. Ideally what needs to happen here is that the program code needs to be placed somewhere, so that "fetching" based on the program counter would give you the next function. If you have implemented the "decode" and "encode" functions of the Instruction structure, then all you need to do is use those to save the program in memory, starting at address 0. So with encode and decode implemented:

- Given the instruction list/program, `load` encodes each instruction in turn and saves it to a memory location, starting at 0.
- `fetch` reads the program counter, goes to the corresponding place in memory, and fetches the memWord stored there, and decodes it into an instruction.

If you want to avoid using `encode` and `decode`, you can keep the instructions into a Vector of

`I.instr`s, then use the program counter to looks things up in this vector instead of in memory. As the instruction counter may have been set to an out of bounds value by a `JMP` instruction, you need to decide how to handle out of bounds errors. It does however make the load and fetch code a lot easier.

Next let us talk about `reset`. It simply has to set the program counter to 0, and the stack pointer to the top of our memory, as our stack will start from there and walk its way down towards 0. You can choose to reset more state if you like.

`read_exec_loop` is next. It simply fetches the "next instruction" and increments the program counter, then if that instruction is HALT it returns, otherwise it `exec`s that instruction and recursively calls itself. Try to write it yourself first before looking at the rest:

```
fun read_exec_loop () =
   case fetch () before inc_ip() of
      I.HALT  => ()
    | instr => (exec instr; read_exec_loop ())
```

We are using here the "before" construct. `e1 before e2` evaluates `e1`, then evaluates `e2` but returns `e1` as its value. It is a way to ensure the program counter is incremented *after* it has been used.

Lastly, the `exec` function, which is the heart and soul of the CPU. It is going to be a large case expression on the various forms that the instruction can take. Keep in mind that as the instructions are really defined inside the Instruction structure, which is called `I` within our `Cpu` structure, so you need to refer to them in the same way we did with `I.HALT` in the previous code snippet.

Now what needs to happen on each instruction of course depends on the instruction. Most instructions will access memory or register values, so a helper method that returns the memory/register reference pointed to by an operand would be critical and probably used multiple times. Some instructions may change the program counter. All instructions will need to set the flags before wrapping up. In general, most instructions will need to do all of the following:

1. Fetch two references corresponding to the values to manipulate.
2. Operate on the values stored there.
3. Store the result in the target reference, or perform whatever other change should happen.
4. Set the flags.

The End

Well, this is the end of our journey. If you've managed to make it this far, congratulations! This was quite a lot to take in. Try to write some assembly programs and watch them execute!

What's that? You want to expand this more? Well here's some ideas:

1. Add some function-calling discipline, probably around two new instructions `CALL` and `RET`. You will need some way to "save" functions, so that that the program writer doesn't need to keep track of memory addresses for everything.
2. In general, add support for labels, which later turn into memory addresses.
3. Add "built-in functions" for things like basic input/output.
4. Post your suggestions!

3 · flag

+ Comment

William Blackerby · 13 days ago

This is so cool. Lots to chew on. Hope I can get around to it. Perhaps not during the actual run of this course, but when I'm looking for something to do when it's over. Thanks!

0 · flag

+ Comment

Pierre Barbier de Reuille · 12 days ago

For the hash tables, how are the `insert_list` and `remove_list` functions supposed to do anything? Their signature is:

```
insert_list: ('a * 'a -> bool) * ('a * 'b) list -> ('a, 'b) -> unit
remove_list: ('a * 'a -> bool) * ('a * 'b) list -> 'a -> unit
```

So they are not working on references and return a `unit` ...

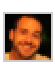
I would expect them to return `('a * 'b) list`!

0 · flag

Charilaos Skiadas COMMUNITY TA · 12 days ago

You're correct, fixed.

0 · flag

 Marco Fabbri Signature Track · 10 days ago

As an "addition" to the HashMap problem, it's nice to implement it as a module and define an "appropriate" signature (maybe you want to hide some helper functions).

I had to introduce an equality type for keys like the following (I have to admit it, I spent more time with the type checker than implementing lookup/insert/remove functionality, I run into some value restriction issues):

```
val lookup : (''a, 'b) hashMap -> ''a -> 'b option
```

The structure implementation is 27 LOC (with a ref int for load), higher order functions really shine :-)

Thanks Charilaos the great problems!

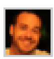
0 · flag

Charilaos Skiadas COMMUNITY TA · 10 days ago

Ah I think you might find the `table.sml` and `table.sig` implementations here relevant, it uses a functor where the passed-in structure essentially is there to tell you how to compare your keys (well, sort of). You can probably get some ideas from there on how to use a functor to not have to assume your keys are an equality type.

Glad to see people having fun with these!

0 · flag



Marco Fabbri Signature Track · 8 days ago

Thanks! (very interesting resource). Actually, I had mistakenly used a `<>` inside the HashMap (instead of `eq`, which in fact shields from the equality type issue), in the end I am pretty pleased with the result [SPOILER] https://gist.github.com/mrfabbri/86b94ce9d2b06c641fd7

0 · flag

+ Comment

New post

To ensure a positive and productive discussion, please read our forum posting policies before posting.

B I Link <code> Pic Math Edit: Rich Preview

Make this post anonymous to other students

Subscribe to this thread at the same time

Add post