Last Week: Compiler Lemma

- If (e, m) $\longrightarrow_a$ n then
  (compa e, [], m) $\longrightarrow_m^*$ ([], [n], m).
- If (e, m) $\longrightarrow_a$ b then
  (compb e, [], m) $\longrightarrow_m^*$ ([], [WRAP b], m).

- Booleans: TRUE ↦ 1; FALSE ↦ 0
- We had to generalise over all stacks s.
- We only considered a small subset of the machine.

A Machine

- ... has a stack, a single register and memory
- runs a list of instructions

```
datatype instr =
    JMPF  "nat"                   jump forward n steps, if reg. is False
  | JMPB  "nat"                          backward n steps
  | FETCH "loc"                          move memory to top of stack
  | STORE "loc"                          pop top of stack to memory
  | PUSH  "nat"                          push to stack
  | POP                                  stack to register
  | SET   "nat"                          set register
  | OPU   "nat ⇒ nat"                    pop one from stack and apply f
  | OPB   "nat ⇒ nat ⇒ nat"   pop two from stack and apply f
```

A Machine

- Last week we had “linear” programs and therefore used a simplified machine

```
inductive
  step ("'(_,_,_') ⟶m '(_,_,_')")
where
    "(PUSH n#p, s, m) ⟶m (p, n#s, m)"
  | "(FETCH l#p, s, m) ⟶m (p, m l#s, m)"
  | "(OPU f#p, n#s, m) ⟶m (p, f n#s, m)"
  | "(OPB f#p, n1#n2#s, m) ⟶m (p, f n2 n1#s, m)"
```

- Behaviour of the “full” machine

```
inductive
 step2 ("'(_,_,_,_,_') ⟶m '(_,_,_,_,_')")
where
 "(PUSH n#p, q, r, s, m) ⟶m (p, PUSH n#q, r, n#s, m)"
| "(FETCH l#p, q, r, s, m) ⟶m (p, FETCH l#q, r, m l#s, m)"
| "(OPU f#p, q, r, n#s, m) ⟶m (p, OPU f#q, r, f n#s, m)"
| "(OPB f#p, q, r, n1#n2#s, m) ⟶m (p, OPB f#q, r, f n2 n1#s, m)"
| "(POP#p, q, r, n#s, m) ⟶m (p, POP#q, n, s, m)"
| "(SET n#p, q, r, s, m) ⟶m (p, SET n#q, n, s, m)"
| "(STORE x#p, q, r, n#s, m) ⟶m (p, STORE l#q, r, s, m(x:=n))"
| "(JMPF i#p, q, Suc 0, s, m) ⟶m (p, JMPF i#q, Suc 0, s, m)"
| "i≤length p ⟹
  (JMPF i#p, q, 0, s, m)
     ⟶m (drop i p, (rev (take i p))@(JMPF i#q), 0, s, m)"
| "i≤length q ⟹
  (JMPB i#p, q, r, s, m)
     ⟶m ((rev (take i q))@(JMPB i#p), drop i q, r, s, m)"
```

```
fun
  compc :: "cmd ⇒ instr list"
where
  "compc SKIP = []"
| "compc (x::=a) = (compa a) @ [STORE x]"
| "compc (c1;c2) = compc c1 @ compc c2"
| "compc (IF b THEN c1 ELSE c2) =
    (compb b) @ [POP] @
    [JMPF (length(compc c1) + 2)] @ compc c1 @
    [SET 0, JMPF (length(compc c2))] @ compc c2"
| "compc (WHILE b DO c) =
    (compb b) @ [POP] @
    [JMPF (length(compc c) + 1)] @ compc c @
    [JMPB (length(compc c)+length(compb b) + 2)]"
```

Compiler Lemma for Commands

- If $(e, m) \longrightarrow c\; m'$ then $\exists\, r'$
  $(\text{compc } e, [\,], r, [\,], m) \longrightarrow m^*$
  $([\,], \text{rev } (\text{compc } e), r', [\,], m')$.

- We need to prove: If $(e, m) \longrightarrow c\; m'$ then $\exists\, r'$
  $(\text{compc } e, q, r, s, m) \longrightarrow m^*$
  $([\,], \text{rev } (\text{compc } e) @ q, r', s, m')$.

- The content of the register is determined by the program (therefore $\exists\, r'$).

What Have We Achieved?

- I caught an “off-by-one” error in the compiler function and a “copy-paste” error in the machine definition.
- We can “play-around” with the formalisation.

- The language is very simple: no problems with null pointers.
- The language makes essentially no promises (we are completely on our own).
- No local variables.

Current Research

- Scaling the reasoning to real languages (they have for example garbage collectors)
- Reasoning about languages with binders

  $$t ::= x \mid t_1\; t_2 \mid \lambda x.t$$

  The last constructor represents a function which takes one argument.

- $(\lambda x.\lambda y.x + y)\; 2\; 3 \longrightarrow 2 + 3$
- In general

  $$(\lambda x.t)\, t' \longrightarrow t[x := t']$$

Current Research

- Design languages that make “promises”

  For example, if the compiler accepts the program, it will never crash during run-time.

- This needs expressive type-systems.
- There is a tentions about what you can decide at compile-time and what you have to check at run-time (for example out-of-bounds errors).