Home task №2: /* Big, hard, painful, and a bit philosophical */

1. Implement mix algorithm (see. Fig 1).
2. Implement I Futamura projection:
$$target_1 = [\![mix^{FlowChart}_{FlowChart}]\!]_{FlowChart}[int^{TM}_{FlowChart}, div_{int_{TM}}, vs_0 = [Q \to source_{TM}; \dots]]$$
3. Implement II Futamura projection:
$$\begin{aligned} comp = [\![mix^{FlowChart}_{FlowChart}]\!]_{FlowChart}[\quad & mix^{FlowChart}_{FlowChart}, \\ & div_{mix^{FlowChart}_{FlowChart}}, \\ & vs_0 = [program \to int^{TM}_{FlowChart}; \\ & \qquad\qquad div \to div_{int^{T}_{FlowChart}M}; \dots]] \end{aligned}$$
and compare the result with the expected one (see. Fig 2). One may need auxiliary functions like relabelling, pretty-printing, and so on.
Also, check that $comp$ really is a compiler, i.e. $target_2 = [\![comp]\!]source_{TM}$ is indeed a compiled version of a Turing-Machine program $source_{TM}$.
Compare $target_1$ with $target_2$. Are they identical? If no, why and is it normal?
4. Improve mix algorithm from task 1 by using "The Trick" on variable bb (basic block). Repeat tasks 2 and 3 and compare the results.
5. Implement FlowChart interpreter on FlowChart, $int^{FlowChart}_{FlowChart}$ and redo tasks 2 and 3 with $int^{FlowChart}_{FlowChart}$ instead of $int^{TM}_{FlowChart}$ and $find\_name_{FlowChart}$ example instead of $source_{TM}$.
Does the compiled program contains any of interpreter source code or data? If so, could you improve mix or/and interpreter(-s) in such a way that the generated target program contains no parts of the interpreter (i.e. for example, pretty-printed and relabelled $find\_name$ program is identical to the source one)?
6. Implement III Futamura projection and check that it really results in a compiler generator for both interpreters. Evaluate your result.

Remember a self-application recipe:

- "The Trick".
- Note that maybe pp' may achive not all source program labels (since we perform transition compression during specialization). We may only scan `blocks-in-pending`. Note that mix-generated compiler from Figure 2 contains only three of interpreter's fifteen labels.
- Live and dead static variables (a la "live variables analysis").
Exclude from specialized program point all variables that are dead in it, i.e. those static variables whose values are not used in the basic block.
- NB:
  - $poly$ has to be small,
  - in self application check that you do not confuse variables of different mix-es of the same name.

```
read (program, division, vs0);
pending ← { (pp0 , vs0) };          (* pp0 — initial program point *)
marked ← ∅;
while pending ≠ ∅ do
  Pick (pp, vs) ∈ pending and remove it;
  marked ← marked ∪ {(pp, vs)};
  bb     ← lookup (pp, program);  (* Find correcponding basic block labeled by pp *)
  code   ← initial_code (pp, vs); (* An empty basic block with label (pp, vs) : *)
  while bb ≠ ∅ do
    command ← first_command (bb); bb ← rest (bb);
    case command of
    X ← exp:
      if X is static by division
      then vs   ← vs [X ↦ eval(exp, vs)];          (* Static assignment  *)
      else code ← extend(code, X ← reduce(exp, vs)); (* Dynamic assignment *)
    goto pp': bb ← lookup (pp', program);  (* Compress the transition *)
    if exp then goto pp' else goto pp'':
      if exp is static by division
      then (* Static conditional *)
        if eval (exp, vs) = true          (* Compress the transition *)
        then bb ← lookup (pp' , program);
        else bb ← lookup (pp'', program);
      else (* Dynamic conditional *)
        pending ← pending ∪ ({(pp', vs), (pp',vs)} \ marked );
        code    ← extend (code, if reduce(exp, vs) goto (pp', vs) else (pp'', vs));
    return exp:
      code ← extend(code, return reduce(exp, vs));
    otherwise: error;
  residual ← extend(residual, code); (* Add new residual basic block *)
return residual;
```

Fig. 1: The mix algorithm for FlowChart

```
 1  read (Q);
 2    pending ← {('init, Q)}; (* A la recursion stack in recursive descent compiter; *)
 3    marked  ← {};           (* Also track correspondence between labels in the source *)
 4                            (*   and target programs *)
 5    while pending ≠ '() do
 6    |  Pick (pp, vs) ∈ pending and remore it;
 7    |  marked ← marked ∪ {(pp, vs)};
 8    |  case pp of
 9    |  init:
10    |    Qtail ← Q;
11    |    generate initial code;
12    |    while Qtail ≠ '() do (* While loop from TM interpreter *)
13    |    |  Instruction ← car (Qtail);
14    |    |  Qtail       ← cdr (Qtail);
15    |    |  case Instruction of (* TM interpreter dispatch *)
16    |    |  right:
17    |    |    code ← extend (code, Left  ← cons (firstsym (Right), Left);
18    |    |                         Right ← cdr  (Right););
19    |    |  left:
20    |    |    code ← extend (code, Right ← cons (firstsym (Left), Right);
21    |    |                         Left  ← cdr  (Left););
22    |    |  write s:
23    |    |    code ← extend (code, Right ← cons (s, cdr (Right)););
24    |    |  goto label:
25    |    |    Qtail ← new_tail (label, Q);
26    |    |  if s goto label:
27    |    |    pending ← pending ∪ ({('cont, Qtail), ('jump, label)} \ marked);
28    |    |    code    ← extend (code, if s = firstsym (Right)
29    |    |                            goto ('jump, label)
30    |    |                            else ('cont, Qtail););
31    |    |_ otherwise: error;
32    |  cont: if Qtail ≠ '() goto line12; (* The first while−command *)
33    |  jump: Qtail ← new_tail (label, Q);
34    |        if Qtail ≠ '() goto line12;
35    |  otherwise: error;
36    |_ residual ← extend (residual, code);
37    return residual;
```

Fig. 2: A mix generated compiler from Turing-Machine to FlowChart