Figure 1: The mix algorithm for FlowChart

```
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;
```

0.1 The second Futamura projection

Recall that we are not address problem of providing automated *bta*. Instead, for the sake of simplisity, we provide program's division as an additional argument for *mix*.

The first Futamura projection: $target = [\![mix]\!]\ [int,\ div_{int},\ \overbrace{[p]}^{vs0_{int}}\ ].$

Our goal (The second Futamura projection):

. $$compiler = [\![mix_2]\!]\ [mix_1,\ div_{mix_1},\ \overbrace{[int,\ div_{int}]}^{vs0_{mix_2}}].$$

Let us construct *mix*'s division:

1. Variables *program*, *division* are clearly static;
2. Variables *vs*, *vs*0 are clearly dynamic;
3. Variables *pending*, *marked*, *code*, *residual* are dynamic by the congruence principle;
4. Variables *pp* and *bb* are also dynamic for the same reason; And thus, so are variables *command*, *pp*′, *pp*″ and so on.

The result can be seen on Figure 2. It is obvious that no good result can be achieved with such division.

Figure 2: The mix algorithm with division defined by congruence principle

```
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;
```

Let us recall that variable *pp* is a label from a static, i.e. known, program *program* and *bb* is its basic block. It is natural be expect *pp*, and *bb*, and *command* to be static.

def A variable is said to be of *bounded static variation* if it can only assume one of finitly many values, and that its possible value set is statically computable.

Obviously, *pp* is of *bounded static variabtion*. Thus, we are able to generate a specialized code for each of its values. As a consequence, variables *bb*, *command* and others become static. The corresponding division is hsown on Figure 3.

Figure 3: The mix algorithm with division where pp is classified as a variable of bounded static variation

```
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;
```

0.2 Mix generated compiler

The mix generated compiler is close to a traditional recursive descent compiler. Its structure is a mixture of the mix and the TM interpreter structures. The mix generated compiler is shown on Figure 4.

The compiler obvously derived from the interpreter: it follows the *flow* of control as determined by semantics (which itself determined by the interpreter) instead of classical linear source program scan. Moreover syntactic despatch in the inner while-loop is obviousle the one derived from the interpreter.

Note, that the pairs (init, Q), textsf(cont, Qtail), and (jump, label) are claimed to be of form (pp, vs), i.e. vs has to contain *all* values of interpreter's static variables. But, it is reasonable to reduce this set to the only essentional ones: only variables Q, Qtail, and label can be refferenced after the program points init, cont, and jump. This can be detected by a classical static data-flow analisys aka *live (static) variables* analisys.

Efficiency: classical papers claims that $target = [\![compiler]\!]\ source$ is computed about 9-10 times as fast as $target = [\![mix]\!]\ [mix,\ source]$.

Figure 4: A mix generated compiler

```
read (Q);
  pending ← {('init , Q)}; (* A la recursion stack in recursive descent compiter; *)
  marked  ← {};            (* Also track correspondence between labels in the source *)
                           (*   and target programs *)
  while pending ≠ '() do
  |  Pick (pp, vs) ∈ pending and remore it;
  |  marked ← marked ∪ {(pp, vs)};
  |  case pp of
  |  init:
  |    Qtail ← Q;
  |    generate initial code;
  |    while Qtail ≠ '() do (* While loop from TM interpreter *)
  |    |  Instruction ← car (Qtail);
  |    |  Qtail       ← cdr (Qtail);
  |    |  case Instruction of (* TM interpreter dispatch *)
  |    |  right:
  |    |    code ← extend (code, Left  ← cons (firstsym (Right), Left);
  |    |                         Right ← cdr  (Right););
  |    |  left:
  |    |    code ← extend (code, Right ← cons (firstsym (Left), Right);
  |    |                         Left  ← cdr  (Left););
  |    |  write s:
  |    |    code ← extend (code, Right ← cons (s, cdr (Right)););
  |    |  goto label:
  |    |    Qtail ← new_tail (label, Q);
  |    |  if s goto label:
  |    |    pending ← pending ∪ ({('cont, Qtail), ('jump, label)} \ marked);
  |    |    code    ← extend (code, if s = firstsym (Right)
  |    |                            goto ('jump, label)
  |    |                            else ('cont, Qtail););
  |    |_ otherwise: error;
  |  cont: if Qtail ≠ '() goto line12; (* The first while-command *)
  |  jump: Qtail ← new_tail (label, Q);
  |        if Qtail ≠ '() goto line12;
  |  otherwise: error;
  |_ residual ← extend (residual, code);
  return residual;
```

Consider the following line of code from *mix*:

$$\text{bb} \leftarrow \textit{lookup}\ (\text{pp, program});$$

"The Trick": implement *lookup pp* as follows:

```
pp' ← pp0;                              (* pp' is static        *)
while pp != pp' do                      (* Note: pp is dynamic *)
|  pp' ← next_label_after pp';          (* pp' remains static  *)
|  bb  ← basic_block_labbeled_by pp';   (* bb  is also static  *)
|  ...                                  (* Computations involving bb *)
```

As a consequence, in the mix's residual code the FlowChart equivalent appears:

```
case pp of
|  pp0: <specialized code with respect to bb0>
|  pp1: <specialized code with respect to bb1>
|  ...:
|  ppn: <specialized code with respect to bbn>
```

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 4 contains only three of interpreter's fifteen lables.
- Pointwise division (and polyvariant division)
- Live and dead static variabels (live variables analysis)
- Remember:
  - *poly* has to be small
  - in self application check that you do not confuse variables of different mixes of the same name