1 Motivating example

We consider the following example that may benefit from partial evaluation of bytecodes:

```
let rec mymap =
  fun f -> function [] -> [] | hd :: tl -> let a = f hd in a :: mymap f tl in
let rec power =
  fun n -> fun x -> if n == 0 then 1 else x * power (n-1) x in
let raiseto =
  fun exp -> fun bases -> mymap (power exp) bases in
raiseto 3 [2; 4]
```

```
0:      BRANCH 98
---------------------------
2:      RESTART
3:      GRAB 1
5:      ACC 1
7:      BRANCHIFNOT 32
9:      ACC 1
11:     GETFIELD 0       M
13:     PUSH             Y
14:     ACC 1            M
16:     APPLY 1          A
18:     PUSH             P
19:     ACC 2
21:     GETFIELD 1
23:     PUSH
24:     ACC 2
26:     PUSH
27:     OFFSETCLOSURE 0
29:     APPLY 2
31:     PUSH
32:     ACC 1
34:     MAKEBLOCK 2
36:     POP 1
38:     RETURN 2
40:     MAKEBLOCK 0
42:     RETURN 2
---------------------------
44:     RESTART
45:     GRAB 1
47:     CONST 0
49:     PUSH
50:     ACC 1            P
52:     EQINT            O
53:     BRANCHIFNOT 5    W
55:     CONST 1          E
57:     RETURN 2         R
59:     ACC 1
61:     PUSH
62:     CONST 1
64:     PUSH
65:     ACC 2
67:     SUBINT
68:     PUSH
69:     OFFSETCLOSURE 0
71:     APPLY 2
73:     PUSH
74:     ACC 2
76:     MULINT
77:     RETURN 2
----------------------------
79:     RESTART
80:     GRAB 1
82:     ACC 1            R
84:     PUSH             A
85:     ACC 1            I
87:     PUSH             S
88:     ENVACC 2         E
90:     APPLY 1          T
92:     PUSH             O
93:     ENVACC 1
95:     APPLY 2
97:     RETURN 2
----------------------------
99:     CLOSUREREC 1, 0, -99
103:    CLOSUREREC 1, 0, -61
107:    ACC 0
109:    PUSH
110:    ACC 2
112:    CLOSURE 2, -34
115:    PUSH
116:    GETGLOBAL 0
118:    PUSH
119:    CONST 3
121:    PUSH
122:    ACC 2
124:    APPLY 2
126:    POP 1
128:    POP 1
130:    POP 1
132:    ATOM0
```

The couple of bytecodes `RESTART/GRAB` has been especially designed to handle partially applied functions. For instance, during the interpretation of the above program, the following sequences of states occur [1]:

[1] A state is a tuple that consists of a stack, a code pointer pointing to the next bytecode to execute, an accumulator, an environment, and an additional parameter. The stack is represented vertically and is growing upwards.

```
--------------------------------------------------------------------------
        B(Cst(3))
        B(Str(24))
        ...
  90*B_c(45, )*B_c(80, B_c(3, )B_c(45, ))*0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
        B(Cst(3))
        Code_p(92)
        B_c(80, B_c(3, )B_c(45, ))
        Extra_a(0)
        B(Str(24))
        ...
  45*B_c(45, )*B_c(45, )*0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
        B(Str(24))
        ...
  92*B_c(44, B_c(45, )B(Cst(3)))*B_c(80, B_c(3, )B_c(45, ))*0
--------------------------------------------------------------------------
```

...

```
--------------------------------------------------------------------------
        B(Cst(2))
        ...
  16*B_c(44, B_c(45, )B(Cst(3)))*B_c(3, )*0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
        B(Cst(2))
        Code_p(18)
        B_c(3, )
        Extra_a(0)
        ...
  44*B_c(44, B_c(45, )B(Cst(3)))*B_c(44, B_c(45, )B(Cst(3)))*0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
        B(Cst(3))
        B(Cst(2))
        Code_p(18)
        B_c(3, )
        Extra_a(0)
        ...
  45*B_c(44, B_c(45, )B(Cst(3)))*B_c(45, )*1
--------------------------------------------------------------------------
--------------------------------------------------------------------------
        B(Cst(3))
        B(Cst(2))
        Code_p(18)
        B_c(3, )
        Extra_a(0)
        ...
  47*B_c(44, B_c(45, )B(Cst(3)))*B_c(45, )*0
--------------------------------------------------------------------------
```

2 A closer look at the `power` function

The partial evaluation of the `power` function with regard to the exponent may be seen as the transformation between the two following programs:

```
let rec pow = (fun n -> fun x -> if n == 0 then 1 else x * pow (n-1) x)
in pow 3 ?
```

```
0:      BRANCH 36            26:     PUSH
2:      RESTART              27:     OFFSETCLOSURE 0
3:      GRAB 1               29:     APPLY 2
5:      CONST 0              31:     PUSH
7:      PUSH                 32:     ACC 2
8:      ACC 1                34:     MULINT
10:     EQINT                35:     RETURN 2
11:     BRANCHIFNOT 5        37:     CLOSUREREC 1, 0, -37
13:     CONST 1              41:     DCONST
15:     RETURN 2             42:     PUSH
17:     ACC 1                43:     CONST 3
19:     PUSH                 45:     PUSH
20:     CONST 1              46:     ACC 2
22:     PUSH                 48:     APPLY 2
23:     ACC 2                50:     POP 1
25:     SUBINT               52:     ATOM0
```

`let powthree = fun x -> x * x * x * 1 in powthree ?`

```
0:      BRANCH 17            15:     MULINT
2:      CONST 1              16:     RETURN 1
4:      PUSH                 18:     CLOSURE 0, -18
5:      ACC 1                21:     PUSH
7:      PUSH                 22:     DCONST
8:      ACC 2                23:     PUSH
10:     PUSH                 24:     ACC 1
11:     ACC 3                26:     APPLY 1
13:     MULINT               28:     POP 1
14:     MULINT               30:     ATOM0
```

3 A tentative schema for online pe

To lighten the transformation, we transform the code as shown below (from left to right). We can therefore reuse the back-end of the compiler.

```
0:  BRANCH 36                |       BRANCH(0)
-------------------------    |       ------------------------
2:  RESTART                  |       RESTART
3:  GRAB 1                   |       LABEL(1)
5:  CONSTINT 0               |       GRAB(1)
7:  PUSH                     |       CONST(0)
8:  ACC 1                    |       PUSH
10: EQ                       |       ACCESS(1)
11: BRANCHIFNOT 5            |       EQINT
                             |       BRANCHIFNOT(2)
-------------------------    |       ------------------------
13: CONSTINT 1               |       CONST(1)
15: RETURN 2                 |       RETURN(2)
-------------------------    |       ------------------------
17: ACC 1                    |       LABEL(2)
19: PUSH                     |       ACCESS(1)
20: CONSTINT 1               |       PUSH
22: PUSH                     |       CONST(1)
23: ACC 2                    |       PUSH
25: SUBINT                   |       ACCESS(2)
26: PUSH                     |       SUBINT
27: OFFSETCLOSURE 0          |       PUSH
29: APPLY 2                  |       OFFSETCLOSURE(0)
                             |       APPLY(2)
```

```
-------------------------       |       ------------------------
31: PUSH                        |       PUSH
32: ACC 2                       |       ACCESS(2)
34: MULINT                      |       MULINT
35: RETURN 2                    |       RETURN(2)
-------------------------       |       ------------------------
37: CLOSUREREC 1, 0, -37        |       LABEL(0)
41: DCONST                      |       CLOSUREREC(1, 0, 1)
42: PUSH                        |       CONST(2)
43: CONSTINT 3                  |       PUSH
45: PUSH                        |       CONST(3)
46: ACC 2                       |       PUSH
48: APPLY 2                     |       ACCESS(2)
                                |       APPLY(2)
-------------------------       |       ------------------------
50: POP 1                       |       POP(1)
52: ATOM0                       |       ATOM0
```

We improvise a transformation of the source code at the bytecode level. We design an operational semantics for bytecode partial evaluation.

The following sequence of blocks represents the specialization trace of partial evaluating the above program. Each block is shown with its starting code pointer, its label, its residual code, its entry and exit states according to what is sketched below.

```
---------------------------------------------------------------------------
| starting code pointer (corresponding label)                             |

residual code

  code pointer * accumulator * environment * extra arguments (entry)

                    entry stack    exit stack

  code pointer * accumulator * environment * extra arguments (exit)
|                                                                         |
---------------------------------------------------------------------------
```

The accumulator and the environment are written in terms of binding-time run-time values in which `S` means static and `N` means dynamic. We distinguish five kinds of run-time values: base constants (e.g., `B(Cst(0))`), tuples (`B_t(...)`), closures (`B_c(...)`), code pointers (e.g., `Code_p(7)`), and extra arguments parameters (e.g., `Extra_a(1)`).

From here starts the trace of partial evaluation.

```
---------------------------------------------------------------------------
|  0 (lbl =  1)                                                           |
LBL(1); BRANCH(2)
   0*S(B(Cst(0)))*S(B_t())*0


   0*S(B(Cst(0)))*S(B_t())*0
|                                                                         |
---------------------------------------------------------------------------
---------------------------------------------------------------------------
| 37 (lbl =  2)                                                           |
LBL(2); DCST; PUSH
  37*S(B(Cst(0)))*S(B_t())*0

                                  S(B(Cst(3)))
                                  N(_u1)
                                  S(B_c(3, ))
```

```
  48*S(B_c(3, ))*S(B_t())*0
|                                                                          |
----------------------------------------------------------------------------
```

The first block corresponds to the unconditional branch and the second one corresponds to the application of the power function to its arguments.

To produce the residual code, we make use of following rules:

$$\frac{m[cp] = \mathtt{CONST}\ i}{\langle st, ac, cp, en, ea\rangle \rightarrow \langle st, S(i), cp{+}2, en, ea\rangle}$$

$$\frac{m[cp] = \mathtt{DCONST}}{\langle st, ac, cp, en, ea\rangle \rightarrow \langle st, N(\_u_i), cp{+}1, en, ea\rangle}$$
$$ow := \mathtt{DCONST} :: !ow$$

$$\frac{m[cp] = \mathtt{PUSH}}{\langle st, N(\_u_i), \mathtt{cp}, en, ea\rangle \rightarrow \langle N(\_u_i)\cdot st, N(\_u_i), cp{+}1, en, ea\rangle}$$
$$ow := \mathtt{PUSH} :: !ow$$

Every time a dynamic value is newly introduced into the accumulator, we mark it with a fresh new variable (here, $\_u_i$). We naturally push run-time values when it is asked, but we output a PUSH instruction only when a dynamic value is in the accumulator.

```
----------------------------------------------------------------------------
|  3 (lbl =  3)                                                            |
LBL(3); BRANCHIFNOT(4)
   3*S(B_c(3, ))*S(B_c(3, ))*1

                      S(B(Cst(3))) S(B(Cst(3)))
                            N(_u1) N(_u1)
                     S(Code_p(50)) S(Code_p(50))
                          S(B_t()) S(B_t())
                     S(Extra_a(0)) S(Extra_a(0))
                       S(B_c(3, )) S(B_c(3, ))

  11*S(B(Cst(0)))*S(B_c(3, ))*0
|                                                                          |
----------------------------------------------------------------------------
```

$$\frac{m[cp] = \mathtt{ACC}\ i}{\begin{array}{c}\langle st[0]\cdots st[i-1]\cdot S(v)\cdot st, ac, cp, en, ea\rangle \rightarrow \\ \langle st[0]\cdots st[i-1]\cdot st, S(v), cp{+}2, en, ea\rangle\end{array}}$$

$$\frac{m[cp] = \mathtt{EQINT}}{\langle S(v2)\cdot st, S(v1), cp, en, ea\rangle \rightarrow \langle st, S(v1\ =\ v2), cp{+}1, en, ea\rangle}$$

The block we have just specialized is called a BRANCH block. It doesn't do anything but changing the contents of the accumulator and the extra arguments parameter. As it leaves the state globally unchanged, it won't part of the residual program. We'll eliminate it during the transition compression phase.

```
----------------------------------------------------------------------------
| 17 (lbl =  4)                                                            |
LBL(4); ACC(0); PUSH
  17*S(B(Cst(0)))*S(B_c(3, ))*0

                      S(B(Cst(3))) S(B(Cst(2)))
                            N(_u1) N(_u1)
```

```
                    S(Code_p(50)) S(B(Cst(3)))
                         S(B_t()) N(_u1)
                    S(Extra_a(0)) S(Code_p(50))
                      S(B_c(3, )) S(B_t())
                                  S(Extra_a(0))
                                  S(B_c(3, ))

  29*S(B_c(3, ))*S(B_c(3, ))*0
|                                                                           |
-----------------------------------------------------------------------------
```

$$\frac{m[cp] = \texttt{ACC}\ i}{\begin{array}{c}\langle st[0]\cdots st[i-1]\cdot N(\_u_i)\cdot st,\, ac,\, cp,\, en,\, ea\rangle \rightarrow\\ \langle st[0]\cdots st[i-1]\cdot N(\_u_i)\cdot st, N(\_u_i),\, cp{+}2,\, en,\, ea\rangle\\ ow := \texttt{ACC}(\mathsf{rk\_of}\ \_u_i) :: !ow\end{array}}$$

During this APPLY block, the program prepares the stack for a function application. The access to a dynamic value is translated into a new access whose access rank is the rank of the accessed variable (here, $\_u_i$) in the execution stack of the residual program. To determine this value, we maintain a second stack onto which we only push dynamic run-time values.

$$\frac{m[cp] = \texttt{OFFSETCLOSURE0}}{\begin{array}{c}let\ en = S(\mathsf{block}_{closure}(c,e))\ \ in\\ \langle st,\, ac,\, cp,\, en,\, ea\rangle \rightarrow \langle st,\, en',\, cp{+}1,\, en,\, ea\rangle\\ where\ \ en' = S(\mathsf{block}_{closure}(c+1,e))\end{array}}$$

The closure drawn from the environment is to point to the next labeled block, hence the '+1'.

```
-----------------------------------------------------------------------------
|  3 (lbl =  5)                                                             |
LBL(5); BRANCHIFNOT(6)
|                                                                           |
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
| 17 (lbl =  6)                                                             |
LBL(6); ACC(0); PUSH
|                                                                           |
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
|  3 (lbl =  7)                                                             |
LBL(7); BRANCHIFNOT(8)
|                                                                           |
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
| 17 (lbl =  8)                                                             |
LBL(8); ACC(0); PUSH
|                                                                           |
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
|  3 (lbl =  9)                                                             |
LBL(9); BRANCHIFNOT(10)
   3*S(B_c(3, ))*S(B_c(3, ))*1

                     S(B(Cst(0))) S(B(Cst(0)))
                           N(_u1) N(_u1)
                    S(Code_p(31)) S(Code_p(31))
                      S(B_c(3, )) S(B_c(3, ))
                    S(Extra_a(0)) S(Extra_a(0))
                     S(B(Cst(1))) S(B(Cst(1)))
                           N(_u1) N(_u1)
                    S(Code_p(31)) S(Code_p(31))
```

```
                         S(B_c(3, )) S(B_c(3, ))
                       S(Extra_a(0)) S(Extra_a(0))
                        S(B(Cst(2))) S(B(Cst(2)))
                             N(_u1) N(_u1)
                       S(Code_p(31)) S(Code_p(31))
                         S(B_c(3, )) S(B_c(3, ))
                       S(Extra_a(0)) S(Extra_a(0))
                        S(B(Cst(3))) S(B(Cst(3)))
                             N(_u1) N(_u1)
                       S(Code_p(50)) S(Code_p(50))
                            S(B_t()) S(B_t())
                       S(Extra_a(0)) S(Extra_a(0))
                         S(B_c(3, )) S(B_c(3, ))

  11*S(B(Cst(1)))*S(B_c(3, ))*0
|                                                                           |
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
| 13 (lbl = 10)                                                             |
LBL(10)
|                                                                           |
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
| 31 (lbl = 11)                                                             |
LBL(11); POP(1); ACC(0); PUSH; CST(1); PUSH; ACC(1); MULINT; POP(1)
  31*S(B(Cst(1)))*S(B_c(3, ))*0

                        S(B(Cst(1))) S(B(Cst(1)))
                             N(_u1) N(_u1)
                       S(Code_p(31)) S(Code_p(31))
                         S(B_c(3, )) S(B_c(3, ))
                       S(Extra_a(0)) S(Extra_a(0))
                        S(B(Cst(2))) S(B(Cst(2)))
                             N(_u1) N(_u1)
                       S(Code_p(31)) S(Code_p(31))
                         S(B_c(3, )) S(B_c(3, ))
                       S(Extra_a(0)) S(Extra_a(0))
                        S(B(Cst(3))) S(B(Cst(3)))
                             N(_u1) N(_u1)
                       S(Code_p(50)) S(Code_p(50))
                            S(B_t()) S(B_t())
                       S(Extra_a(0)) S(Extra_a(0))
                         S(B_c(3, )) S(B_c(3, ))

  35*N(_u2)*S(B_c(3, ))*0
|                                                                           |
-----------------------------------------------------------------------------
```

$$\frac{m[cp] = \texttt{MULINT}}{\langle N(\_u_i)\cdot st, N(\_u_j), cp, en, ea\rangle \to \langle st, N(\_u_k), cp{+}1, en, ea\rangle}$$
$$ow := \texttt{MULINT} :: !ow$$

$$\frac{m[cp] = \texttt{MULINT}}{\langle N(\_u_i)\cdot st, S(v_i), cp, en, ea\rangle \to \langle st, N(\_u_j), cp{+}1, en, ea\rangle}$$
$$ow := \texttt{MULINT} :: \texttt{CONST}(v_i) :: !ow$$

$$\frac{m[cp] = \texttt{MULINT}}{\langle S(v_i)\cdot st, N(\_u_i), cp, en, ea\rangle \to \langle st, N(\_u_j), cp{+}1, en, ea\rangle}$$
$$ow := \texttt{POP}(1) :: \texttt{MULINT} :: \texttt{ACCESS}(1) :: \texttt{PUSH} :: \texttt{CONST}(v_i) :: \texttt{PUSH} :: !ow$$

At the end of a RETURN block, we generate the code corresponding to the contents of the accumulator. Here, it is the result of dynamically applying the MULINT instruction.

```
---------------------------------------------------------------------------
| 31 (lbl = 11)                                                           |
LBL(11); POP(1); ACC(0); PUSH; CST(1); PUSH; ACC(1); MULINT; POP(1)
  31*S(B(Cst(1)))*S(B_c(3, ))*0

                       S(B(Cst(1))) S(B(Cst(1)))
                             N(_u1) N(_u1)
                      S(Code_p(31)) S(Code_p(31))
                        S(B_c(3, )) S(B_c(3, ))
                      S(Extra_a(0)) S(Extra_a(0))
                       S(B(Cst(2))) S(B(Cst(2)))
                             N(_u1) N(_u1)
                      S(Code_p(31)) S(Code_p(31))
                        S(B_c(3, )) S(B_c(3, ))
                      S(Extra_a(0)) S(Extra_a(0))
                       S(B(Cst(3))) S(B(Cst(3)))
                             N(_u1) N(_u1)
                      S(Code_p(50)) S(Code_p(50))
                           S(B_t()) S(B_t())
                      S(Extra_a(0)) S(Extra_a(0))
                        S(B_c(3, )) S(B_c(3, ))

  35*N(_u2)*S(B_c(3, ))*0
|                                                                         |
---------------------------------------------------------------------------
---------------------------------------------------------------------------
| 31 (lbl = 12)                                                           |
LBL(12); POP(1); PUSH; ACC(1); MULINT
  31*N(_u2)*S(B_c(3, ))*0

                       S(B(Cst(2))) S(B(Cst(2)))
                             N(_u1) N(_u1)
                      S(Code_p(31)) S(Code_p(31))
                        S(B_c(3, )) S(B_c(3, ))
                      S(Extra_a(0)) S(Extra_a(0))
                       S(B(Cst(3))) S(B(Cst(3)))
                             N(_u1) N(_u1)
                      S(Code_p(50)) S(Code_p(50))
                           S(B_t()) S(B_t())
                      S(Extra_a(0)) S(Extra_a(0))
                        S(B_c(3, )) S(B_c(3, ))

  35*N(_u3)*S(B_c(3, ))*0
|                                                                         |
---------------------------------------------------------------------------
---------------------------------------------------------------------------
| 31 (lbl = 13)                                                           |
LBL(13); POP(1); PUSH; ACC(1); MULINT
  31*N(_u3)*S(B_c(3, ))*0

                       S(B(Cst(3))) S(B(Cst(3)))
                             N(_u1) N(_u1)
                      S(Code_p(50)) S(Code_p(50))
                           S(B_t()) S(B_t())
                      S(Extra_a(0)) S(Extra_a(0))
                        S(B_c(3, )) S(B_c(3, ))

  35*N(_u4)*S(B_c(3, ))*0
|                                                                         |
---------------------------------------------------------------------------
---------------------------------------------------------------------------
| 50 (lbl = 14)                                                           |
LBL(14); POP(1); ATOM0
  50*N(_u4)*S(B_t())*0
```

```
                    S(B_c(3, ))

 52*N(_u4)*S(B_t())*0
|                                                                 |
-------------------------------------------------------------------
```

After the transition compression phase, we concatenate the residual codes, which yields the following code, semantically equivalent to the original one, despite of verbiage. One can compare this output with the bytecodes of the `powerthree` function hereafter.

```
0:      DCONST
1:      PUSH
2:      ACC 0
4:      PUSH
5:      ACC 0
7:      PUSH
8:      ACC 0
10:     PUSH
11:     POP 1
13:     ACC 0
15:     PUSH
16:     CONST 1
18:     PUSH
19:     ACC 1
21:     MULINT
22:     POP 1
24:     POP 1
26:     PUSH
27:     ACC 1
29:     MULINT
30:     POP 1
32:     PUSH
33:     ACC 1
35:     MULINT
36:     POP 1
38:     ATOM0
```

```
let powthree = fun x -> x * x * x * 1 in powthree ?

0:      BRANCH 17
2:      CONST 1
4:      PUSH
5:      ACC 1
7:      PUSH
8:      ACC 2
10:     PUSH
11:     ACC 3
13:     MULINT
14:     MULINT
15:     MULINT
16:     RETURN 1
18:     CLOSURE 0, -18
21:     PUSH
22:     DCONST
23:     PUSH
24:     ACC 1
26:     APPLY 1
28:     POP 1
30:     ATOM0
```