# Solutions to Exercise 1

**Encoding sketches**

Reliable and Interpretable Artificial Intelligence 2017
ETH Zürich

September 27, 2017

## 1 Problem 1

```
1  t    := a[0]
2  a[0] := a[0] + a[1]
3  a[1] := a[1] + a[2]
4  a[2] := a[2] + t
5  c    := b[a[0]] + b[a[1]] + b[a[2]]
```

Z3 encoding:

```
(declare-const a (Array Int Int))
(declare-const b (Array Int Int))
(declare-const t Int)
(declare-const c Int)
(declare-const a0 (Array Int Int))
(declare-const a1 (Array Int Int))
(declare-const a2 (Array Int Int))
(declare-const a3 (Array Int Int))

(assert (and (= a0 a)
             (= t  (select a0 0))
             (= a1 (store a0 0 (+ (select a0 0) (select a0 1))))
             (= a2 (store a1 1 (+ (select a1 1) (select a1 2))))
             (= a3 (store a2 2 (+ (select a2 2) t)))
             (= c  (+ (select b (select a3 0))
                      (select b (select a3 1))
                      (select b (select a3 2))))))

(assert (and (<= 0 (select a3 0)) (< (select a3 0) 64)
             (<= 0 (select a3 1)) (< (select a3 1) 64)
             (<= 0 (select a3 1)) (< (select a3 1) 64)))

(check-sat)
(get-model)
```

## 2 Problem 2

```
1  Word Pop_Count(Word x)
2  {
3      Word n;
4      for (n = ??; x {| < | = | > |} ??; n += ??) {
5          x = ?? && (?? - ??);
6      }
7      return ??;
8  }
```

Since the Z3 syntax might not be very familiar, we will first expand the sketch to a C program whose global variables correspond to unknown parameters of the sketch. For simplicity we will limit what the different holes can be filled-in with:

1. loop init/condition/increment holes can be filled with fresh constants only;

2. holes in the loop body can be filled with fresh constants or any variable in scope;

3. the hole in the final return expression can be filled with any variable in scope.

In order to make the program more readable, we will use some auxiliary functions. After we define the expanded sketch, we will translate it to the Z3 input format.

## 2.1 Unrolling the body of Pop_Count

```
1  // unrolled Pop_Count
2  Word Pop_Count(Word x)
3  {
4    Word n0 = loop_init();
5    Word x0 = x;
6    if (loop_cond(x0)) {
7      Word n1 = loop_bump(n0);
8      Word x1 = loop_body(x0);
9      if (loop_cond(x1)) {
10       Word n2 = loop_bump(n1);
11       Word x2 = loop_body(x1);
12       if (loop_cond(x2)) {
13         Word n3 = loop_bump(n2);
14         Word x3 = loop_body(x2);
15         if (loop_cond(x3)) {
16           Word n4 = loop_bump(n3);
17           Word x4 = loop_body(x3);
18           if (loop_cond(x4)) {
19             Word n5 = loop_bump(n4);
20             Word x5 = loop_body(x4);
21             if (loop_cond(x5)) {
22               Word n6 = loop_bump(n5);
23               Word x6 = loop_body(x5);
24               if (loop_cond(x6)) {
25                 Word n7 = loop_bump(n6);
26                 Word x7 = loop_body(x6);
27                 if (loop_cond(x7)) {
28                   Word n8 = loop_bump(n7);
29                   Word x8 = loop_body(x7);
30                   if (loop_cond(x8)) {
31                     Word n9 = loop_bump(n8);
32                     Word x9 = loop_body(x8);
33                     return return_expr(x9);
34                   } else { return return_expr(n8, x8); }
35                 } else { return return_expr(n7, x7); }
36               } else { return return_expr(n6, x6); }
37             } else { return return_expr(n5, x5); }
38           } else { return return_expr(n4, x4); }
39         } else { return return_expr(n3, x3); }
40       } else { return return_expr(n2, x2); }
41     } else { return return_expr(n1, x1); }
42   } else { return return_expr(n0, x0); }
43 }
```

## 2.2 Loop initialization function

```
1  // loop initialization
2  Word loop_init_param_1;
3
4  Word loop_init()
5  {
6    return loop_init_param_1;
7  }
```

## 2.3 Loop termination condition

```
1  // loop condition
2  enum LOOP_COND_EXPR { LC_EQ, LC_LT, LC_GT, };
3
4  enum LOOP_COND_EXPR loop_cond_param_1;
5  Word loop_cond_param_2;
6
7  bool loop_cond(Word x)
8  {
9    if (loop_cond_param_1 == LC_EQ) {
10     return (x == loop_cond_param_2);
11   } else
12   if (loop_cond_param_1 == LC_LT) {
13     return (x < loop_cond_param_2);
14   } else
15   if (loop_cond_param_1 == LC_GT) {
16     return (x > loop_cond_param_2);
17   } else {
18     return false;
19   }
20 }
```

## 2.4 Loop increment expression

```
1  // loop increment
2  Word loop_bump_param_1;
3
4  Word loop_bump(Word n)
5  {
6    return n + loop_bump_param_1;
7  }
```

## 2.5 Update of $x$ in the loop body

```
1  // loop body
2  enum LOOP_BODY_EXPR { LB_C, LB_N, LB_X, };
3
4  Word loop_body_expr(enum LOOP_BODY_EXPR s, Word c, Word, x, Word n)
5  {
```

```
6    if (s == LB_C) {
7      return c;
8    } else
9    if (s == LB_N) {
10     return n;
11   } else
12   if (s == LB_X) {
13     return x;
14   } else {
15     return x;
16   }
17 }
18
19 enum LOOP_BOPDY_EXPR loop_body_param_1;
20 Word loop_body_param_2;
21
22 enum LOOP_BOPDY_EXPR loop_body_param_3;
23 Word loop_body_param_4;
24
25 enum LOOP_BOPDY_EXPR loop_body_param_5;
26 Word loop_body_param_6;
27
28 Word loop_body(Word n, Word x)
29 {
30   return loop_body_expr(loop_body_param_1, loop_body_param_2)
31          && (loop_body_expr(loop_body_param_3, loop_body_param_4)
32              - loop_body_expr(loop_body_param_5, loop_body_param_6));
33 }
```

## 2.6 Return expression

```
1  // return expr
2  enum RETURN_EXPR { RET_N, RET_X, };
3
4  enum RETURN_EXPR return_expr_param_1;
5
6  Word return_expr(Word n, Word x)
7  {
8    if (return_expr_param_1 == RET_N) {
9      return n;
10   } else
11   if (return_expr_param_1 == RET_X) {
12     return x;
13   } else {
14     return x;
15   }
16 }
```

## 2.7 Z3 encoding and I/O examples

```
;; loop init
(declare-const loop-init-param!1 (_ BitVec 8))

(define-fun loop-init () (_ BitVec 8)
  loop-init-param!1)

;; loop cond
(declare-datatypes () ((LOOP-COND-EXPR LC-EQ LC-LT LC-GT)))
(declare-const loop-cond-param!1 LOOP-COND-EXPR)
(declare-const loop-cond-param!2 (_ BitVec 8))

(define-fun loop-cond ((x (_ BitVec 8))) Bool
  (ite (= loop-cond-param!1 LC-EQ) (=     x loop-cond-param!2)
  (ite (= loop-cond-param!1 LC-LT) (bvult x loop-cond-param!2)
  (ite (= loop-cond-param!1 LC-GT) (bvugt x loop-cond-param!2)
  false))))

;; loop increment
(declare-const loop-bump-param!1 (_ BitVec 8))

(define-fun loop-bump ((n (_ BitVec 8))) (_ BitVec 8)
  (bvadd n loop-bump-param!1))

;; loop body
(declare-datatypes () ((LOOP-BODY-EXPR LB-C LB-N LB-X)))

(declare-const loop-body-param!1 LOOP-BODY-EXPR)
(declare-const loop-body-param!2 (_ BitVec 8))
(declare-const loop-body-param!3 LOOP-BODY-EXPR)
(declare-const loop-body-param!4 (_ BitVec 8))
(declare-const loop-body-param!5 LOOP-BODY-EXPR)
(declare-const loop-body-param!6 (_ BitVec 8))

(define-fun loop-body-expr ((s LOOP-BODY-EXPR)
                            (c (_ BitVec 8))
                            (n (_ BitVec 8))
                            (x (_ BitVec 8)))   (_ BitVec 8)
  (ite (= s LB-C) c
  (ite (= s LB-N) n
  (ite (= s LB-X) x x))))

(define-fun loop-body ((n (_ BitVec 8)) (x (_ BitVec 8))) (_ BitVec 8)
  (bvand (loop-body-expr loop-body-param!1 loop-body-param!2 n x)
         (bvsub (loop-body-expr loop-body-param!3 loop-body-param!4 n x)
                (loop-body-expr loop-body-param!5 loop-body-param!6 n x))))

;; pop-count return
(declare-datatypes () ((RETURN-EXPR RET-N RET-X)))

(declare-const return-expr-param!1 RETURN-EXPR)

(define-fun return-expr ((n (_ BitVec 8)) (x (_ BitVec 8))) (_ BitVec 8)
```

```
  (ite (= return-expr-param!1 RET-N) n
  (ite (= return-expr-param!1 RET-X) x x)))

;; pop-count
(define-fun pop-count ((x (_ BitVec 8))) (_ BitVec 8)
  (let ((n0 loop-init) (x0 x))
  (ite (loop-cond x0)
  (let ((n1 (loop-bump n0)) (x1 (loop-body n0 x0)))
  (ite (loop-cond x1)
  (let ((n2 (loop-bump n1)) (x2 (loop-body n1 x1)))
  (ite (loop-cond x2)
  (let ((n3 (loop-bump n2)) (x3 (loop-body n2 x2)))
  (ite (loop-cond x3)
  (let ((n4 (loop-bump n3)) (x4 (loop-body n3 x3)))
  (ite (loop-cond x4)
  (let ((n5 (loop-bump n4)) (x5 (loop-body n4 x4)))
  (ite (loop-cond x5)
  (let ((n6 (loop-bump n5)) (x6 (loop-body n5 x5)))
  (ite (loop-cond x6)
  (let ((n7 (loop-bump n6)) (x7 (loop-body n6 x6)))
  (ite (loop-cond x7)
  (let ((n8 (loop-bump n7)) (x8 (loop-body n7 x7)))
  (ite (loop-cond x8)
  (let ((n9 (loop-bump n8)) (x9 (loop-body n8 x8)))
  (return-expr n9 x9))
  (return-expr n8 x8)))
  (return-expr n7 x7)))
  (return-expr n6 x6)))
  (return-expr n5 x5)))
  (return-expr n4 x4)))
  (return-expr n3 x3)))
  (return-expr n2 x2)))
  (return-expr n1 x1)))
  (return-expr n0 x0)))
)

;; i/o examples
(assert (and
  (= (pop-count #b00000000) #x00)
  (= (pop-count #b00000001) #x01)
  (= (pop-count #b00000011) #x02)
  (= (pop-count #b00000111) #x03)
  (= (pop-count #b00001111) #x04)
  (= (pop-count #b00011111) #x05)
  (= (pop-count #b00111111) #x06)
  (= (pop-count #b01111111) #x07)
  (= (pop-count #b11111111) #x08)
))

;; model
(check-sat)
(get-model)
```