Assignment
Art of Invariant Generation applied to Symbolic Bound Computation
Oregon Summer School, July 2009
Lecturer: Sumit Gulwani

1. **Precision of a Bound**
   Observe that n is a bound for each of the following procedures $P1$, $P2$, $P3$. Identify the procedures for which it is precise. Write down a precision witness for each of those procedures.

```
$P1$(uint $n$):
  if ($n \geq 100$) {
      $i$ := 0;
      while ($i < n$)
          $i$ := $i$+1;
  }
```

```
$P2$(uint $n$, uint $m$):
  $i$ := 0; $j$ := 0;
  while (($i < n$) $\wedge$ ($j < m$))
      $i$ := $i$+1;
      $j$ := $j$+1;
```

```
$P3$(uint $n$):
  if ($n\%2 == 0$) {
      $i$ := 0;
      while ($i < n$)
          $i$ := $i$+1;
  }
```

2. **Reduction of Bound Computation to Invariant Generation**
   Recall the reduction from bound computation to invariant generation using a counter c that is initialized to 0 at the beginning of the procedure, and is incremented by 1 at the control-location π of interest. We claimed the following:

   Claim 1 *If $c < F(n)$ is an invariant* that holds at π*, then* $\texttt{Max}(0, F(n))$ *is an upper bound on the number of visits to control-location π.*

   Instead, consider the following incorrect claim.

   Claim 2 *If $c < F(n)$ is an invariant* that holds at the end of the procedure*, then then* $\texttt{Max}(0, F(n))$ *is an upper bound on the number of visits to control-location π.*

   Give an example that demonstrates that Claim 2 is incorrect. Under what additional conditions would Claim 2 be valid?

3. **Symbolic Bound Computation**
   For each of the non-recursive procedures Q below, provide a bound (as precise as you can) on the number of loop iterations of the outermost loop. For each of the recursive procedures R below, provide a bound on the number of recursive procedure call invocations. In case any of the procedures is not always terminating, identify weakest conditions on the procedure inputs that ensure termination, and provide a bound under that condition.

```
Q1(int n, int m):
  x := n+1;
  while ($x \neq n$)
     if ($x \geq m$) x := 0;
     else x := x+1;
```

```
R1(Tree x):
  while ($x \neq$ Null)
     for (y := x; $y \neq$ Null; y := y.Left)
        R1(y.Right);
```

```
Q2(int x, int z, int n):
  while ($x < n$)
     if ($x < z$) x++;
     else z++;
```

```
R2(Tree x):
  if ($x \neq$ Null) {
     R2(x.Left); R2(x.Right);
     R2(x.Left); R2(x.Right);
  }
```

```
Q3(int y, int n):
  x := 0;
  while ($x < n$)
     x := x+y;
     y := y+1;
```

```
Q5(List L):
  ToDo.Init();
  L.MoveTo(L.Head(),ToDo);
  while ($\neg$ ToDo.IsEmpty())
     e := ToDo.Head();
     ToDo.Delete(e);
     foreach successor s in e.Successors()
        if (L.contains(s)) L.MoveTo(s,ToDo);
```

```
Q4(int A[], uint n):
  change := true;
  while (change)
     change := false;
     for (j:=0; $j < n - 1$; j++)
        if ($A[j] > A[j + 1]$) {
           swap($A[j], A[j + 1]$);
           change := true;
        }
```

```
Q6(Bitvector b):
  while (BitScanForward(&id1,b))
     // set all bits before id1
     b := b || ((1 << id1)-1);
     if (BitScanForward(&id2,~ b)) break;
     // reset bits before id2
     b := b & (~((1 << id2)-1));
```

In procedure $Q5$, the input list L is a list of all nodes from some graph. For any such node e, e.Successors() returns the list of all successors of e in the graph. The method L.MoveTo(s,ToDo) moves node s from list L to the list $ToDo$. The other methods have the expected semantics.

In procedure $Q6$, &, $\|$, $\sim$ denote the bitwise-and, bitwise-or, bitwise-negation operators respectively. The function BitScanForward(&id, b) returns 1 iff the bit-vector b contains a 1-bit, and sets id to the position of the least significant 1 bit present in b.

4. **Logic: Theory of Uninterpreted Functions**
Let n and m be relatively prime positive integers.

- What is the strongest atomic fact in the theory of uninterpreted functions that is implied by the formula $(y = F^n(y) \wedge y = F^m(y))$.
- What is the strongest atomic fact in the theory of uninterpreted functions that is implied by the formula $(y = F^n(y) \vee y = F^m(y))$.

5. **Logic: Nelson-Oppen Combination of Decision Procedures**
The purpose of this example is to demonstrate the importance of disjointedness condition required on theories T1 and T2 in the Nelson-Oppen combination methodology. Consider the following parity theory that shares constants (which can be treated as nullary functions) as well as the binary operators $\pm$ with the integer linear arithmetic.

$$\begin{array}{rl} \text{Expressions } e & := \ y \mid c \mid e_1 \pm e_2 \\ \text{Atomic facts } g & := \ \texttt{IsOdd}(e) \mid \texttt{IsEven}(e) \\ \text{Axioms: } & \texttt{IsOdd}(i) \text{ for any odd integer } i \\ & \forall e_1, e_2 : \texttt{IsOdd}(e_1) \wedge \texttt{IsEven}(e_2) \Rightarrow \texttt{IsOdd}(e_1 + e_2) \\ & \text{and so on.} \end{array}$$

Give an example of a formula ϕ s.t.

- ϕ is over combination of integer linear arithmetic and parity theory, (i.e., ϕ only uses the binary relation $\geq$ and unary relations `IsOdd`, `IsEven`).
- ϕ is unsatisfiable.
- Nelson-oppen combination methodology (in which we also share disjunction of equalities between variables) would fail to identify unsatisfiability.

6. **Logic: Theory of Linear Arithmetic and Farkas Lemma**
The problem of checking unsatisfiability of conjunction of linear inequalities is in PTIME. However, the polynomial time algorithms are quite involved. Instead, a worst-case exponential time algorithm called Simplex is commonly used, and it performs well in practice. Why can't we simply use Farkas Lemma to translate a set of linear inequalities into a conjunction of linear equalities over Farkas witness coefficients λ's, which can then be solved using Gaussian Elimination.

7. **Join Algorithm: Theory of Uninterpreted Functions**
The domain of conjunctions of atomic facts over the theory of uninterpreted functions is not closed under disjunction. For example, consider the following two facts:

$$\begin{array}{ll} E_1 : & x = y \\ E_2 : & x = F(x) \ \wedge \ y = F(y) \ \wedge \ G(x) = G(y) \end{array}$$

(a) The number of independent atomic facts that are implied by both E_1 and E_2 individually is infinite. Write down one such infinite family of atomic facts.

(b) The transfer function for join that we described in class for the theory of uninterpreted functions is thus not complete, i.e., it does not generate all atomic facts that are implied by each of the inputs to the join algorithm. Write down the result of the join transfer function that we studied in the class for the above example.

Aside: However, the join algorithm that we discussed in class is complete for the case when there are no cyclic dependencies like $x = F(x)$ and this leads to a PTIME algorithm for assertion checking in presence of non-deterministic conditionals since cyclic dependencies can arise only in presence of deterministic conditionals.

8. **Join Algorithm: Combination**
The combined domain of atomic facts is not closed under disjunction even if the individual domains of atomic facts are closed under disjunction. For example, consider the following two facts:

$$E_1: \quad x = 0$$
$$E_2: \quad x = 1$$

   (a) Write down the set of all atomic facts that are implied by both E_1 and E_2 individually in the theory of linear arithmetic.
   (b) Write down the set of all atomic facts that are implied by both E_1 and E_2 individually in the theory of uninterpreted functions.
   (c) The number of independent atomic facts that are implied by both E_1 and E_2 individually in the combined theory of linear arithmetic and uninterpreted functions is infinite. Write down one such infinite family of atomic facts.
   (d) The transfer function for join that we described in class for combined domain in terms of the join transfer function for individual domains is thus not complete. Write down the result of the join transfer function that we studied in the class for the above example.

   Aside: However, the join algorithm that we discussed in class is *partially* complete; it generates all atomic facts that involve terms that are semantically represented in both the inputs.

9. **Inductive Loop Invariants**
Consider the following program.

```
$a_1 := 0$; $a_2 := 0$;
$b_1 := 1$; $b_2 := F(1)$;
$c_1 := 2$; $c_2 := 2$;
while(*) {
        $a_1 := a_1 + 1$; $a_2 := a_2 + 2$;
        $b_1 := F(b_1)$; $b_2 := F(b_2)$;
        $c_1 := F(2c_1 - c_2)$; $c_2 := F(c_2)$;
}
Assert($a_2 = 2a_1$);
Assert($b_2 = F(b_1)$);
Assert($c_2 = c_1$);
```

   (a) For each assertion in the above program, write down the inductive loop invariant required to validate the assertion.
   (b) Which of these assertions can be validated by which of the following abstract domains: Difference Constraints (discussed in class), Linear Equalities (Karr, 1976), Linear Inequalities (Cousot, Halbwachs, POPL 1978), Uninterpreted Functions (Gulwani, Necula, SAS 2004; discussed in class), Combination of any two of these (Gulwani, Tiwari, PLDI 2006; discussed in class).