TermComp Proposal: Pushdown Systems as a Model for Programs with Procedures

Marc Brockschmidt Andrey Rybalchenko

Microsoft Research

June 5, 2014

Abstract

A program with procedures can be formally modelled by a set of transition relations that form a pushdown system. These transition relations keep track how valuations of variables in scope of individual procedures evolve during program execution. We present such a model together with its representation using the SMTLIB2 language.[1]

1 Pushdown systems

A pushdown system consists of a finite set of procedures Proc, a set of initial states represented by an assertion $\mathit{init}_p(v_p)$ over variables v_p (i.e., a state is a valuation of variables) of the so-called main procedure $p \in \mathit{Proc}$, a set of intra-procedural transition relations represented by assertions $\mathit{step}_p(v_p, v'_p)$ for each procedure $p \in \mathit{Proc}$, a set of call transition relations represented by assertions $\mathit{call}^p_q(v_p, v_q)$ for each procedure $p \in \mathit{Proc}$ that calls a procedure $q \in \mathit{Proc}$, and a set of return transition relations $\mathit{return}^p_q(v_p, v_q, v'_q)$ for each procedure $p \in \mathit{Proc}$ that is called by a procedure $q \in \mathit{Proc}$.

We define a (global) transition relation next as follows.

$$
\begin{aligned}
&((x \cdot xs), (y \cdot ys)) \in \mathit{next} \text{ iff} \\
&\qquad \exists p \in \mathit{Proc} : \mathit{step}_p(x, y) \land xs = ys \\
&\qquad\qquad \lor \exists q \in \mathit{Proc} : \mathit{call}^p_q(x, y) \land (x \cdot xs) = ys \\
&\qquad\qquad\qquad \lor \mathit{return}^p_q(x, z, y) \land xs = (z \cdot ys)
\end{aligned}
$$

A computation is a sequence $s_1, s_2, \ldots$ such that s_1 is an initial state and every pair of adjecent states (s, s') is an element of next.

[1] This document is intended to be sufficiently precise, and to be refined when needed.

2 Representation in SMTLIB2

The non-terminals $\langle identifier \rangle$, $\langle term \rangle$, $\langle sorted\_var \rangle$ refer to the non-terminals of the same name in the SMTLIB language. "`#`" signifies a comment (in the definition) that continues until the line end.

We make use of a special sort `Loc` to model control locations of a progam. $\langle scope \rangle$ models a tuple of program variables that are in scope of a given procedure, which includes the program counter as well as the return variable, and does not distinguish between global, local, and formal variables. To faciliate construction of control-flow graphs, we use auxiliary functions $init\_aux$, $trans2\_aux$, and $trans3\_aux$ to define initial states, as well as step, call, and return transition relations in a stylised form.

```
      ⟨scope⟩ ::=  (⟨identifier⟩ Loc) ⟨sorted_var⟩+
   ⟨init_aux⟩ ::=  (cfg_init ⟨identifier⟩ ⟨identifier⟩ ⟨term⟩)
 ⟨trans2_aux⟩ ::=  (cfg_trans2 ⟨identifier⟩ ⟨identifier⟩
                               ⟨identifier⟩ ⟨identifier⟩ ⟨term⟩)
 ⟨trans3_aux⟩ ::=  (cfg_trans3 ⟨identifier⟩ ⟨identifier⟩
                               ⟨identifier⟩ ⟨identifier⟩
                               ⟨identifier⟩ ⟨identifier⟩ ⟨term⟩)

   ⟨init_def⟩ ::=  (define-fun init_⟨procedure_id⟩ (⟨scope⟩) Bool
                     ⟨init_aux⟩)

   ⟨next_def⟩ ::=  (define-fun next_⟨procedure_id⟩ (
                                       ⟨scope⟩  # from
                                       ⟨scope⟩  # to
                                    ) Bool
                     (or ⟨trans2_aux⟩+ ))

   ⟨call_def⟩ ::=  (define-fun ⟨procedure_id⟩_call_⟨procedure_id⟩ (
                                       ⟨scope⟩  # caller
                                       ⟨scope⟩  # callee
                                    ) Bool
                     (or ⟨trans2_aux⟩+ ))

 ⟨trans3_def⟩ ::=  (define-fun ⟨procedure_id⟩_ret_⟨procedure_id⟩ (
                                       ⟨scope⟩  # callee exit
                                       ⟨scope⟩  # caller call
                                       ⟨scope⟩  # caller return
                                    ) Bool
                     (or ⟨trans3_aux⟩+ ))
```

$\langle program \rangle$ ::= (declare-sort Loc 0)

```
(declare-const ⟨identifier⟩ Loc)+
(assert (distinct ⟨identifier⟩+))
(define-fun cfg_init ( (pc Loc) (src Loc) (rel Bool) ) Bool
  (and (= pc src) rel))
(define-fun cfg_trans2 ( (pc Loc) (src Loc)
                         (pc1 Loc) (dst Loc)
                         (rel Bool) ) Bool
  (and (= pc src) (= pc1 dst) rel))
(define-fun cfg_trans3 ( (pc Loc) (exit Loc)
                         (pc1 Loc) (call Loc)
                         (pc2 Loc) (return Loc)
                         (rel Bool) ) Bool
  (and (= pc exit) (= pc1 call) (= pc2 return) rel))
⟨init_def⟩
(⟨next_def⟩ | ⟨call_def⟩ | ⟨return_def⟩)+
```

3 Examples

3.1 A while loop

We consider the following program in a C-like language:

```
L0: k = 1;
L1: while (x > 0) {
L2:   x -= k;
    }
L3:
```

We model this program as the following transition system.

```
(declare-sort Loc 0)
(declare-const loc0 Loc)
(declare-const loc1 Loc)
(declare-const loc2 Loc)
(declare-const loc3 Loc)

(assert (distinct loc0 loc1 loc2 loc3))

(define-fun cfg_init ( (pc Loc) (src Loc) (rel Bool) ) Bool
  (and (= pc src) rel))
```

```
(define-fun cfg_trans2 ( (pc Loc) (src Loc)
                         (pc1 Loc) (dst Loc)
                         (rel Bool) ) Bool
  (and (= pc src) (= pc1 dst) rel))

(define-fun init_main ( (pc Loc) (k Int) (x Int) ) Bool
  (cfg_init pc loc0 (> x 0)))

(define-fun next_main ( (pc Loc)  (k Int)  (x Int)
                        (pc1 Loc) (k1 Int) (x1 Int) ) Bool
  (or
    (cfg_trans2 pc loc0 pc1 loc1 (and (= k1 1) (= x1 x)))
    (cfg_trans2 pc loc1 pc1 loc2 (and (> x 0) (= k1 k) (= x1 x)))
    (cfg_trans2 pc loc1 pc1 loc3 (and (not (> x 0)) (= k1 k) (= x1 x)))
    (cfg_trans2 pc loc2 pc1 loc1 (and (= k1 k) (= x1 (- x k))))
  )
)
```

For comparision, we show an equivalent `int`-based TRS.

```
loc0(k, x) -> loc1(k1, x1) [ x1 = x /\ k1 = 1 ]
loc1(k, x) -> loc2(k1, x1) [ x1 = x /\ k1 = k /\ x > 0 ]
loc1(k, x) -> loc3(k1, x1) [ x1 = x /\ k1 = k /\ x-1 <= -1 ]
loc2(k, x) -> loc1(k1, x1) [ x1 = x-k /\ k1 = k ]
```

3.2 Program with procedures

As recursive example, we consider a program `mc91`:

```
    int x;

    main() {
      int y;
L5:   assume(y <= 100);
L6:   x = mc91(y);
    }  // E_MAIN:

    int mc91(int n) {
     int t;
L1:  if (n > 100) {
L2:    return n - 10;
     } else {
L3:    t = mc91(n+11);
L4:    return mc91(t);
     }
   } // E_MC91:
```

We model the above program using the following pushdown system.

```
(declare-sort Loc 0)
(declare-const l5 Loc)
(declare-const l6 Loc)
(declare-const e_main Loc)
(declare-const l1 Loc)
(declare-const l2 Loc)
(declare-const l3 Loc)
(declare-const l4 Loc)
(declare-const e_mc91 Loc)

(assert (distinct l1 l2 l3 l4 e_mc91 l5 l6 e_main))

(define-fun cfg_init ( (pc Loc) (src Loc) (rel Bool) ) Bool
  (and (= pc src) rel))

(define-fun cfg_trans2 ( (pc Loc) (src Loc)
                         (pc1 Loc) (dst Loc)
                         (rel Bool) ) Bool
  (and (= pc src) (= pc1 dst) rel))

(define-fun cfg_trans3 ( (pc Loc) (exit Loc)
                         (pc1 Loc) (call Loc)
                         (pc2 Loc) (return Loc)
                         (rel Bool) ) Bool
  (and (= pc exit) (= pc1 call) (= pc2 return) rel))

; init
(define-fun init_main ((pc Loc) (x Int) (y Int)) Bool
  (cfg_init pc l5 true))

; main
(define-fun next_main (
                        (pc Loc) (x Int) (y Int)
                        (pc1 Loc) (x1 Int) (y1 Int)
                      ) Bool
  (or
   (cfg_trans2 pc l5 pc1 l6     (and (<= y 100) (= x1 x) (= y1 y)))
   (cfg_trans2 pc l6 pc1 e_main (and (<= y 100) (= x1 x) (= y1 y)))))

(define-fun main_call_mc91 ( (pc Loc) (x Int) (y Int)
                             (pc1 Loc) (x1 Int) (n1 Int) (t1 Int) (r1 Int)
                           ) Bool
  (or
   (cfg_trans2 pc l6 pc1 l1 (and (= x1 x) (= n1 y)))))

(define-fun mc91_return_main ( (pc Loc) (x Int) (n Int) (t Int) (r Int)
                               (pc1 Loc) (x1 Int) (y1 Int)
                               (pc2 Loc) (x2 Int) (y2 Int)
                             ) Bool
```

```
  (or
   (cfg_trans3 pc e_mc91 pc1 l6 pc2 e_main (and (= x2 r) (= y2 y1)))))

; mc91
(define-fun next_mc91 ( (pc Loc) (x Int) (n Int) (t Int) (r Int)
                        (pc1 Loc) (x1 Int) (n1 Int) (t1 Int) (r1 Int)
                      ) Bool
  (or
   (cfg_trans2 pc l1 pc1 l2     (and (> n 100) (= x x1) (= n1 n)
                                     (= t1 t) (= r1 r)))
   (cfg_trans2 pc l1 pc1 l3     (and (not(> n 100)) (= x x1) (= n1 n)
                                     (= t1 t) (= r1 r)))
   (cfg_trans2 pc l2 pc1 e_mc91 (and (= x x1) (= n1 n) (= t1 t)
                                     (= r1 (- n 10))))))

(define-fun mc91_call_mc91 ( (pc Loc) (x Int) (n Int) (t Int) (r Int)
                             (pc1 Loc) (x1 Int) (n1 Int) (t1  Int) (r1 Int)
                           ) Bool
  (or
   (cfg_trans2 pc l3 pc1 l1 (and (= n1 (+ n 11)) (= x1 x)))
   (cfg_trans2 pc l4 pc1 l1 (and (= n1 t) (= x1 x)))))

(define-fun mc91_return_mc91 ( (pc Loc)  (x Int) (n Int) (t Int) (r Int)
                               (pc1 Loc) (x1 Int) (n1 Int) (t1 Int) (r1 Int)
                               (pc2 Loc) (x2 Int) (n2 Int) (t2 Int) (r2 Int)
                             ) Bool
  (or
   (cfg_trans3 pc e_mc91 pc1 l3 pc2 l4     (and (= n2 n1) (= r2 r1)
                                                (= x2 x) (= t2 r)))
   (cfg_trans3 pc e_mc91 pc1 l4 pc2 e_mc91 (and (= n2 n1) (= t2 t1)
                                                (= x2 x) (= r2 r)))))
```