Software Verification with Satisfiability Modulo Theories

Nikolaj Bjørner
Microsoft Research
SSFT 2014, Menlo Park

Contents

A primer on SMT with Z3

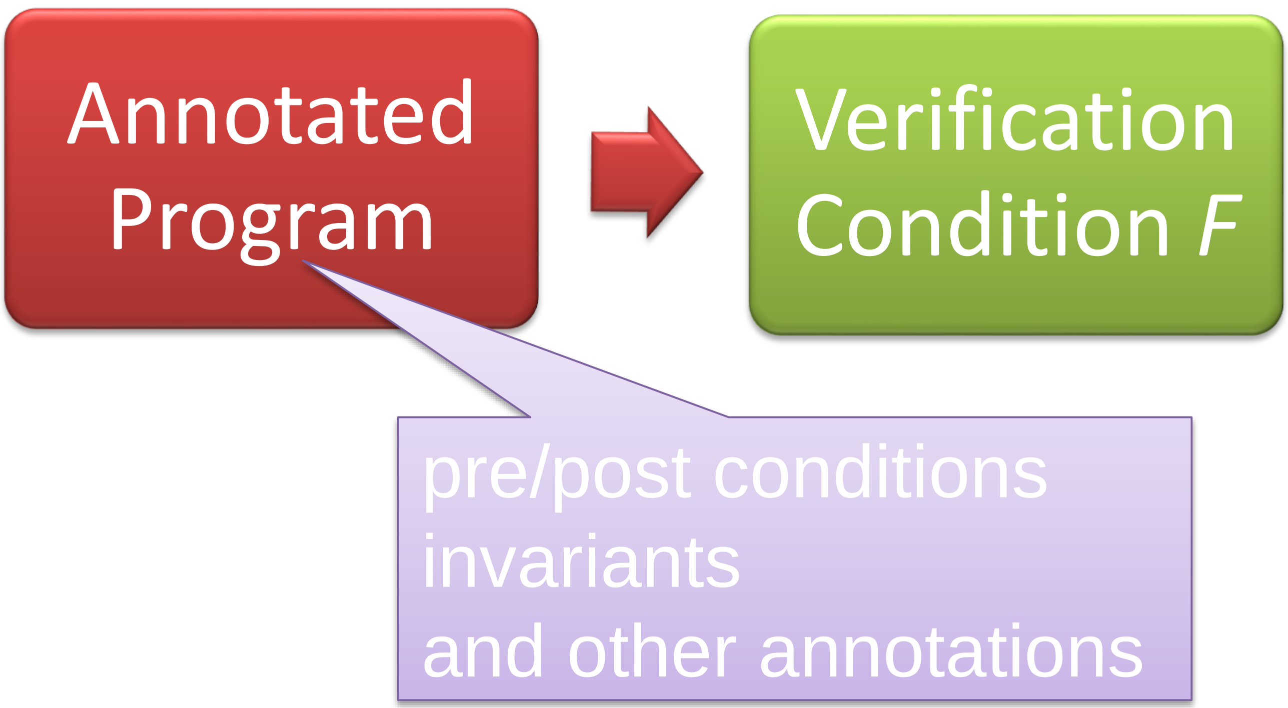
SMT & Verification by Assertion Checking

- Boogie GC, Quantifiers, Theories

SMT & Verification by Assertion Inference

- Symbolic Software Model Checking,
Horn Clauses

Verifying Compilers



Programs as Logic in Disguise

Program 1.2.1 A recursion-free program with bounded loops and an SSA unfolding.

```
int Main(int x, int y)
{
  if (x < y)
    x = x + y;
  for (int i = 0; i < 3; ++i) {
    y = x + Next(y);
  }
  return x + y;
}

int Next(int x) {
  return x + 1;
}
```

```
int Main(int x0, int y0)
{
  int x1;
  if (x0 < y0)
    x1 = x0 + y0;
  else
    x1 = x0;
  int y1 = x1 + y0 + 1;
  int y2 = x1 + y1 + 1;
  int y3 = x1 + y2 + 1;
  return x1 + y3;
}
```

$$\exists x_1, y_1, y_2, y_3 \left(\begin{array}{l} (x_0 < y_0 \implies x_1 = x_0 + y_0) \;\wedge\; (\neg(x_0 < y_0) \implies x_1 = x_0) \;\wedge \\ y_1 = x_1 + y_0 + 1 \;\wedge\; y_2 = x_1 + y_1 + 1 \;\wedge\; y_3 = x_1 + y_2 + 1 \;\wedge \\ \mathit{result} = x_1 + y_3 \end{array} \right)$$

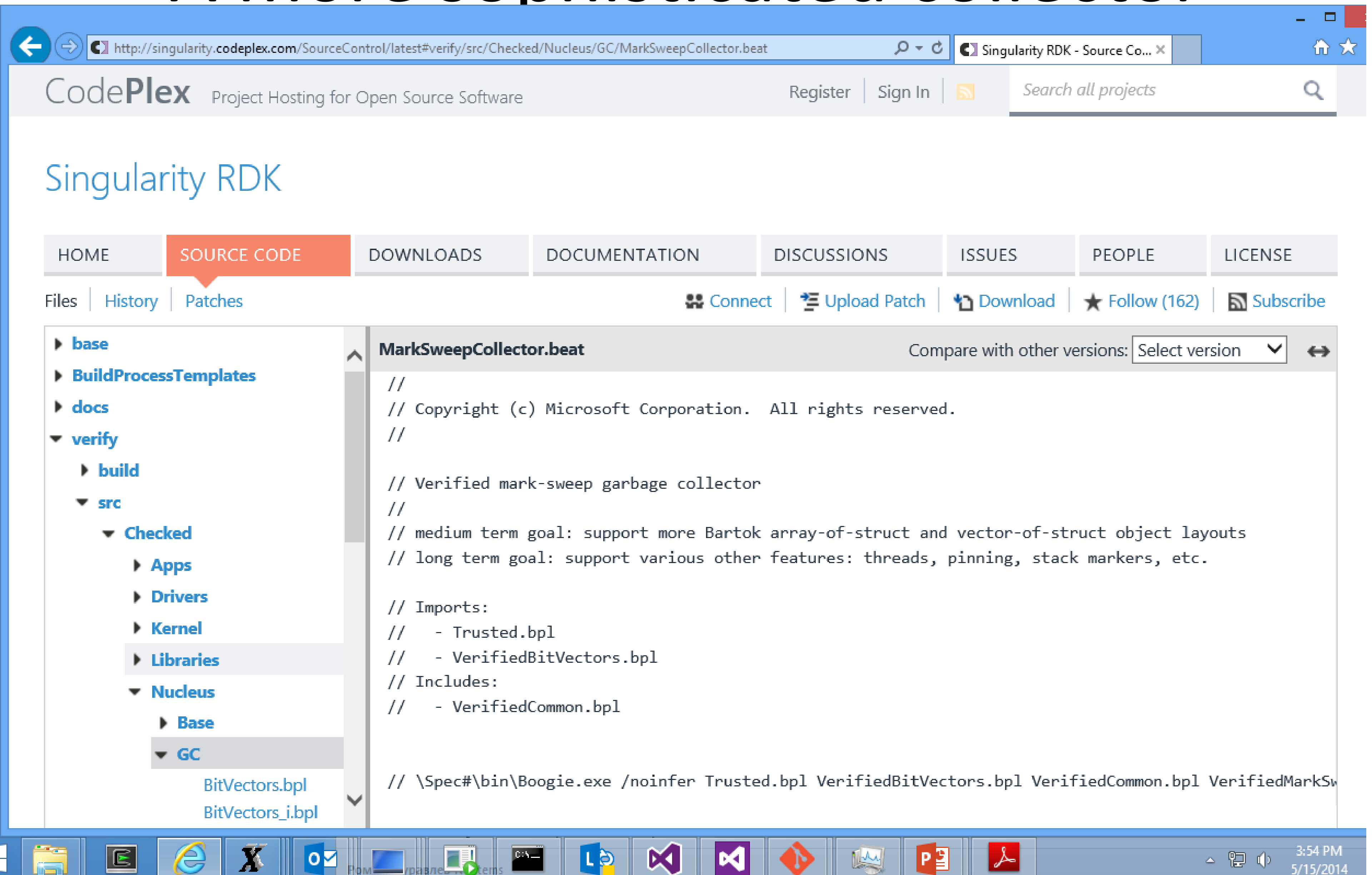
Verifying Compilers

http://rise4fun.com/Boogie

http://rise4fun.com/Dafny

A Verified GC

A more sophisticated collector



Boogie Command language

- x := E
  - x := x + 1
  - x := 10
- havoc x
- S ; T
- assert P
- assume P
- S □ T

Reasoning about execution traces

- Hoare triple: $\{ P \}\ S\ \{ Q \}$
  - Starting in P, either S diverges, or
  - Terminates safely in a state satisfying Q

- Weakest precondition:
  - $\{ wp(S, Q) \}\ S\ \{ Q \}$, and
  - If $\{ P \}\ S\ \{ Q \}$ then $P \Rightarrow wp(S, Q)$

Weakest preconditions

wp(x := E, Q) =	Q[E / x]
wp(havoc x, Q) =	$(\forall x \bullet Q)$
wp(assert P, Q) =	$P \wedge Q$
wp(assume P, Q) =	$P \Rightarrow Q$
wp(S ; T, Q) =	wp(S, wp(T, Q))
wp(S $\square$ T, Q) =	wp(S, Q) $\wedge$ wp(T, Q)

Structured if statement

if E then S else T end =

assume E; S

□

assume ¬E; T

While loop with loop invariant

```
while E
    invariant J
do
    S
end
```

where x denotes the assignment targets of S

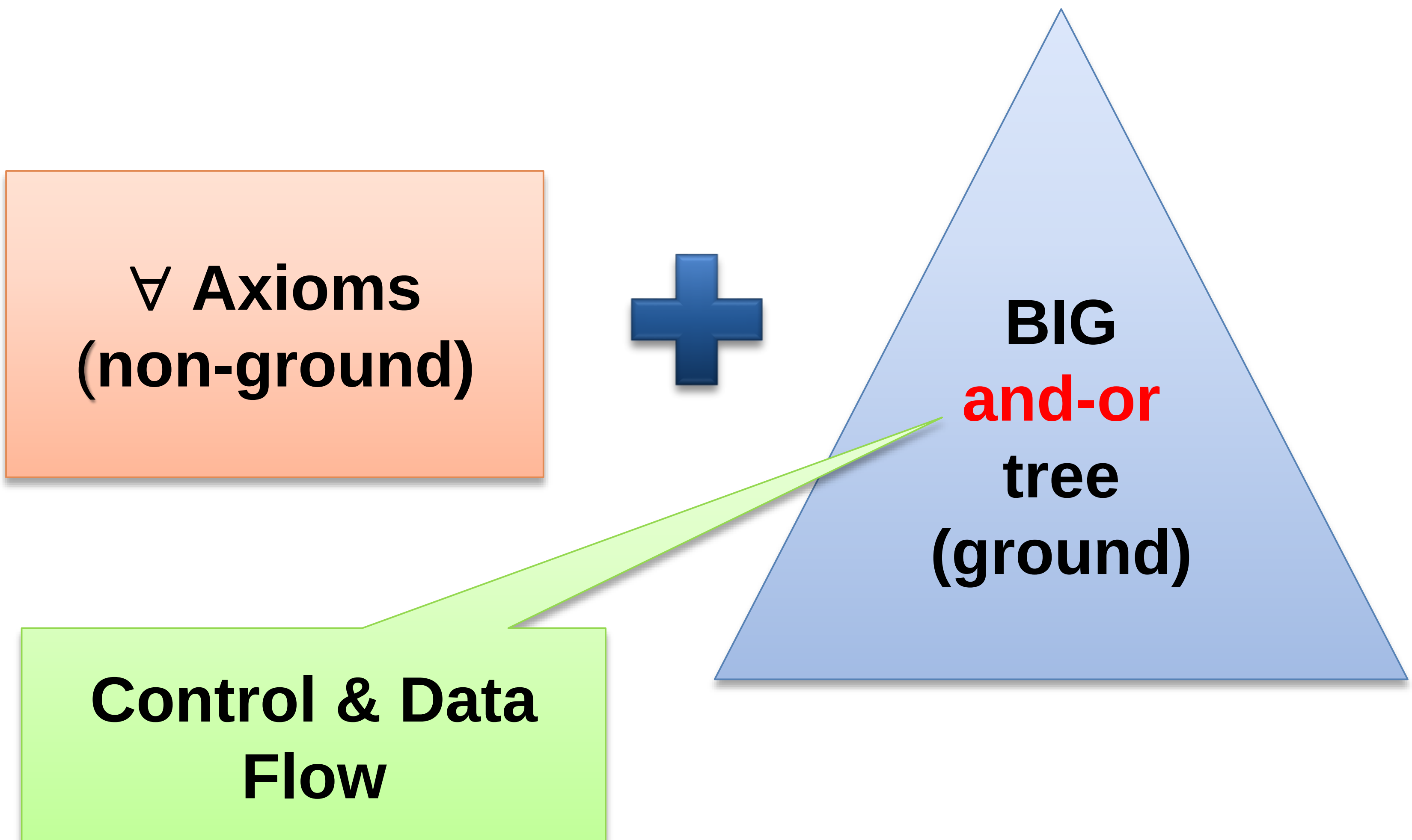
```
=   assert J;
    havoc x;  assume J;
    (   assume E;  S;  assert J;  assume false
    □   assume ¬E
    )
```

check that the loop invariant holds initially

"fast forward" to an arbitrary iteration of the loop

check that the loop invariant is maintained by the loop body

Verification conditions: Structure



Spec# Chunker.NextChunk translation

```
procedure Chunker.NextChunk(this: ref where $IsNotNull(this, Chunker)) returns ($result: ref where $IsNotNull($result, System.String));
  // in-parameter:  target object
  free requires $Heap[this, $allocated];
  requires ($Heap[this, $ownerFrame] == $PeerGroupPlaceholder || !($Heap[$Heap[this, $ownerRef], $inv] <: $Heap[this, $ownerFrame]) ||
    $Heap[$Heap[this, $ownerRef], $localinv] == $BaseClass($Heap[this, $ownerFrame])) && (forall $pc: ref :: $pc != null && $Heap[$pc, $allocated]
    && $Heap[$pc, $ownerRef] == $Heap[this, $ownerRef] && $Heap[$pc, $ownerFrame] == $Heap[this, $ownerFrame] ==> $Heap[$pc, $inv] ==
    $typeof($pc) && $Heap[$pc, $localinv] == $typeof($pc));
  // out-parameter:  return value
  free ensures $Heap[$result, $allocated];
  ensures ($Heap[$result, $ownerFrame] == $PeerGroupPlaceholder || !($Heap[$Heap[$result, $ownerRef], $inv] <: $Heap[$result, $ownerFrame]) ||
    $Heap[$Heap[$result, $ownerRef], $localinv] == $BaseClass($Heap[$result, $ownerFrame])) && (forall $pc: ref :: $pc != null && $Heap[$pc,
    $allocated] && $Heap[$pc, $ownerRef] == $Heap[$result, $ownerRef] && $Heap[$pc, $ownerFrame] == $Heap[$result, $ownerFrame] ==>
    $Heap[$pc, $inv] == $typeof($pc) && $Heap[$pc, $localinv] == $typeof($pc));
  // user-declared postconditions
  ensures $StringLength($result) <= $Heap[this, Chunker.ChunkSize];
  // frame condition
  modifies $Heap;
  free ensures (forall $o: ref, $f: name :: { $Heap[$o, $f] } $f != $inv && $f != $localinv && $f != $FirstConsistentOwner && (!IsStaticField($f) ||
    !IsDirectlyModifiableField($f)) && $o != null && old($Heap)[$o, $allocated] && (old($Heap)[$o, $ownerFrame] == $PeerGroupPlaceholder ||
    !(old($Heap)[old($Heap)[$o, $ownerRef], $inv] <: old($Heap)[$o, $ownerFrame]) || old($Heap)[old($Heap)[$o, $ownerRef], $localinv] ==
    $BaseClass(old($Heap)[$o, $ownerFrame])) && old($o != this || !(Chunker <: DeclType($f)) || !$IncludedInModifiesStar($f)) && old($o != this || $f
    != $exposeVersion) ==> old($Heap)[$o, $f] == $Heap[$o, $f]);
  // boilerplate
  free requires $BeingConstructed == null;
  free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } $o != null && !old($Heap)[$o, $allocated] && $Heap[$o, $allocated] ==>
    $Heap[$o, $inv] == $typeof($o) && $Heap[$o, $localinv] == $typeof($o));
  free ensures (forall $o: ref :: { $Heap[$o, $FirstConsistentOwner] } old($Heap)[old($Heap)[$o, $FirstConsistentOwner], $exposeVersion] ==
    $Heap[old($Heap)[$o, $FirstConsistentOwner], $exposeVersion] ==> old($Heap)[$o, $FirstConsistentOwner] == $Heap[$o,
    $FirstConsistentOwner]);
  free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } old($Heap)[$o, $allocated] ==> old($Heap)[$o, $inv] == $Heap[$o, $inv] &&
    old($Heap)[$o, $localinv] == $Heap[$o, $localinv]);
  free ensures (forall $o: ref :: { $Heap[$o, $allocated] } old($Heap)[$o, $allocated] ==> $Heap[$o, $allocated]) && (forall $ot: ref :: { $Heap[$ot,
    $ownerFrame] } { $Heap[$ot, $ownerRef] } old($Heap)[$ot, $allocated] && old($Heap)[$ot, $ownerFrame] != $PeerGroupPlaceholder ==>
    old($Heap)[$ot, $ownerRef] == $Heap[$ot, $ownerRef] && old($Heap)[$ot, $ownerFrame] == $Heap[$ot, $ownerFrame]) &&
    old($Heap)[$BeingConstructed, $NonNullFieldsAreInitialized] == $Heap[$BeingConstructed, $NonNullFieldsAreInitialized];
```

Equality-Matching

$$
\begin{aligned}
& p_{(\forall \ldots)} \\
\wedge \quad & a = g(b, b) \\
\wedge \quad & b = c \\
\wedge \quad & f(a) \neq c \\
\wedge \quad & p_{(\forall x \ldots)} \rightarrow f\big(g(c, b)\big) = b
\end{aligned}
$$

$g(c, x)$ matches $g(b, b)$
with substitution $[x \mapsto b]$
modulo $b = c$

[de Moura, B. CADE 2007]

Reachability and EPR

struct *cell* {
  int *data;*
  cell next;*
};

void $zero(cell * c)$ {
  $\mathbf{while}(c)\{\, c \rightarrow data = 0; c = c \rightarrow next; \}$

  $\mathbf{assert}\ \left(\forall d \in c_{old} \xrightarrow{*} next\,.\, d = null \vee d \rightarrow data = 0\right);$
}

[Itzhaky et.al. CAV 13, CAV 14, POPL 14]

Reachability and EPR

$\text{void } zero(cell * c)\ \{$

$\quad \textbf{while}(c)\{\ c \rightarrow data = 0; c = \ c \rightarrow next; \}$

$\quad \textbf{assert}\ \left(\forall d \in c_{old} \stackrel{*}{\rightarrow} next\ .\ d = null \vee d \rightarrow data = 0\right);$

$\}$

Classical memory model:

$Next: Cell \rightarrow Cell$
$Data: Cell \rightarrow \textbf{int}$

$wp(c = c \rightarrow next, Q) \coloneqq Q[Next(c)/c]$

$Next^{*}: Cell \times Cell \rightarrow Bool \quad \coloneqq TC(Next)$

Reachability and EPR

$\text{void } zero(cell * c) \{$

$\quad \textbf{while}(c)\{ c \rightarrow data = 0; c = c \rightarrow next; \}$

$\quad \textbf{assert} \left(\forall d \in c_{old} \overset{*}{\rightarrow} next \,.\, d = null \vee d \rightarrow data = 0 \right);$

$\}$

Memory model based on $Next^*$

$Next^*: Cell \times Cell \rightarrow Bool$
$Data: Cell \times \textbf{int} \rightarrow Bool$

$Next^*$ *is* *Transitive, Reflexive, Linear , Anti-symmetric for acyclic lists*
$Next^+(c,d) \coloneqq c \neq d \wedge Next^*(c,d)$
$Next^!(c,d) \coloneqq Next^+(c,d) \wedge \forall e. Next^+(c,e) \rightarrow Next^*(d,e)$

$$wp(d = c \rightarrow next, Q) \coloneqq \forall e\, Next^!(c,e) \rightarrow Q[e/d]$$

Reachability and EPR

$Next^*$ *is* *Transitive, Reflexive, Linear, Anti-symmetric*

$Next^+(c,d) \coloneqq c \neq d \land Next^*(c,d)$

$Next^!(c,d) \coloneqq Next^+(c,d) \land \forall e. Next^+(c,e) \rightarrow Next^*(d,e)$

$$wp(d = c \rightarrow next, Q) \coloneqq$$
$$\forall e\,.\,Next^!(c,e) \rightarrow Q[e/d] \quad \land alloc(c) \land c \neq null$$

$$wp(c \rightarrow next \ = null, Q) \coloneqq$$
$$Q[\lambda ab. Next^*(a,b) \land (Next^*(a,c) \rightarrow Next^*(b,c))/Next^*]$$

$$wp(c \rightarrow next = d, Q) \coloneqq$$
$$Q[\lambda ab. Next^*(a,b) \lor (Next^*(a,c) \land Next^*(d,b))/Next^*]$$

Assuming $c \rightarrow next = d;$ *is preceded by* $c \rightarrow next = null$

Reachability and EPR

- Verification
  - *Python exercise:* implement *wp* for $Next^*$
- Synthesizing Inductive Invariants
  - [Itzhaky et.al CAV 14] uses Predicate Abstraction for EPR.

Verification by Assertion Inference

Horn Clauses

mc(x) = x-10 if $x > 100$
mc(x) = mc(mc(x+11)) if $x \leq 100$

assert $(x \leq 101 \rightarrow mc(x) = 91)$

$$\forall X.\ X > 100 \rightarrow mc(X, X-10)$$
$$\forall X, Y, R.\ X \leq 100 \wedge mc(X+11, Y) \wedge mc(Y, R) \rightarrow mc(X, R)$$
$$\forall X, R.\ mc(X, R) \wedge X \leq 101 \rightarrow R = 91$$

Solver finds solution for mc

[Hoder, B. SAT 2012]

Transition System

- V - program variables
- init(V) - initial states
- step(V, V') - transition relation
- safe(V) - safe states

Safe Transition System

$\exists Inv.$

- $\forall V$. init(V) $\rightarrow Inv(V)$
- $\forall V, V'$. $Inv(V) \wedge$ step(V, V') $\rightarrow Inv(V')$
- $\forall V$. safe(V)$\rightarrow Inv(V)$

  – [Rybalchenko et.al. PLDI 2012, POPL 2014] Termination and reactivity are also handled in framework of solving systems of logical formulas.

Recursive Procedures

Formulate as Horn clauses:

$$\forall X.\ X > 100 \rightarrow \mathsf{mc}(X, X-10)$$
$$\forall X, Y, R.\ X \leq 100 \wedge \mathsf{mc}(X+11, Y) \wedge \mathsf{mc}(Y, R) \rightarrow \mathsf{mc}(X, R)$$
$$\forall X, R.\ \mathsf{mc}(X, R) \wedge X \geq 101 \rightarrow R = 91$$

***Solve for* mc**

Recursive Procedures

Formulate as Predicate Transformer:

$$\mathscr{F}(\textbf{mc})(X,R) = \begin{array}{c} X > 100 \land R = X - 10 \\ \lor\ X \leq 100 \land \exists Y.\, \textbf{mc}(X+11, Y) \land \textbf{mc}(Y, R) \end{array}$$

Check: $\mu\mathscr{F}(\textbf{mc})(X,R) \land X \geq 101 \rightarrow R = 91$

Recursive Procedures

Instead of computing $\mu\mathscr{F}(\textbf{mc})(X,R)$,
then checking $\mu\mathscr{F}(\textbf{mc})(X,R) \land X \leq 101 \rightarrow R = 91$

Suffices to find post-fixed point $\textbf{mc}_{post}$ **satisfying:**

$$\forall \boldsymbol{X}, \boldsymbol{R}. \quad \mathscr{F}(\textbf{mc}_{\boldsymbol{post}})(\boldsymbol{X}, \boldsymbol{R}) \rightarrow \textbf{mc}_{\boldsymbol{post}}(\boldsymbol{X}, \boldsymbol{R})$$

$$\forall \boldsymbol{X}, \boldsymbol{R}. \quad \textbf{mc}_{\boldsymbol{post}}(\boldsymbol{X}, \boldsymbol{R}) \land \boldsymbol{X} \leq 101 \rightarrow \boldsymbol{R} = 91$$

Program Verification as SMT
- aka
A Crusade for Horn*ish* Satisfaction

Program Verification (Safety)

as Solving fixed-points

as Satisfiability of Horn clauses

[Bjørner, McMillan, Rybalchenko, SMT workshop 2012]
Hilbert Sausage Factory: [Grebenshchikov, Lopes, Popeea, Rybalchenko, PLDI 2012]

A model checking Example

Program 1.4.1 Processing requests using locks.

```
do {
        lock();
        old_count = count;
        request = GetNextRequest();
        if (request != NULL) {
            ReleaseRequest(request);
            unlock();
            ProcessRequest(request);
            count = count + 1;
        }
}
while (old_count != count);
unlock();
```

Abstraction as Boolean Program

Program 1.4.2 Processing requests using locks, abstracted.

```
do {
      lock();
      b = true;
      if (*) {
         unlock();
         if (b) {
            b = false;
         }
         else {
           havoc b;
         }
      }
}
while (!b);
unlock();
```

b := count == old_count

[SLAM, BLAST, Graf & Saidi, Uribe, ..]

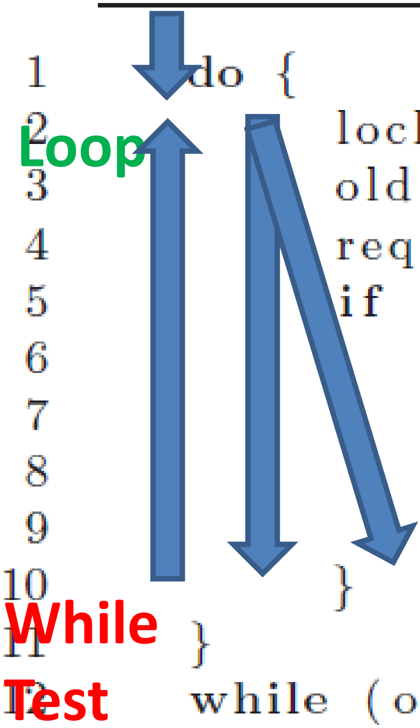
(Predicate) Abstraction/Refinement

- SMT solver used to synthesize (strongest) abstract transition relation *F:*

$$\rho(\vec{x}, \vec{x}') \Rightarrow \;\; F(b_1(\vec{x}) \ldots, b_n(\vec{x}), b_1(\vec{x}') \ldots, b_n(\vec{x}'))$$

Control as Horn Clauses

Program 1.4.1 Processing requests




```
do {
        lock();
        old_count = count;
        request = GetNextRequ
        if (request != NULL)
            ReleaseRequest(req
            unlock();
            ProcessRequest(req
            count = count + 1;
        }
}
while (old_count != count)
unlock();
```

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state) (WhileTest count count true))))

; Loop with if-test
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state) (WhileTest (+ 1 count ) count false))))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (not (= old_count count )) (WhileTest count old_count lock_state))
      (Loop count old_count lock_state))))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (= old_count count ) (WhileTest count old_count lock_state))
       (= lock_state true))))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
      (= lock_state false))))

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

Solving Horn Clauses

Pre-processing

$$HornClauses \rightarrow \ HornClauses'$$

Search

– Find model M such that $M \vDash HornClauses$

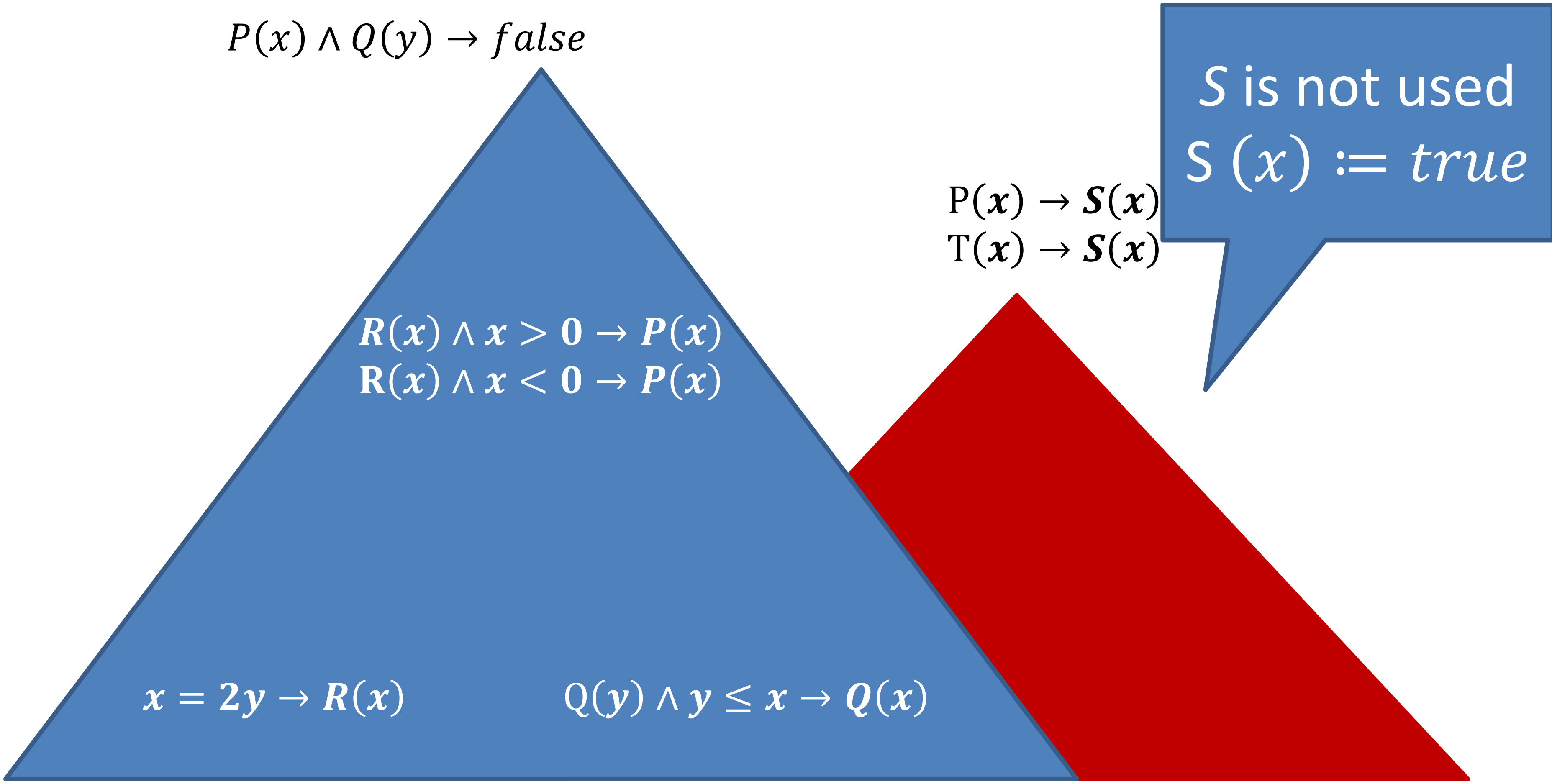
Or

– Find refutation proof π: $HornClauses \vdash_{\pi} \bot$

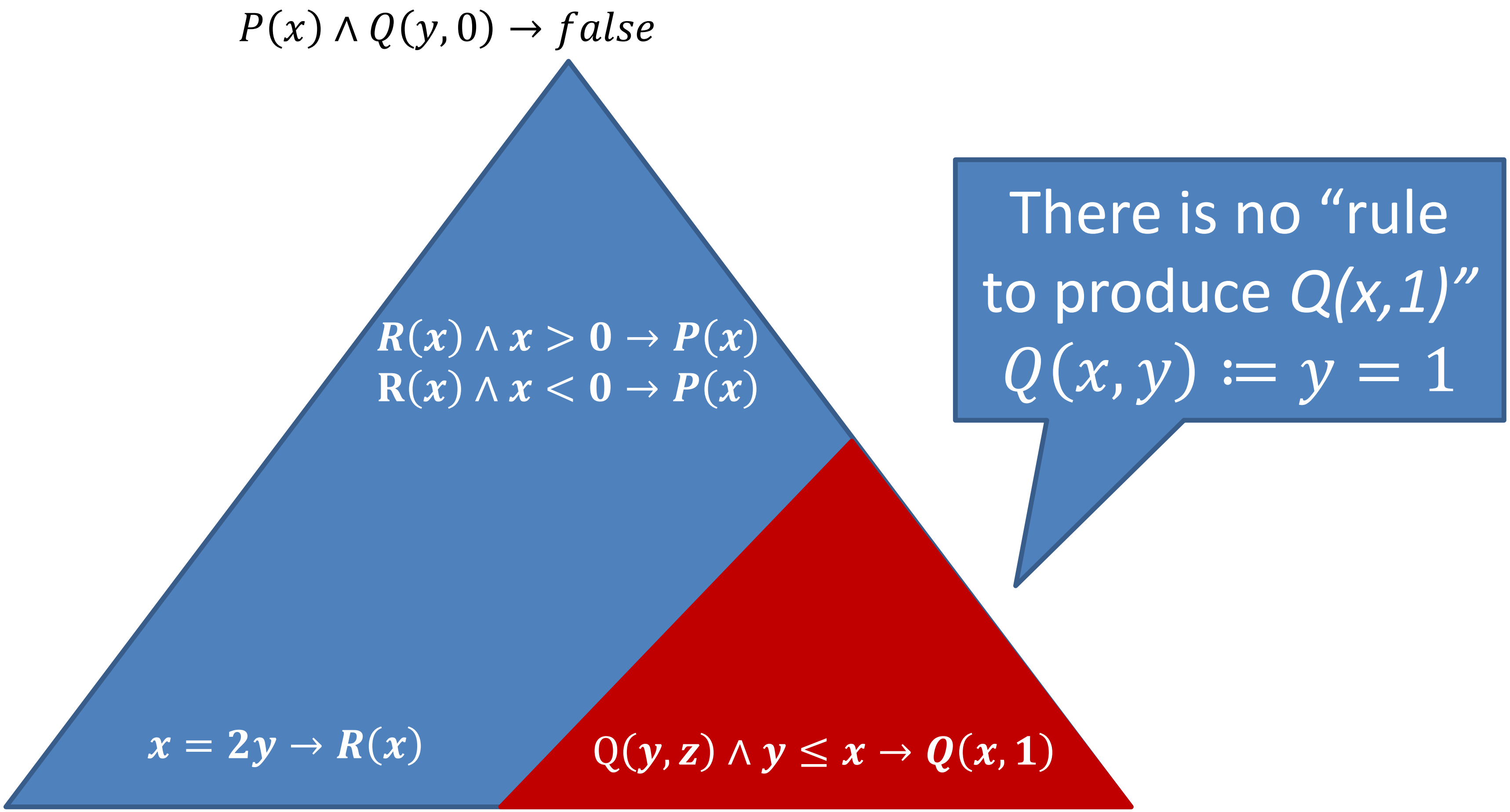
Pre-processing

- Cone of Influence
- Simplification
- Subsumption
- Inlining
- Slicing
- Unfolding

Cone of Influence – top down



Cone of Influence – bottom up



Inlining

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state) (WhileTest count count true))))

; Loop with if-test
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state) (WhileTest (+ 1 count ) count false))))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (not (= old_count count )) (WhileTest count old_count lock_state))
      (Loop count old_count lock_state))))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (= old_count count ) (WhileTest count old_count lock_state))
      (= lock_state true))))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
      (= lock_state false))))

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

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= count count)))
                (Loop count count true)))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= count count))
                (= true true)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= (+ 1 count ) count ))
(Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= (+ 1 count ) count )
                (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
      (= lock_state false))))

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

Simplification

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= count count)))
                    (Loop count count true))))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= count count))
                    (= truetrue)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= (+ 1 count ) count ))
(Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= (+ 1 count ) count )
                    (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
     (= lock_state false))))

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

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=>  (Loop count old_count lock_state)
                    (= true true)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
                    (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
     (= lock_state false))))

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

Simplification

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= count count)))
                    (Loop count count true))))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= count count))
                    (= truetrue)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= (+ 1 count ) count ))
(Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= (+ 1 count ) count )
                    (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
     (= lock_state false))))

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

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop

; Loop without if test + loop exit

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
                    (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count true)
     false)))

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

Cone of Influence

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= count count)))
              (Loop count count true)))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= count count))
              (= truetrue)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= (+ 1 count ) count ))
(Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= (+ 1 count ) count )
              (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
     (= lock_state false))))

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

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop

; Loop without if test + loop exit

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
              (Loop (+ 1 count) count false)))

; Loop with if-test + loop exit

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count true)
     false)))

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

Result

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count
(assert (forall ((count Int) (old_count Int))
    (Loop count old_count false)))

; Loop without if test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= count count)))
                (Loop count count true)))

; Loop without if test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= count count))
                (= truetrue)))

; Loop with if-test + repeat loop
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (not (= (+ 1 count ) count ))
(Loop (+ 1 count) count false)))

; Loop with if-test + loop exit
(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (and (Loop count old_count lock_state) (= (+ 1 count ) count )
                (= false true)))

(assert (forall ((count Int) (old_count Int) (lock_state Bool))
  (=> (Loop count old_count lock_state)
     (= lock_state false))))

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

```
(set-logic HORN)
(declare-fun Loop (Int Int Bool) Bool)
(declare-fun WhileTest (Int Int Bool) Bool)

; Loop is entered in arbitrary values of count, old_count

; Loop without if test + repeat loop

; Loop without if test + loop exit

; Loop with if-test + repeat loop

; Loop with if-test + loop exit

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

IC3/PDR: Property Directed Reachability

The IC3 Algorithm for Symbolic Model Checking by Aaron Bradley

Procedures	*Regular vs. Push Down systems* As a Conflict-driven solver for recursive Horn clauses
Beyond Propositional Logic	*Linear Real Arithmetic* - *Timed Automata Decision Procedure* - *Interpolants from models*

[SAT 2012. Kryštof Hoder & Nikolaj Bjørner]

PDR – the algorithm

Objective is to solve for *R* such that

$$\mathscr{F}(R)(X) \rightarrow R(X), \quad R(X) \rightarrow Safe(X), \ \forall X$$

Key elements of PDR algorithm:

Over-approximate reachable states
$R_0 := \mathscr{F}(\text{false}), R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_N := \text{true}$
Propagate back from $\neg Safe$
Resolve conflicts
Strengthen/propagate using induction

PDR – the algorithm

Objective is to solve for *R* such that

$$\mathscr{F}(R)(X) \rightarrow R(X), \quad R(X) \rightarrow Safe(X), \ \forall X$$

Initialize:

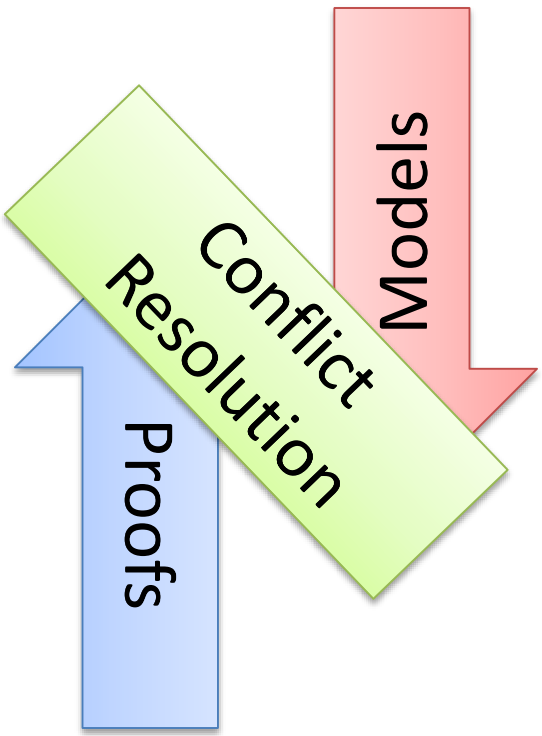
$$Safe \quad \nwarrow \quad R_0 := \mathscr{F}(false) \quad \nearrow \quad R_1 := true \quad \nwarrow \quad \mathscr{F}(R_0)$$

Main invariant:

$$Safe \quad \nwarrow \quad R_i \quad \nearrow \quad R_{i+1} \quad \nwarrow \quad \mathscr{F}(R_i)$$

A digression

Dualities – Recurring Theme



Core DPLL(T) engine

Fixed Points engine

Nonlinear solver

Linear Integer solver

Core Engine in Z3: Modern DPLL/CDCL

Group	Rule	Transition	Condition
	Initialize	$\epsilon \mid F$	F *is a set of clauses*
Model	Decide	$M \mid F \implies M, \ell \mid F$	ℓ *is unassigned*
Model	Propagate	$M \mid F, C \vee \ell \implies M, \ell^{C \vee \ell} \mid F, C \vee \ell$	C *is false under* M
Model	Sat	$M \mid F \implies M$	F *true under* M
Proof	Conflict	$M \mid F, C \implies M \mid F, C \mid C$	C *is false under* M
Proof	Learn	$M \mid F \mid C \implies M \mid F, C \mid C$	
Proof	Unsat	$M \mid F \mid \emptyset \implies Unsat$	
Conflict Resolution	Backjump	$MM' \mid F \mid C \vee \ell \implies M\ell^{C \vee \ell} \mid F$	$\neg\ell \in M', \quad M' \cap \neg C = \emptyset$
Conflict Resolution	Resolve	$M \mid F \mid C' \vee \neg\ell \implies M \mid F \mid C' \vee C$	$\ell^{C \vee \ell} \in M$
	Restart	$M \mid F \implies \epsilon \mid F$	
	Forget	$M \mid F, C \implies M \mid F$	C *is a learned clause*

[Nieuwenhuis, Oliveras, Tinelli J.ACM 06] customized

DPLL(*T*) solver interaction

T- Propagate $M \mid F, C \vee \ell \implies M, \ell^{C \vee \ell} \mid F, C \vee \ell$ $C\ is\ false\ under\ T + M$

T- Conflict $M \mid F \implies M \mid F \mid \neg M'$ $M' \subseteq M\ and\ M'\ is\ false\ under\ T$

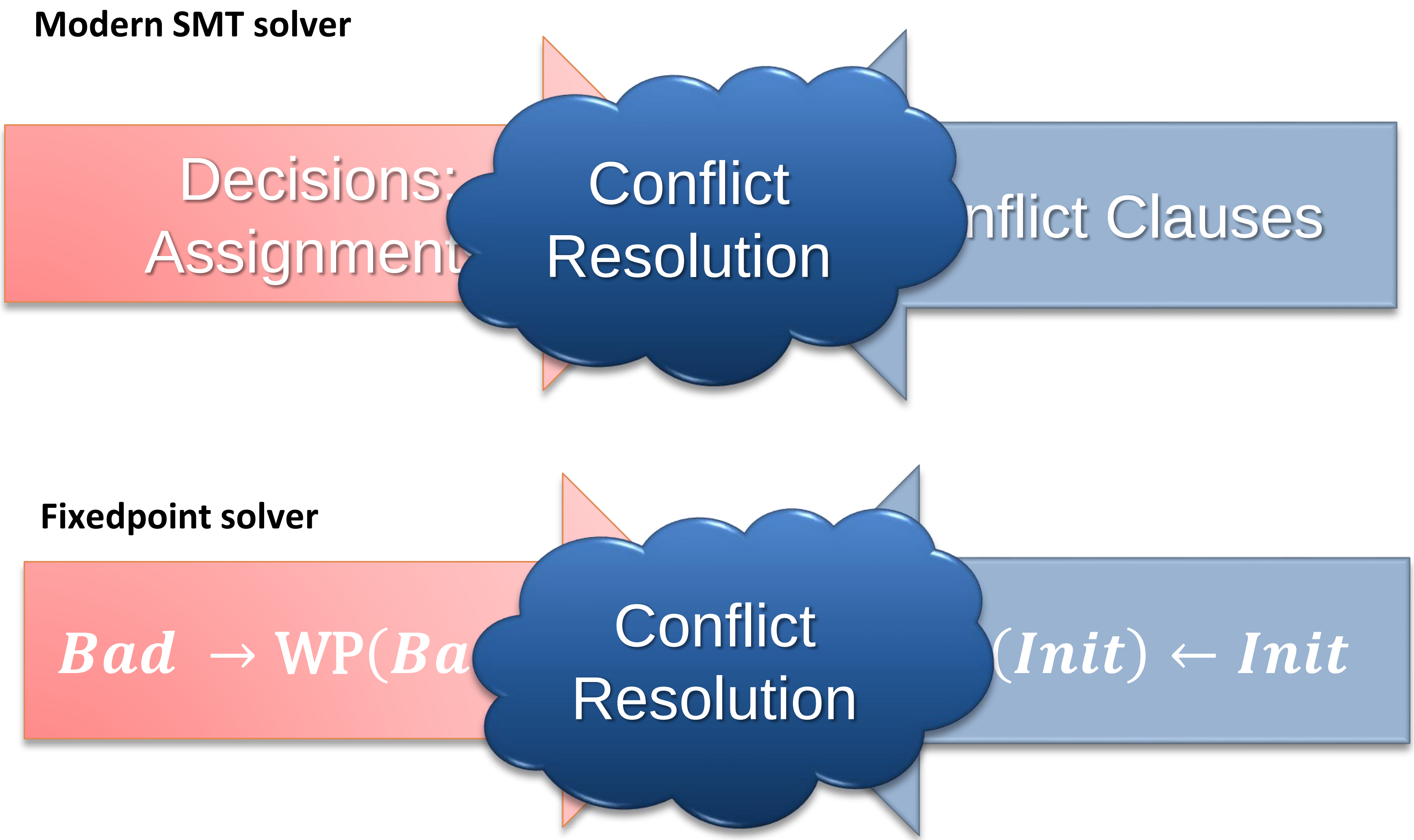
T- Propagate $a > b, b > c \mid F, a \leq c \vee b \leq d \implies$

$a > b, b > c, b \leq d^{a \leq c \vee b \leq d} \mid F, a \leq c \vee b \leq d$

T- Conflict $M \mid F \implies M \mid F,\ a \leq b \vee b \leq c \vee c < a$

$where\ a > b, b > c, a \leq c \subseteq M$

Search: Mile-high perspective



Conflict resolution with arithmetic

$$
\text{initially } y_1 := y_2 := 0;
$$

$$
P_1 :: \left[\begin{array}{l}\text{loop forever do}\\ \left[\begin{array}{l}\ell_0 : y_1 := y_2 + 1;\\ \ell_1 : \text{await } y_2 = 0 \vee y_1 \le y_2;\\ \ell_2 : \text{critical};\\ \ell_3 : y_1 := 0;\end{array}\right]\end{array}\right] \;||\; P_2 :: \left[\begin{array}{l}\text{loop forever do}\\ \left[\begin{array}{l}\ell_0 : y_2 := y_1 + 1;\\ \ell_1 : \text{await } y_1 = 0 \vee y_2 \le y_1;\\ \ell_2 : \text{critical};\\ \ell_3 : y_2 := 0;\end{array}\right]\end{array}\right]
$$

R(0,0,0,0).
T(L,M,Y1,Y2,L',M',Y1',Y2')∧R(L,M,Y1,Y2) → R(L',M
R(2,2,Y1,Y2) → false

Mutual Exclusion

⟺

Clauses have model

Step(L,L',Y1,Y2,Y1') → T(L,M,Y1,Y2,L',M,Y1',Y2)
Step(M,M',Y2,Y1,Y2') → T(L,M,Y1,Y2,L,M',Y1,Y2')

P_2 takes a step

Step(0,1,Y1,Y2,Y2+1).
(Y1 ≤ Y2 ∨ Y2 = 0) → Step(1,2,Y1,Y2,Y1).
Step(2,3,Y1,Y2,Y1).
Step(3,0,Y1,Y2,0).

$\ell_0 : y := \hat{y} + 1;\ goto\ \ell_1$

$\ell_1 : await\ \hat{y} = 0 \vee y \le \hat{y}\ ;\ goto\ \ell_2$

$\ell_2 : critical\ ;\ goto\ \ell_3$

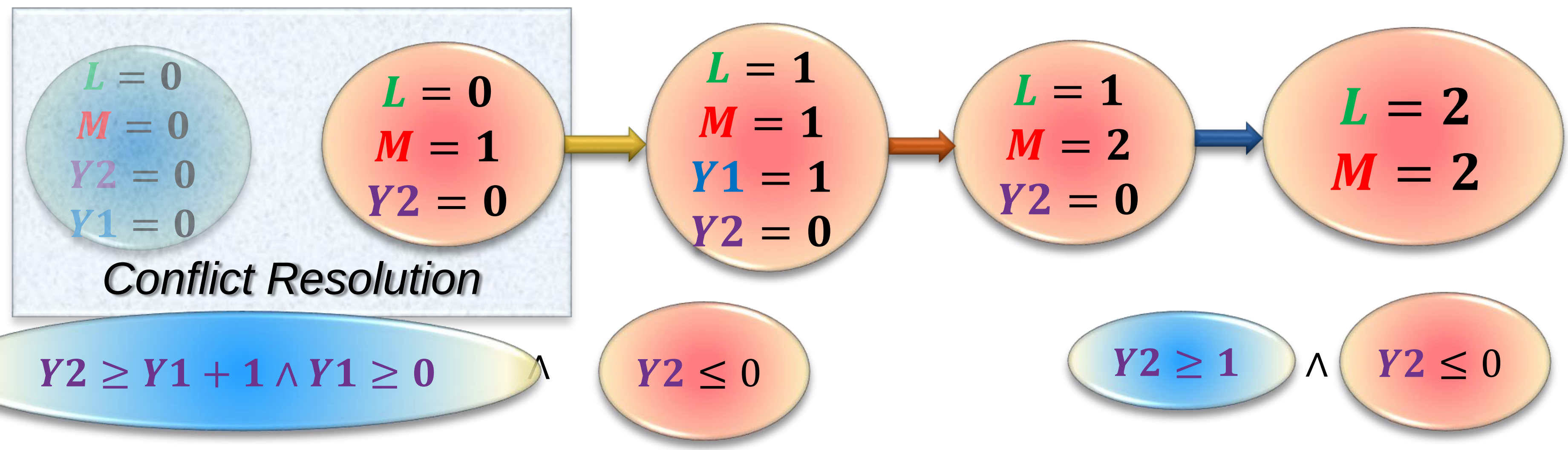
$\ell_3 : y\ := 0;\ goto\ \ell_0$

Search: Mile-high perspective

PDR(T): Conflict Resolution

initially $y_1 := y_2 := 0;$

$$P_1 :: \left[\begin{array}{l} \text{loop forever do} \\ \left[\begin{array}{l} \ell_0 : y_1 := y_2 + 1; \\ \ell_1 : \text{await } y_2 = 0 \lor y_1 \leq y_2; \\ \ell_2 : \text{critical}; \\ \ell_3 : y_1 := 0; \end{array} \right] \end{array} \right] \;||\; P_2 :: \left[\begin{array}{l} \text{loop forever do} \\ \left[\begin{array}{l} \ell_0 : y_2 := y_1 + 1; \\ \ell_1 : \text{await } y_1 = 0 \lor y_2 \leq y_1; \\ \ell_2 : \text{critical}; \\ \ell_3 : y_2 := 0; \end{array} \right] \end{array} \right]$$



Conflict

Resolution

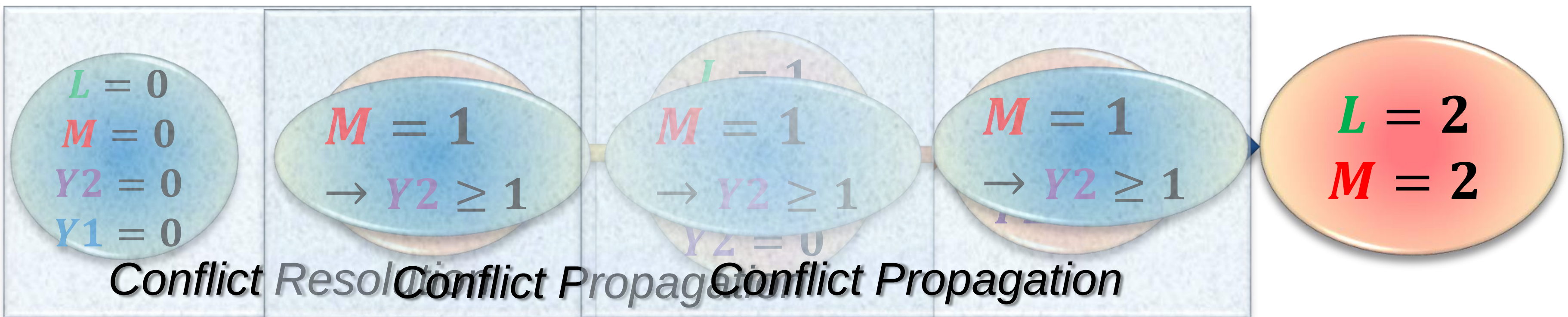
Get Generalization from Farkas Lemma

Eg., resolve away blue internal variables

PDR(T): Conflict Resolution

initially $y_1 := y_2 := 0$;

$$P_1 :: \left[\begin{array}{l}\text{loop forever do}\\ \left[\begin{array}{l}\ell_0 : y_1 := y_2 + 1;\\ \ell_1 : \text{await } y_2 = 0 \lor y_1 \leq y_2;\\ \ell_2 : \text{critical};\\ \ell_3 : y_1 := 0;\end{array}\right]\end{array}\right] \; || \; P_2 :: \left[\begin{array}{l}\text{loop forever do}\\ \left[\begin{array}{l}\ell_0 : y_2 := y_1 + 1;\\ \ell_1 : \text{await } y_1 = 0 \lor y_2 \leq y_1;\\ \ell_2 : \text{critical};\\ \ell_3 : y_2 := 0;\end{array}\right]\end{array}\right]$$



IC3/PDR – some observations

Interpolation $\cong$ Solution to Horn Clauses [Rybalchenko]

- $\forall x, y. A[x, y] \Rightarrow I(x), \quad \forall x, z. I(x) \Rightarrow B[x, z]$
- Instead of mining interpolants from *proofs,*
  PDR uses *models* and *cores*

Timed push-down systems $\cong$ PDR for difference arithmetic

Property Directed Polyhedral Abstraction $\cong$ PDR + *Cute* Interpolants
[Ongoing with Arie Gurfinkel]

Shape analysis $\cong$ PDR with EPR + Predicate Abs/Zipper Interpolants
[Ongoing: Gurfinkel, Itzhaky , Korovin, Lahav, Reps, Talur, Sagiv]

Property + Reachability Directed [CAV 14, Komuravelli, Chaki, Gurfinkel]

High-level Takeaways

- Program Analysis as Solving Logical Formulas
  - I presented some samples of *encoding* analysis problems into logic.
  - I gave a taste of *solving* algorithms for some classes of logical formulas.

SMT SOLVING

DPLL(T) BASED APPROACH

SMT : Basic Architecture

Case Analysis

- Equality + UF
- Arithmetic
- Bit-vectors
- …

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

Assignment

$p_1,\ p_2,\ \neg p_3,\ p_4$

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \lor y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \lor p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

Assignment
$p_1,\ p_2,\ \neg p_3,\ p_4$

$x \geq 0,\ y = x + 1,$
$\neg(y > 2),\ y < 1$

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

Assignment
$p_1,\ p_2,\ \neg p_3,\ p_4$

$x \geq 0,\ y = x + 1,$
$\neg(y > 2),\ y < 1$

Theory Solver

Unsatisfiable
$x \geq 0,\ y = x + 1,\ y < 1$

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \lor y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \lor p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

Assignment
$p_1,\ p_2,\ \neg p_3,\ p_4$

$x \geq 0,\ y = x + 1,$
$\neg(y > 2),\ y < 1$

Theory Solver

Unsatisfiable
$x \geq 0,\ y = x + 1,\ y < 1$

New Lemma
$\neg p_1 \lor \neg p_2 \lor \neg p_4$

SAT + Theory solvers

New Lemma

$\neg p_1 \vee \neg p_2 \vee \neg p_4$

Unsatisfiable

$x \geq 0,\ y = x + 1,\ y < 1$

Theory Solver

AKA
Theory conflict