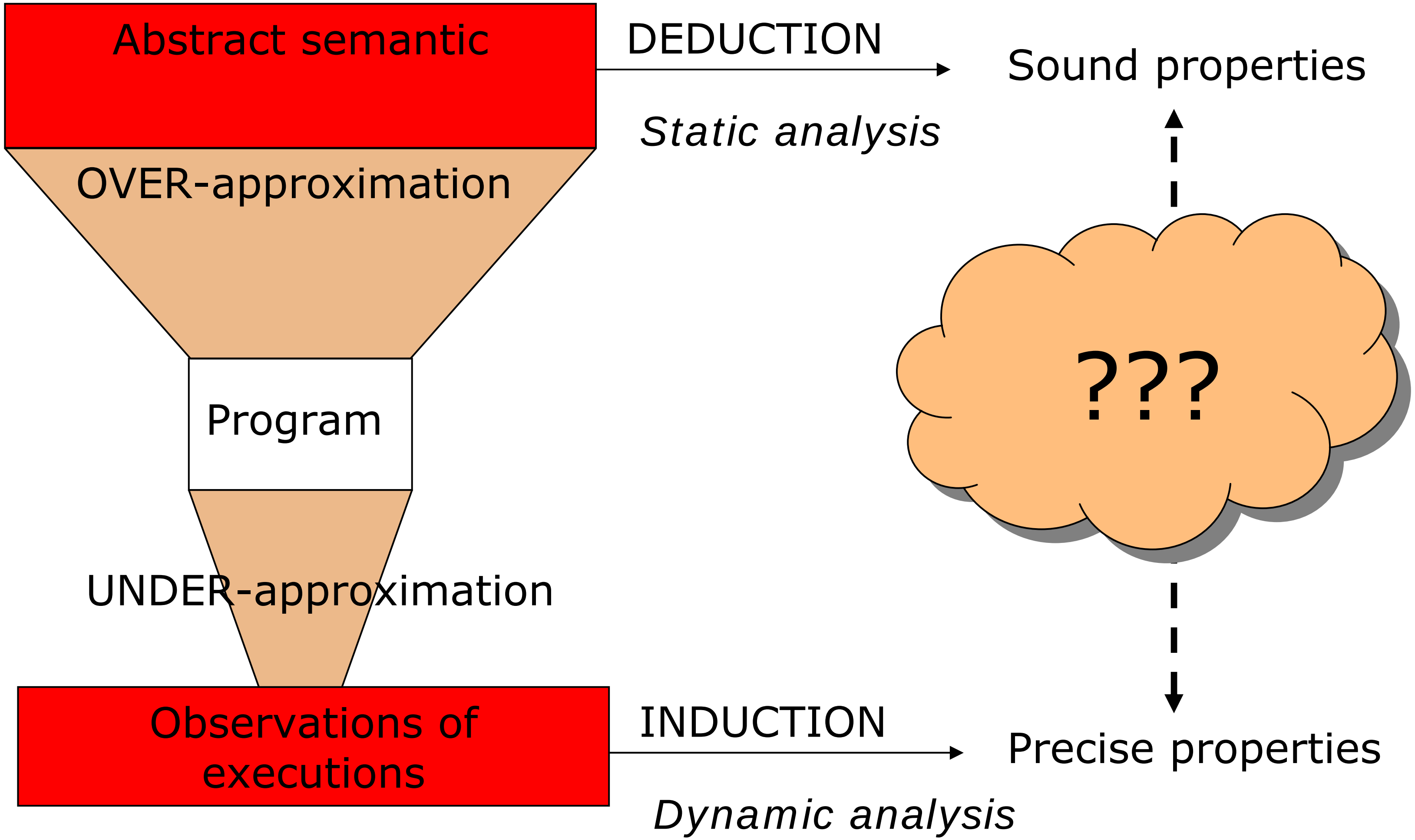
Current Work

Proving or Disproving Properties with Constraint Reasoning

T. Denmat M. Ducassé A. Gotlieb
Irisa Rennes France

WLPE 05 - Sitges

Inference of Program Properties



Abstract semantic

OVER-approximation

Program

UNDER-approximation

Observations of executions

DEDUCTION

Static analysis

Sound properties

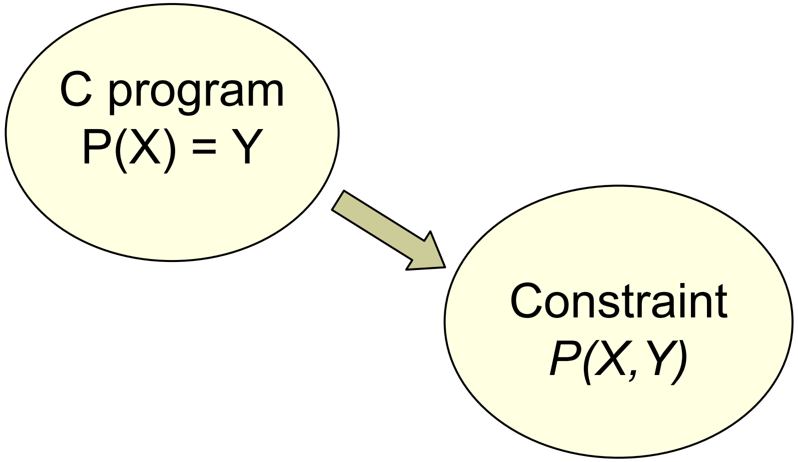
???

INDUCTION

Dynamic analysis

Precise properties

Our approach : 1-Modeling

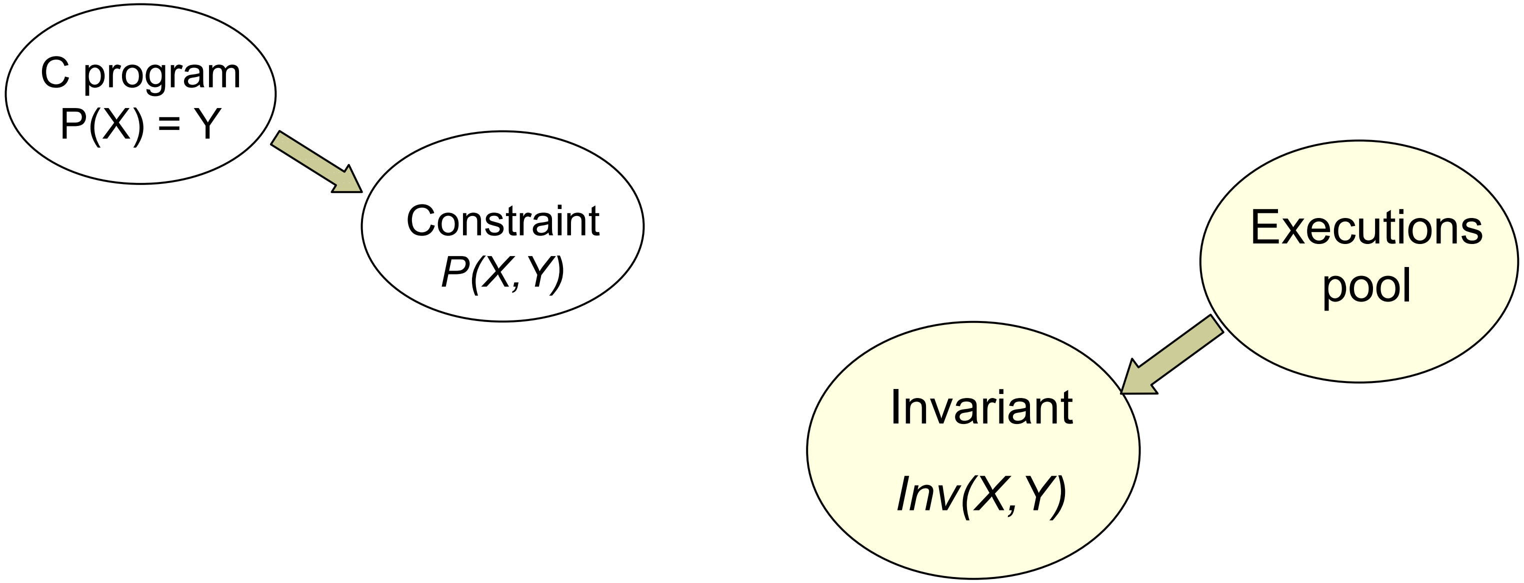


C program
P(X) = Y

Constraint
P(X,Y)

- Modeling of the relational semantics
  - S[P] = {(X,Y) | there exists a trace t with init(t) = X and final(t) = Y}
- Correct and complete
  - P(X,Y) = true $\Leftrightarrow$ (X,Y) $\in$S[P]
- Implemented in Inka

Our approach : 2-Inducing



C program
P(X) = Y

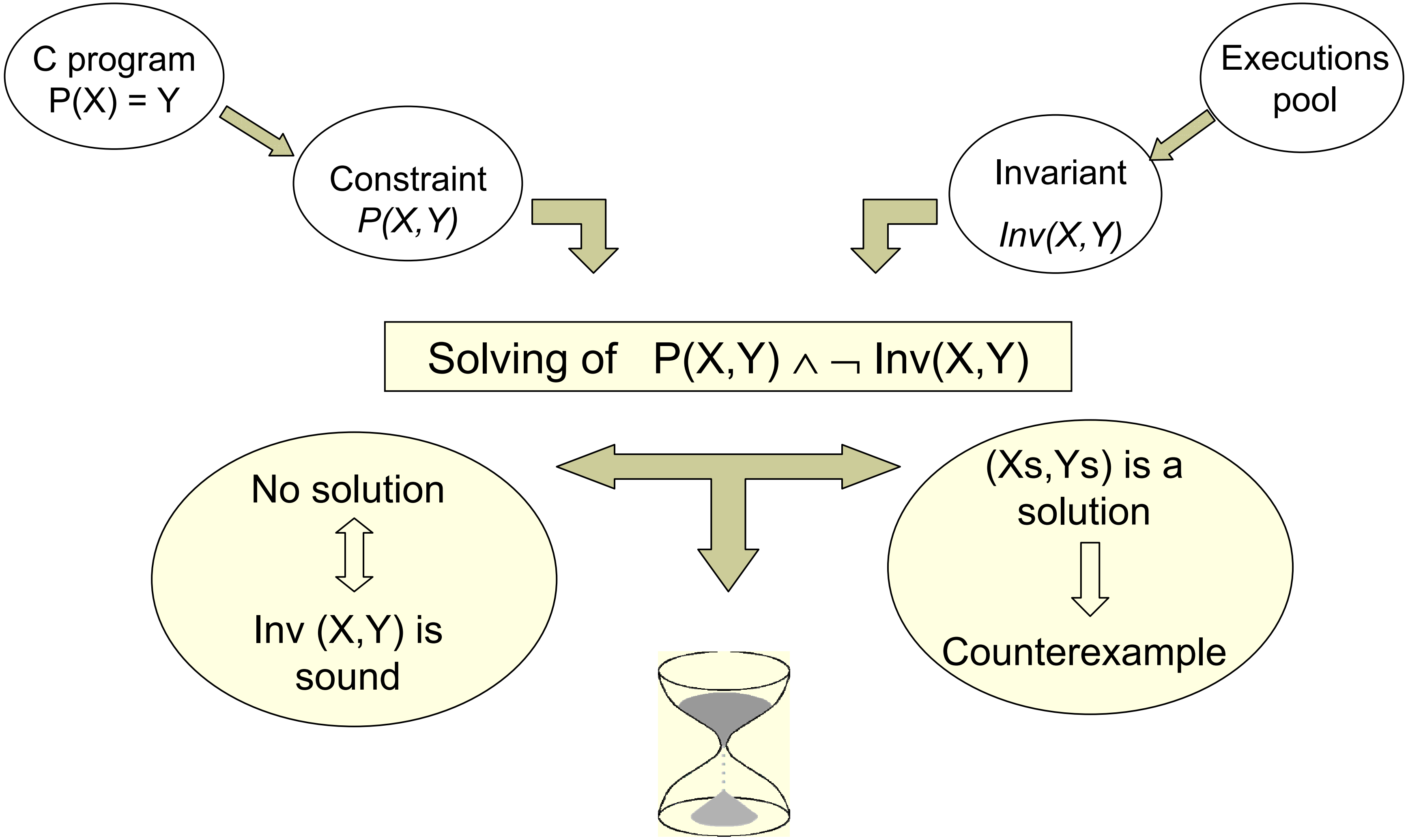
Constraint
P(X,Y)

Executions pool

Invariant
Inv(X,Y)

- Inference of an invariant (= property)
    - Relation between the memory states X and Y
- Could be a relation between intermediary states

Our approach : 3-Refuting



C program
$P(X) = Y$

Constraint
$P(X,Y)$

Executions pool

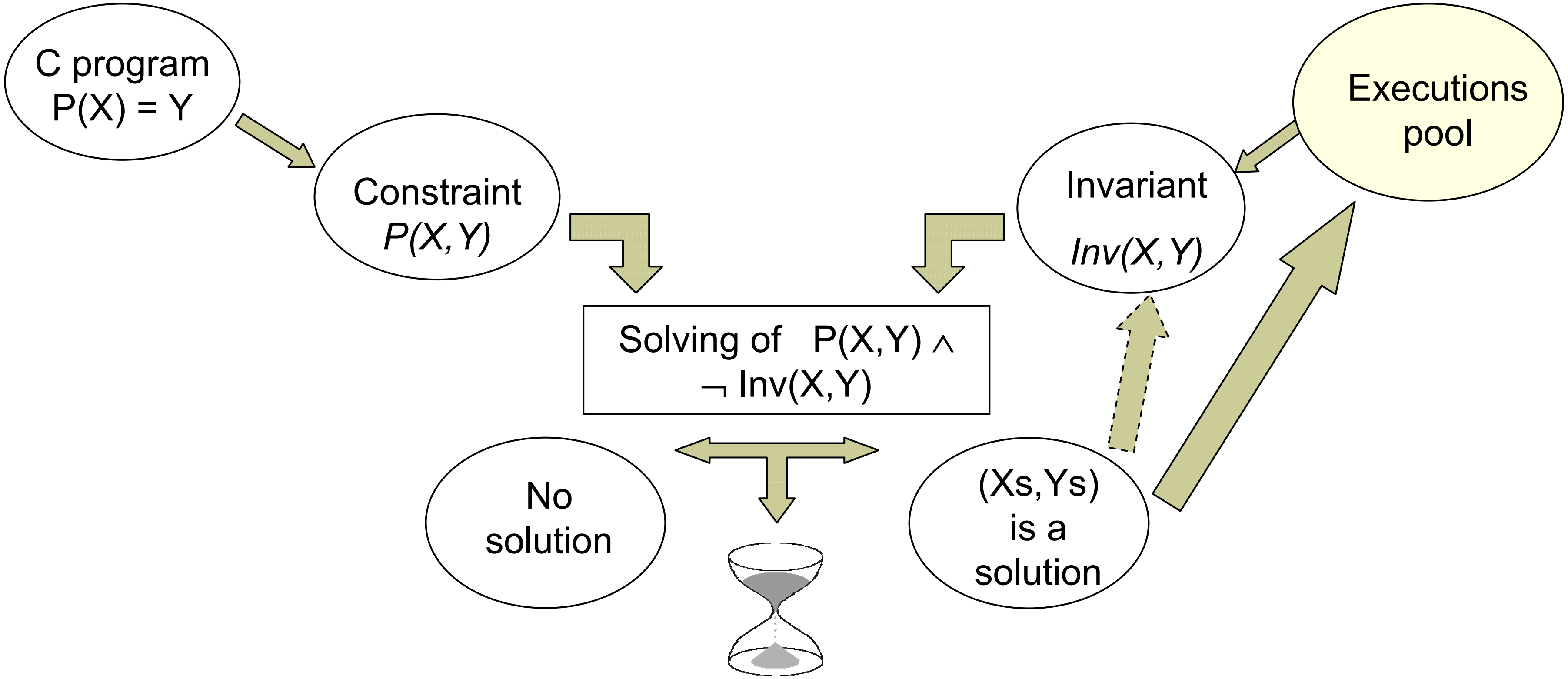
Invariant
$Inv(X,Y)$

Solving of $P(X,Y) \wedge \neg Inv(X,Y)$

No solution
⇕
$Inv(X,Y)$ is sound

(Xs,Ys) is a solution
⇩
Counterexample

Our approach : 4-Refining



C program
P(X) = Y

Constraint
P(X,Y)

Solving of $P(X,Y) \wedge \neg Inv(X,Y)$

No
solution

(Xs,Ys)
is a
solution

Invariant
Inv(X,Y)

Executions
pool

- Enlarge the pool of executions with the new one
- Maybe refine directly the invariant

Expected contributions

- Obtain the correctness of dynamically inferred invariants
- Precise invariants due to the mechanism of refinement
- Potentially very large panel of invariants (all the relations !)

Outline

- *Step 1* : Translation of an imperative program into CLP(FD)
- *Step 2* : Dynamic inference of properties

  (Daïkon as a black-box)
- *Step 3 :* Validation of properties
  - Motivating example
  - Problems and future work

- *No step 4 until now !!!*

Constraint-model of a program

- Translation of an imperative program into a constraint system
- 2 main problems
  - multiple assignments to a variable
  - conditionals and loops

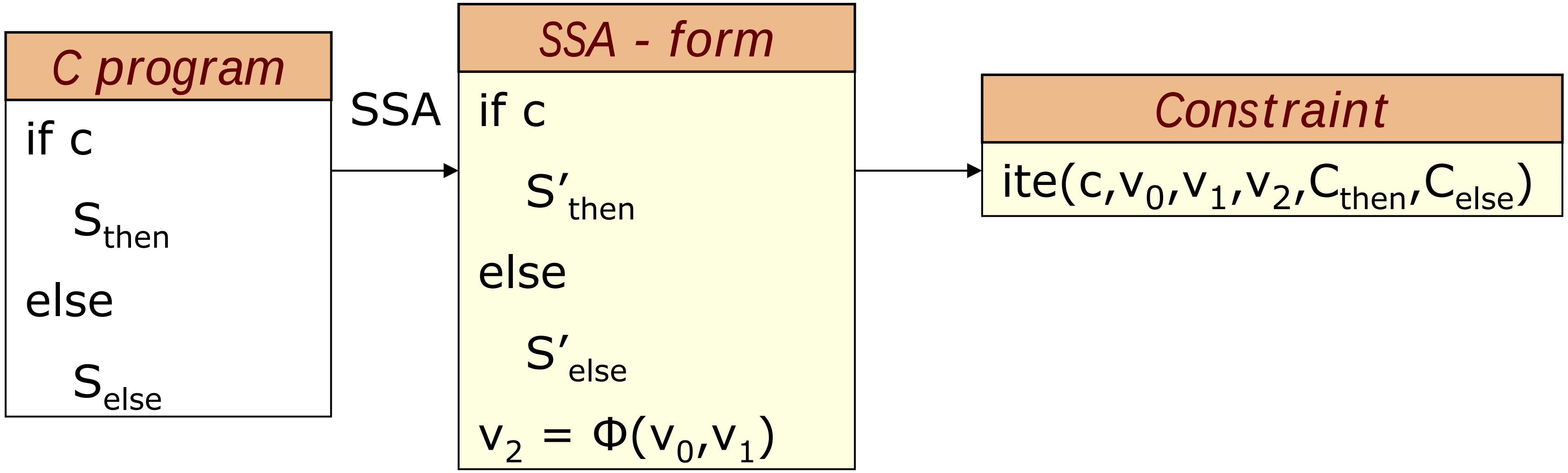
- Approach of Gotlieb et al. [ISSTA 98]
  - SSA-Form
  - New constraint combinators

SSA Form

- Translation of the program into SSA-form
  - Preserves the semantics
  - Each variable is assigned only once during execution
    - Except the iteration structures
  - Data flow is preserved via phi-functions

- Direct translation into constraints
  - A variable in the SSA form -> A logic variable
  - A control-structure -> A constraint

“Ite” combinator



C program

if c

S_{then}

else

S_{else}

SSA

SSA - form

if c

S'_{then}

else

S'_{else}

$v_2 = \Phi(v_0,v_1)$

Constraint

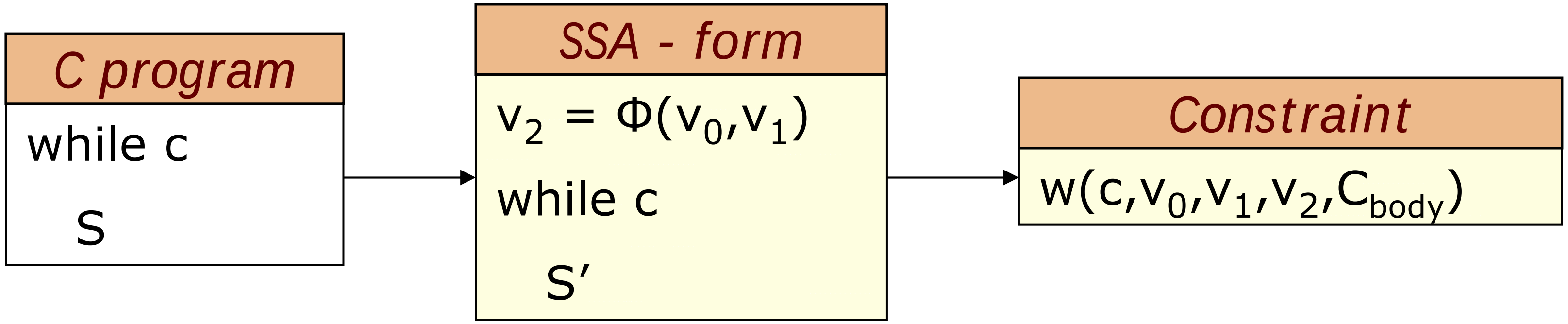
$\mathrm{ite}(c,v_0,v_1,v_2,C_{then},C_{else})$

$\mathrm{ite}(c,v_0,v_1,v_2,C_{then},C_{else}):$

Guarded - constraints

$$\begin{cases} \neg\ (c \wedge C_{then} \wedge v_2 = v_0) \rightarrow \neg c \wedge C_{else} \wedge v_2 = v_1 \\ \neg\ (\neg c \wedge C_{then} \wedge v_2 = v_1) \rightarrow c \wedge C_{then} \wedge v_2 = v_0 \end{cases}$$

“w” Combinator



C program

while c

  S

SSA - form

$v_2 = \Phi(v_0,v_1)$

while c

  S′

Constraint

$w(c,v_0,v_1,v_2,C_{body})$

$W(c,v_0,v_1,v_2,C_{body})$:

$\neg \quad (c \wedge C_{body}) \rightarrow \neg c \wedge v_2 = v_0$

$\neg \quad (\neg c \wedge v_0 = v_2) \rightarrow c \wedge C_{body} \wedge w(c,v_1,v_3,v_2,C'_{body})$

Dynamic inference of properties

- We use Daïkon as a black box, in its by default configuration [Ernst ICSE 99]

- Generate a set of potential relationships between variables of a program
  - At “interesting” points of the program
  - For “interesting” variables
- Run a test suite
- Consider relationships that hold over every test case as a *Likely Invariant*

Motivating example

```
int foo (int n, int r)
   b = 0;
   while (n > 0)
     if (b == 0)
      b = 1;
      r ++;
     else
      b = 0;
      r --;
   return r;
```

Likely invariants inferred

- orig(r) = 0 ⇒ return = 0
- return = 0 ⇒ orig(r) = 0
- return ≥ orig(r)

Validation of likely invariants

- Problem of the Oracle :
  - Difficult to know if likely invariants hold
- Automatically checking these invariants is crucial
- Related work
  - Nimmer and Ernst 02 : based on a theorem prover
    - Proving properties
  - Vaziri and Jackson 00 : based on constraint solving
    - Disproving properties
- Our method :
  - Both proving and disproving invariants

Declarative semantics of invariant validation

- Gopal Gupta [the LP paradigm 99]
  - Pre(X) : pre-condition on input vector X
  - P(X,Y) : denotation of an imperative program
    - Relation between input vector X an output vector Y
  - Post(X,Y) : post-condition

  - Post condition is proved to hold if the following goal has no solution
    - Pre(X), P(X,Y), not Post(X,Y)

State space reduction with CLP

- Using pure horn logic :
  - Generate and Test
  - Try all values of X such that Pre(X)
- Using a CLP denotation :
  - Constrain – generate and Test
  - Asserting not Post(X,Y) reduces the search space
- Conjecture :
  - The reduction makes the approach more tractable

Running example – invariant 1

- Refutation of $orig(r) = 0 \Rightarrow return = 0$
- `foo(N,R,Ret)` $\wedge$ `R = 0` $\wedge$ `Ret \= 0`

  Input domains reduction :

  N ∈ [1,sup], R = 0

  labeling step :

  find a solution : N = 1, R = 0, Ret = 1
- Invariant 1 is disproved

Running example – invariant 2

- Refutation of *return* = *0* $\Rightarrow$ *orig(r)* = *0*

- `foo(N,R,Ret)` $\wedge$ `Ret = 0` $\wedge$ `R \= 0`

  Input domains reduction :

  N $\in$ [1,sup], R $\in$ [inf,-1] $\cup$ [1,sup]

  labeling step :

  find a solution : N = 1, R = -1, Ret = 0

- Invariant 2 is disproved

Comments

- The labeling step is crucial to find counter examples
- In our two examples the default labeling procedure is “magically” efficient enough
  - For example, beginning to label variable R would have been terrible

- Future work
  - Design specialized heuristics for finding counter examples

Running example – invariant 3

- Refutation of *return* ≥ *orig(r)*

- **`foo(N,R,Ret)`** $\wedge$ **`Ret < R`**

  Input domains reduction :

  N ∈ Ø, R ∈ Ø

  No labeling step

- Invariant 3 is proved

Details of the refutation 3

Initial state

Constraint store

B = 0,

w(…)

RET < R

Variables domains

B in [0,0]

N in [-100,100]

R in [-99,100]

RET in [-100,99]

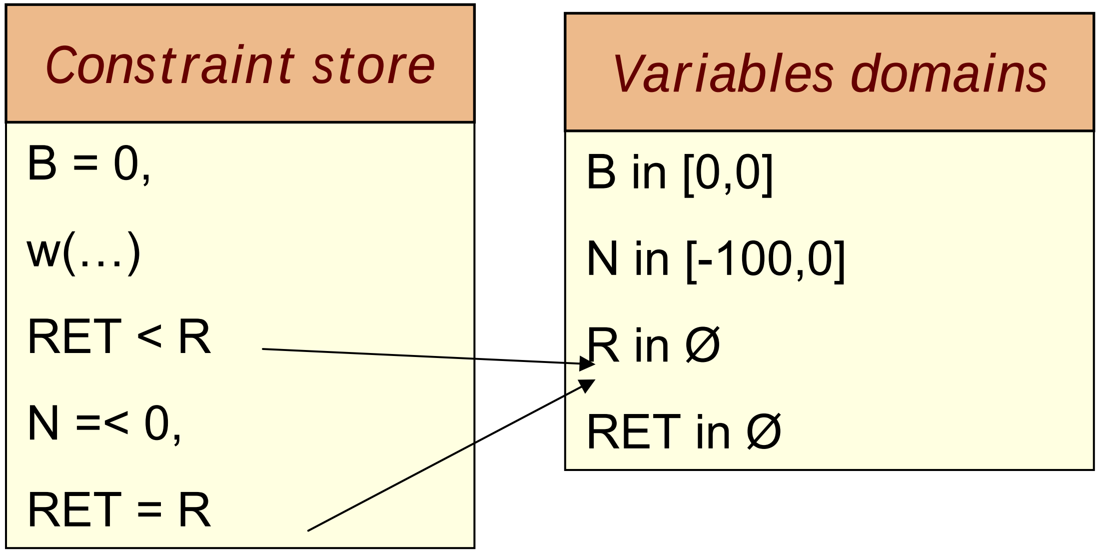
```
int foo (int n, int r)
   b = 0;
   while (n > 0)
     if (b == 0)
      b = 1;
      r ++;
     else
      b = 0;
      r --;
   return r;
```

Details of the refutation 3

Propagation in the w combinator :

entailment checking of the 2nd guard

$$\neg\ (\neg c \wedge v0 = v2) \rightarrow c \wedge Cbody \wedge w(c,v1,v3,v2,C'body)$$



Constraint store

B = 0,

w(…)

RET < R

N =< 0,

RET = R

Variables domains

B in [0,0]

N in [-100,0]

R in Ø

RET in Ø

```
int foo (int n, int r)
   b = 0;
   while (n > 0)
    if (b == 0)
     b = 1;
     r ++;
    else
     b = 0;
     r --;
   return r;
```

Failure ➔ the guard is entailed

Details of the refutation 3

Propagation of the w combinator :

setting the tail of the constraint

$$\neg\ (\neg c \wedge v0 = v2) \rightarrow c \wedge Cbody \wedge w(c,v1,v3,v2,C'body)$$

Constraint store

B = 0,

w(…)

RET < R

N > 0

N1 = N -1

B1 = 1

R1 = R + 1

Variables domains

B in [0,0]

N in [1,100]

R in [-99,99]

RET in [-100,98]

N1 in [0,99]

B1 in [1,1]

R1 in [-98,100]

```
int foo (int n, int r)
  b = 0;
  while (n > 0)
   if (b == 0)
    b = 1;
    r ++;
   else
    b = 0;
    r --;
  return r;
```

Details of the refutation 3

Propagation in the w combinator :

entailment checking of the 2nd guard again

Constraint store

B = 0,
w(…)
RET < R
N > 0
N1 = N -1
B1 = 1
R1 = R + 1
N1 =< 0
RET = R1

Variables domains

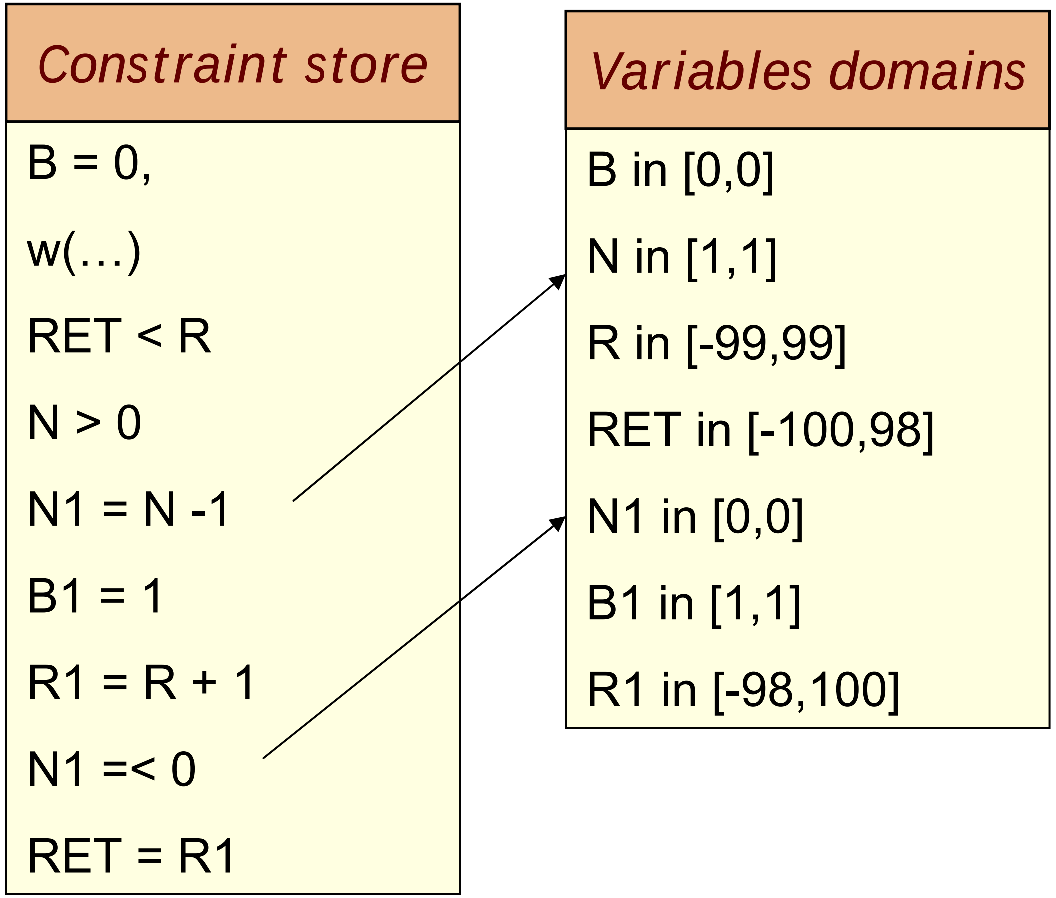
B in [0,0]
N in [1,100]
R in [-99,99]
RET in [-100,98]
N1 in [0,99]
B1 in [1,1]
R1 in [-98,100]

```
int foo (int n, int r)
  b = 0;
  while (n > 0)
    if (b == 0)
     b = 1;
     r ++;
    else
     b = 0;
     r --;
  return r;
```

Details of the refutation 3

Propagation in the w combinator :

entailment checking of the 2nd guard again



Constraint store

B = 0,
w(…)
RET < R
N > 0
N1 = N -1
B1 = 1
R1 = R + 1
N1 =< 0
RET = R1

Variables domains

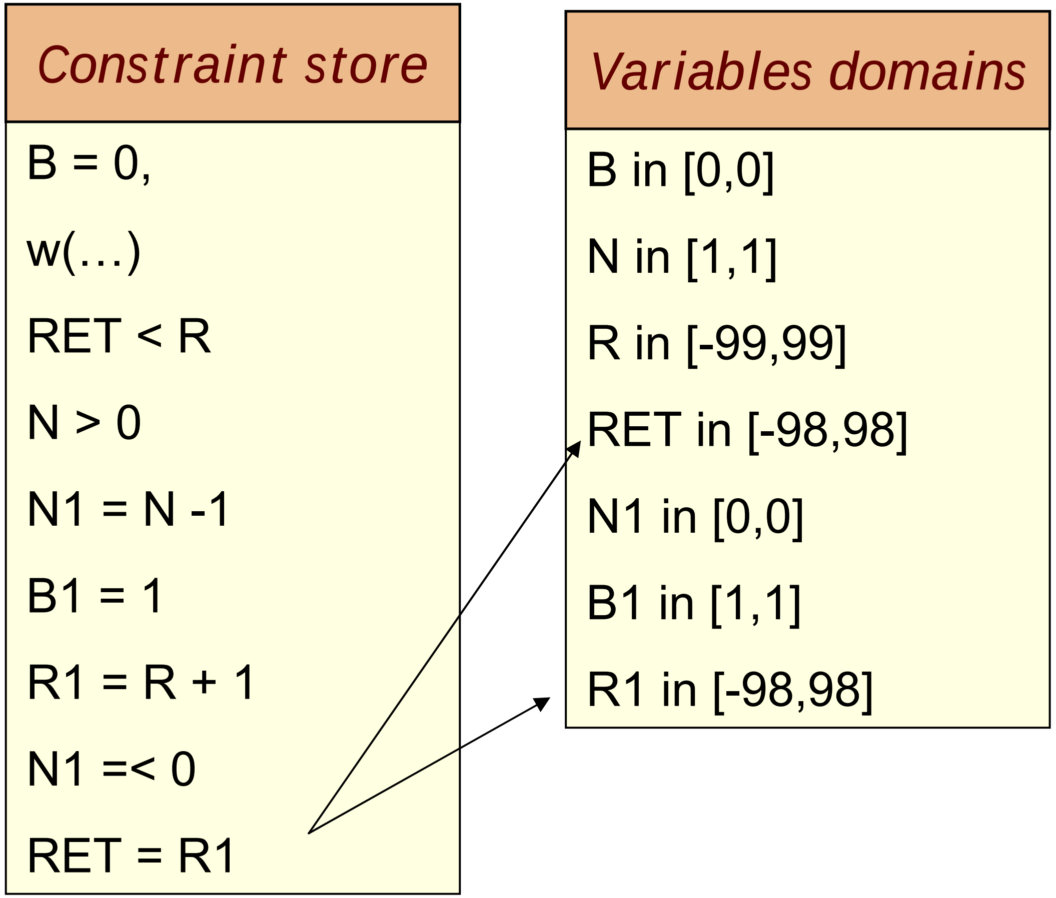
B in [0,0]
N in [1,1]
R in [-99,99]
RET in [-100,98]
N1 in [0,0]
B1 in [1,1]
R1 in [-98,100]

```
int foo (int n, int r)
   b = 0;
   while (n > 0)
     if (b == 0)
      b = 1;
      r ++;
     else
      b = 0;
      r --;
   return r;
```

Details of the refutation 3

Propagation in the w combinator :

entailment checking of the 2nd guard again



Constraint store

B = 0,

w(…)

RET < R

N > 0

N1 = N -1

B1 = 1

R1 = R + 1

N1 =< 0

RET = R1

Variables domains

B in [0,0]

N in [1,1]

R in [-99,99]

RET in [-98,98]

N1 in [0,0]

B1 in [1,1]

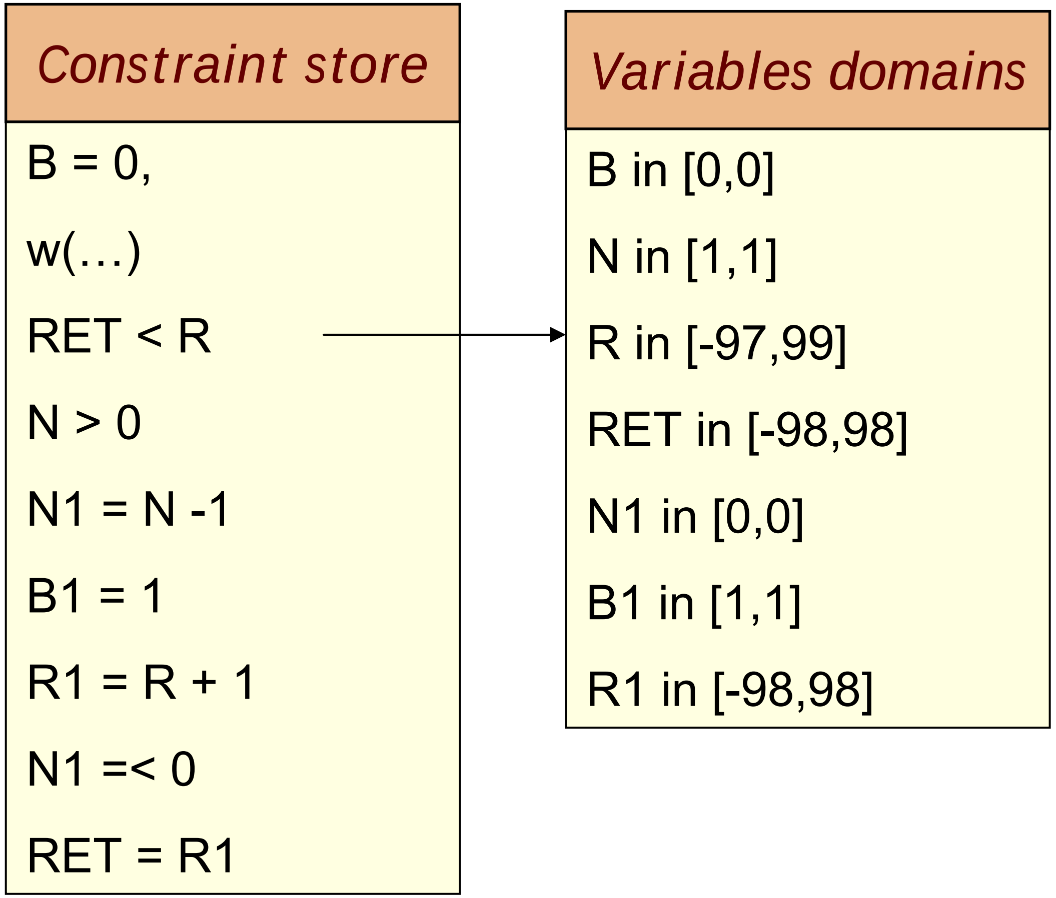
R1 in [-98,98]

```
int foo (int n, int r)
   b = 0;
   while (n > 0)
     if (b == 0)
      b = 1;
      r ++;
     else
      b = 0;
      r --;
   return r;
```

Details of the refutation 3

Propagation in the w combinator :

entailment checking of the 2nd guard again



Constraint store

B = 0,
w(…)
RET < R
N > 0
N1 = N -1
B1 = 1
R1 = R + 1
N1 =< 0
RET = R1

Variables domains

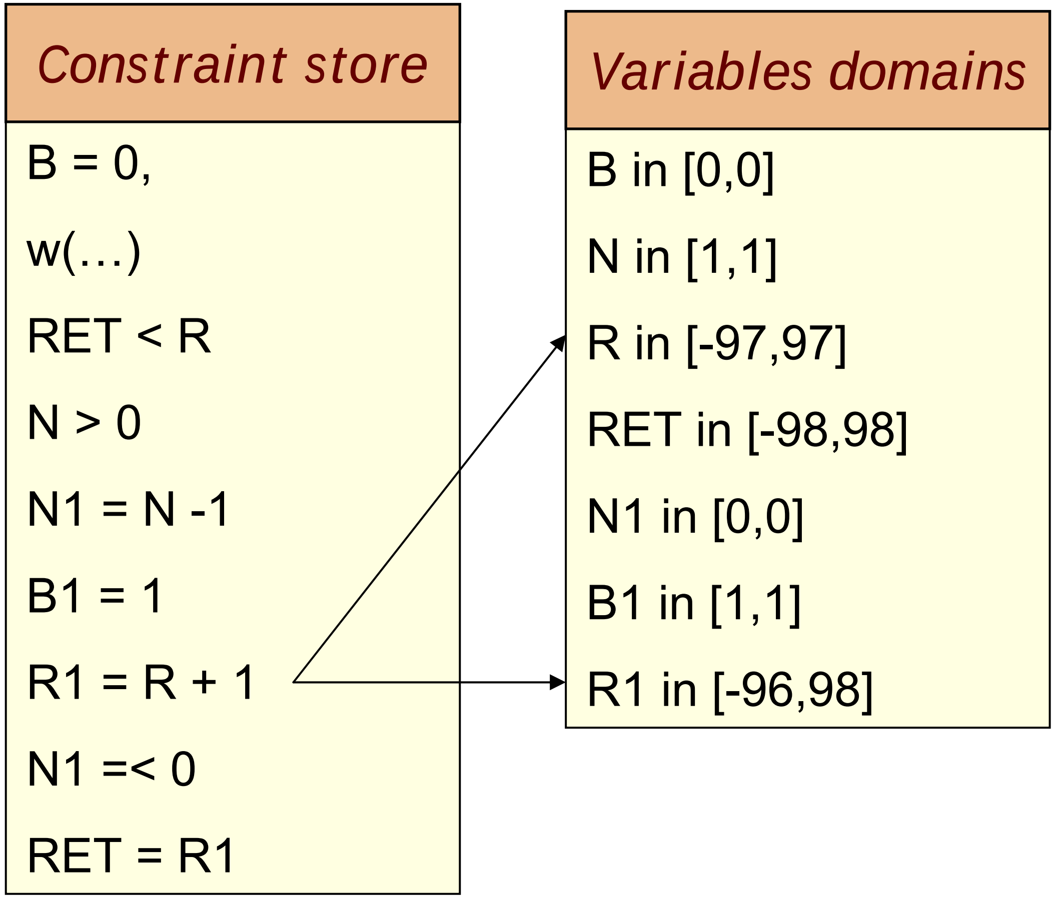
B in [0,0]
N in [1,1]
R in [-97,99]
RET in [-98,98]
N1 in [0,0]
B1 in [1,1]
R1 in [-98,98]

```
int foo (int n, int r)
   b = 0;
   while (n > 0)
     if (b == 0)
      b = 1;
      r ++;
     else
      b = 0;
      r --;
   return r;
```

Details of the refutation 3

Propagation in the w combinator :

entailment checking of the 2nd guard again



Constraint store

B = 0,

w(…)

RET < R

N > 0

N1 = N -1

B1 = 1

R1 = R + 1

N1 =< 0

RET = R1

Variables domains

B in [0,0]

N in [1,1]

R in [-97,97]

RET in [-98,98]

N1 in [0,0]

B1 in [1,1]

R1 in [-96,98]

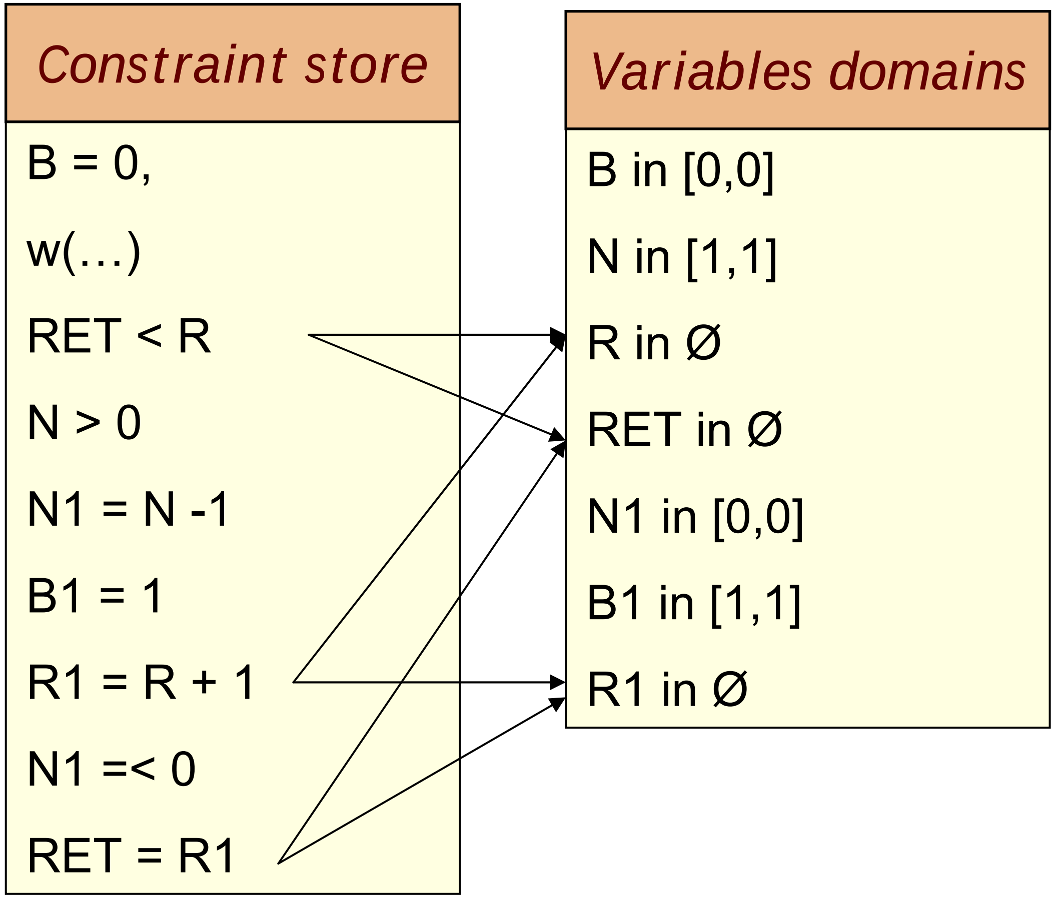
```
int foo (int n, int r)
   b = 0;
   while (n > 0)
     if (b == 0)
      b = 1;
      r ++;
     else
      b = 0;
      r --;
   return r;
```

...

Details of the refutation 3

Propagation in the w combinator :

entailment checking of the 2nd guard again



Constraint store

B = 0,

w(…)

RET < R

N > 0

N1 = N -1

B1 = 1

R1 = R + 1

N1 =< 0

RET = R1

Variables domains

B in [0,0]

N in [1,1]

R in Ø

RET in Ø

N1 in [0,0]

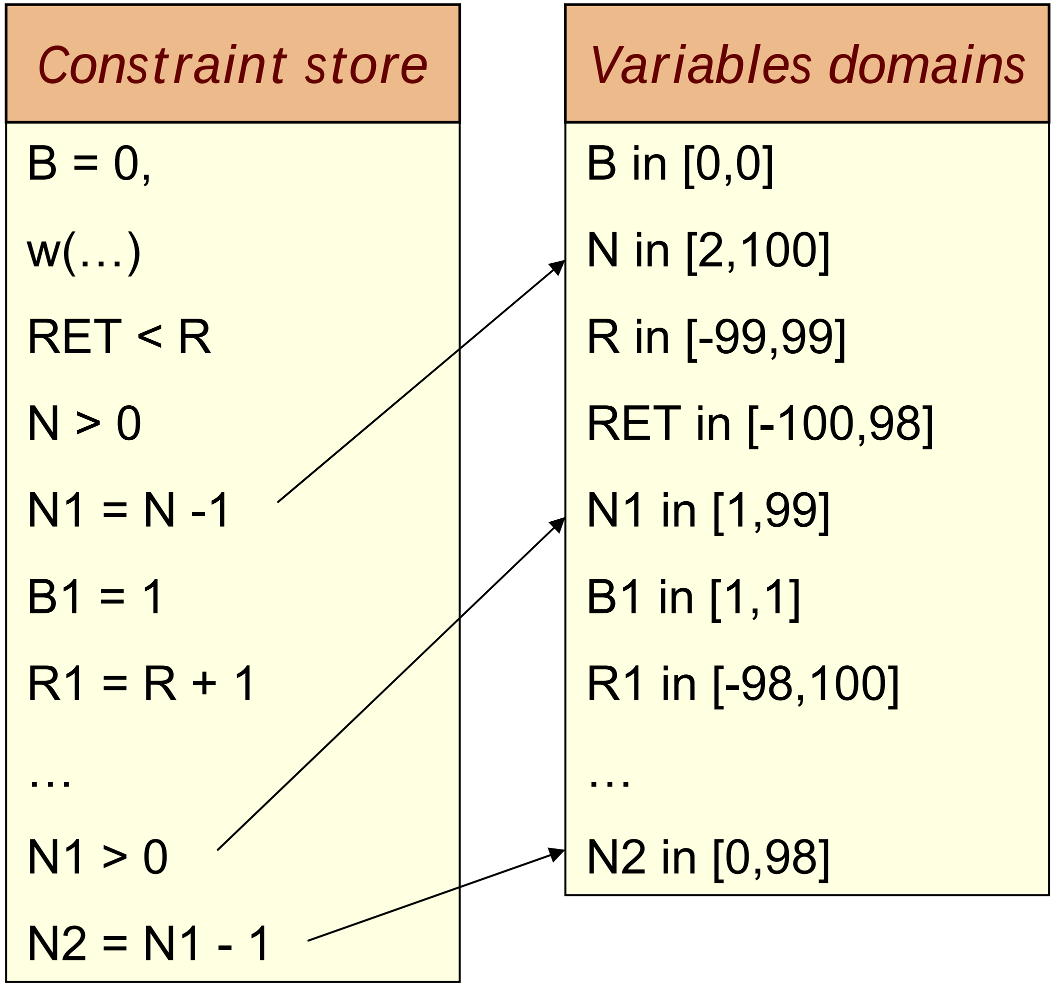
B1 in [1,1]

R1 in Ø

```
int foo (int n, int r)
   b = 0;
   while (n > 0)
     if (b == 0)
      b = 1;
      r ++;
     else
      b = 0;
      r --;
   return r;
```

Details of the refutation 3

Propagation of the w combinator : setting the tail of the constraint



Constraint store

B = 0,

w(…)

RET < R

N > 0

N1 = N -1

B1 = 1

R1 = R + 1

…

N1 > 0

N2 = N1 - 1

Variables domains

B in [0,0]

N in [2,100]

R in [-99,99]

RET in [-100,98]

N1 in [1,99]

B1 in [1,1]

R1 in [-98,100]

…

N2 in [0,98]

```
int foo (int n, int r)
   b = 0;
   while (n > 0)
     if (b == 0)
      b = 1;
      r ++;
     else
      b = 0;
      r --;
   return r;
```

...

Details of the refutation 3

Constraint store

B = 0,
w(…)
RET < R
N > 0
N1 = N -1
B1 = 1
R1 = R + 1
…
N1 > 0
N2 = N1 – 1
N100 = N99 - 1

Variables domains

B in [0,0]
N in [100,100]
R in [-99,99]
RET in [-100,98]
N1 in [99,99]
B1 in [1,1]
R1 in [-98,100]
…
N2 in [98,98]
N100 in [0,0]

We have a failure as it is impossible to unfold the loop and to exit the loop

Comments

- The propagation is very long
  - We need to show inconsistencies at each loop unfolding
  - Each inconsistency is long to demonstrate
    - Bound consistency → slow convergence

- Future work
  - Use information about the loops such as loop invariants to add redundant constraint
  - Mix CLP(FD) with other types of constraint solver

Conclusion

- An approach to both prove and disprove invariants based on constraints
  - No approximation
  - Based on clp(fd)
- Need to specialize constraint techniques to this particular problem
  - Propagation step
  - Labeling step