



Software Verification

Lecture 5: Assertion Inference

Carlo A. Furia
Bertrand Meyer

Proving Programs Automatically



The Program Verification problem:

- **Given:** a program P and a specification $S = [\text{Pre}, \text{Post}]$
- **Determine:** if **every execution** of P , for any value of input parameters, **satisfies** S
- **Equivalently:** establish whether $\{\text{Pre}\} P \{\text{Post}\}$ is **(totally) correct**

- A **general and fully automated solution** to the **Program Verification problem** is **unachievable** because the problem is **undecidable**
- One of the consequences of this inescapable limitation is the **impossibility** of **computing** verification conditions (**VC**) **fully automatically**
 - **VC:** intermediate **assertions** used in the correctness proof
 - (It is not an obvious consequence: formally, a reduction between undecidable problems)

Proving Programs Automatically

The Program Verification problem:

- **Given:** a program P and a specification $S = [\text{Pre}, \text{Post}]$
- **Determine:** if **every execution** of P , for any value of input parameters, **satisfies** S
- **Equivalently:** establish whether $\{\text{Pre}\} P \{\text{Post}\}$ is **(totally) correct**

Practically:

Proving the correctness of a computer program requires knowledge about the program that is not readily available in the program text

-- Chang & Leino

In this lecture, we survey **techniques** to **automatically infer assertions** in interesting special cases

The Assertion Inference Paradox



Correctness is consistency of implementation to specification

The paradox:

if the specification is inferred from the implementation,
what do we prove?

The Assertion Inference Paradox



The paradox:

if the specification is inferred from the implementation,
what do we prove?

Possible retorts:

- The paradox only arises for correctness proofs; there are other applications
(e.g. reverse-engineering legacy software)
- The result may be presented to a programmer for assessment
- Inferred specification may be inconsistent, thus denoting a problem

The Assertion Inference Paradox



The paradox:

if the specification is inferred from the implementation,
what do we prove?

The paradox does **not arise** if we **only infer VC** (i.e., intermediate assertions) and not specifications (pre and postconditions)

- **VC** are a **technical means to an end** (proving correctness)
 - the tool infers loop invariants
- The **specification** is a formal description of **what** the implementation should **do**
 - the programmer writes the specification

Invariants



Let us consider a general (and somewhat informal) definition of **invariant**:

Def. Invariant: **assertion** whose **truth** is **preserved** by the execution of (parts of) a program.

x: INTEGER

from x := 1 until ... loop x := - x end

Some invariants:

- $-1 \leq x \leq 1$
- $x = -1 \vee x = 0 \vee x = 1$
- $x \geq -10$

Kinds of Invariants

Def. Invariant: assertion whose truth is preserved by the execution of (parts of) a program.

We can identify different families of invariants, according to what parts of the program preserve the invariant:

- **Location invariant at x:** assertion that holds whenever the computation reaches location x
- **Program invariant:** predicate that holds in any reachable state of the computation
- **Class invariant:** predicate that holds between (external) feature invocations
- **Loop invariant:** predicate that holds after every iteration of a loop body

$$\frac{\begin{array}{l} \{P\} A \{I\} \\ \{I \wedge \neg c\} B \{I\} \end{array}}{\begin{array}{l} \{P\} \\ \text{from } A \text{ until } c \\ \text{loop } B \text{ end} \\ \{I \wedge c\} \end{array}}$$

Kinds of Invariants

```
1:   x: INTEGER
2:
3:   from x := 1
4:   until ...
5:   loop x := - x end
```

- Location invariant at 2:
 $x = 0$
- Loop invariant:
 $x = -1 \vee x = 1$
- Program invariant:
 $x \geq -10$

Focus on Loop Invariants



If we have **loop invariants** we can get (almost) **everything else** at little cost

- but **not** vice versa:
getting loop invariants requires **invention**

In the following discussion we **focus on loop invariants** (and call them simply "**invariants**")

This focus is also consistent with the Assertion Inference Paradox

Focus on Loop Invariants

The various kinds of invariants are **closely related** by the inference rules of **Hoare logic**

- If Lx is a **location** invariant at x then:

$$@x \Rightarrow Lx$$

is a **program** invariant

- If P is a **program** invariant then it is also a **location** invariant at every location x

- If I is a **loop** invariant of:

x : **from** ... **until** c **loop** ... **end**

then $I \wedge c$ is a **location** invariant at $x+1$

- If L is a **location** invariant at $x+1$:

x : $a := b + 3$

then $L [b + 3 / a]$ is a **location** invariant at x

- Etc...

$$\frac{\begin{array}{l} \{P\} A \{I\} \\ \{I \wedge \neg c\} B \{I\} \end{array}}{\begin{array}{l} \{P\} \\ \text{from } A \text{ until } c \\ \text{loop } B \text{ end} \\ \{I \wedge c\} \end{array}}$$

$$\{P [e / x]\} x := e \{P\}$$

Techniques for Invariant Inference

Classification of invariant inference techniques:

- **Dynamic** techniques
- **Static** techniques
 - **statistical** techniques
 - **exact** techniques

(Roughly) direction of increasing:

soundness



completeness



sophistication
mathematical



Classification is neither sharp nor complete, yet **useful**



Exact Static Techniques for Invariant Inference

Static Invariant Inference: classification



Static exact techniques for invariant inference are further classified in categories:

- Direct
- Assertion-based
 - postcondition weakening
- Based on abstract interpretation
 - forward analysis (bottom-up)
 - backward analysis (top-down)
- Constraint-based
 - usually, template-based



Exact Static Techniques for Invariant Inference:

Postcondition-weakening Approach

The Role of User-provided Contracts



Techniques for invariant inference rarely take advantage of **other annotations** in the program text, such as **contracts** provided by the user

- Not every annotation can (or should, cf. **Assertion Inference Paradox**) be inferred automatically!

However, there is a close **connection** between a **loop's invariant** and its **postcondition**

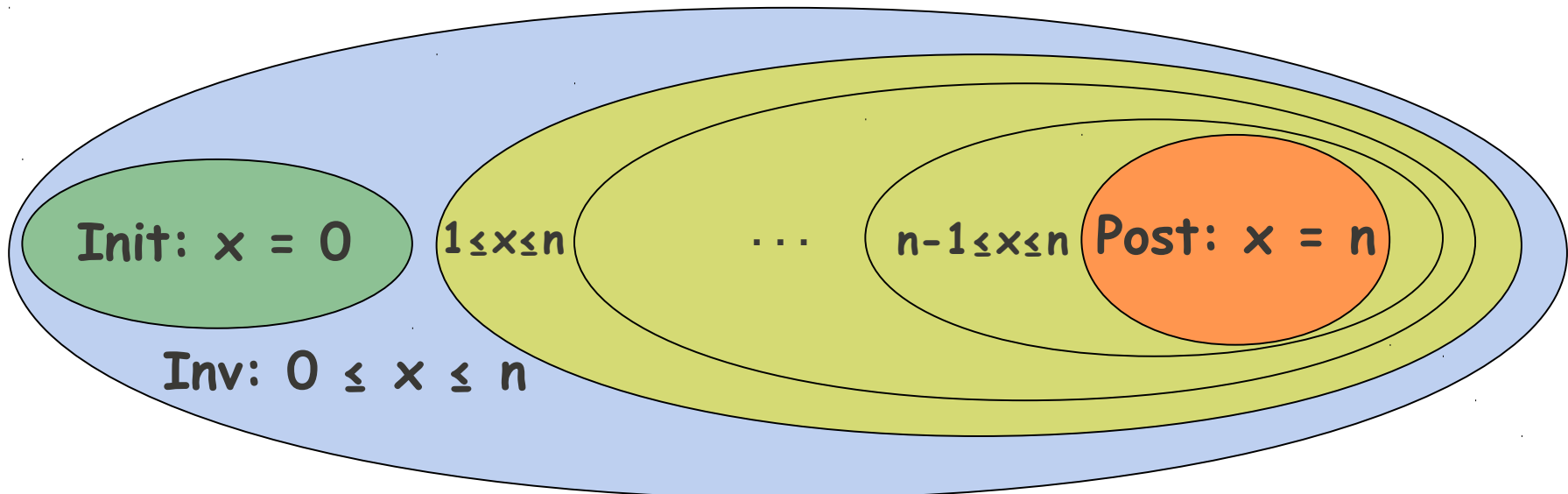
The Role of User-provided Contracts

- However, there is a close **connection** between a **loop's invariant** and its **postcondition**

The **invariant** is a **weakened** form of the **postcondition**

- It is a larger collection of program states

- Example: `from x := 0 until x = n loop x := x + 1 end`
 - Post: $x = n$ (for some $n > 0$)
 - Invariant: $0 \leq x \leq n$



Invariants by Postcondition Weakening

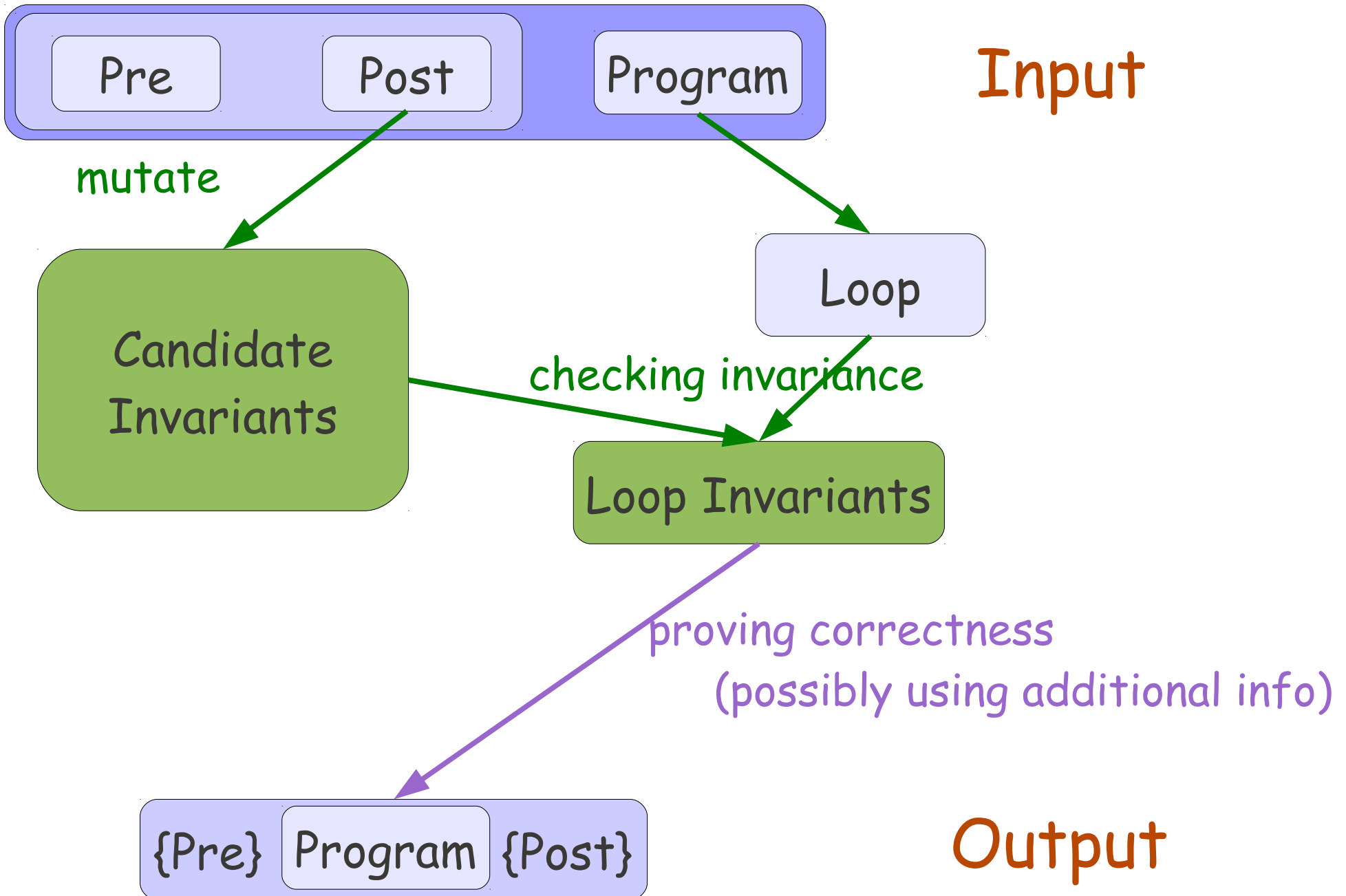
- In a nutshell:

Static verification of candidate invariants
obtained by mutating postconditions

1. Assume the availability of **postconditions**
2. **Mutate postconditions** according to various **heuristics**
 - the **heuristics** mirror common patterns that link postconditions to invariants
 - each mutated postcondition is a candidate invariant
3. **Verify** which candidates are indeed invariants
 - With an **automatic** program prover such as Boogie
4. Retain all verified **invariants**

- 2009 -- C.A. Furia & B. Meyer
- Implementation: gin-pink which finds invariants in Boogie programs

Loop invariant inference



Postcondition Mutation Heuristics



Constant relaxation

- replace "constant" by "variable"
 - cannot/may be changed by any of the loop bodies

Uncoupling

- replace subexpression appearing twice by two subexpressions
 - for example: subexpression = variable id

Term dropping

- remove a conjunct

Variable aging

- replace subexpression by another expression representing its previous value

Invariant Inference: the Algorithm

- **Goal**: find invariants of loops in procedure **proc**
 - For each:
 - **post**: postcondition clause of **proc**
 - **loop**: outer loop in **proc**
- compute **all mutations M** of **post** w.r.t. **loop**
- considering postcondition clauses separately implements **term dropping**
- **Result**: any formula in **M** which can be **verified as invariant** of **any loop** in **proc**

Maximum value of an array

```
max (A: ARRAY [T]; n: INTEGER): T
  require A.length = n ≥ 1
  local i: INTEGER
  do
    from i := 0; Result := A[1];
    until i = n
    loop
      i := i + 1
      if Result ≤ A[i] then Result := A[i] end
    end
  ensure ( ∀ 1 ≤ j ≤ n ⇒ A[j] ≤ Result ) and
         ( ∃ 1 ≤ j ≤ n ∧ A[j] = Result )
```

Maximum value of an array

```
max (A: ARRAY [T]; n: INTEGER): T
  require A.length = n ≥ 1
  ensure  ( ∀ 1 ≤ j ≤ n ⇒ A[j] ≤ Result ) and
          ( ∃ 1 ≤ j ≤ n ∧ A[j] = Result )
```

- **Constant relaxation:** replace "constant" n by "variable" i
- **Term dropping:** remove second conjunct

Invariant: $\forall 1 \leq j \leq i \Rightarrow A[j] \leq \text{Result}$

Maximum value of an array (cont'd)

```
max (A: ARRAY [T]; n: INTEGER): T
  require A.length = n ≥ 1
  ensure  ( ∀ 1 ≤ j ≤ n ⇒ A[j] ≤ Result ) and
          ( ∃ 1 ≤ j ≤ n ∧ A[j] = Result )
```

- Term dropping: remove first conjunct

Invariant: $\exists 1 \leq j \leq n \wedge A[j] = \text{Result}$

Maximum value of an array (2nd version)



```
max_v2 (A: ARRAY [T]; n: INTEGER): T
  require A.length = n ≥ 1
  local i: INTEGER
  do
    from i := 1; Result := A[1];
    until i > n
    loop
      if Result ≤ A[i] then Result := A[i] end
      i := i + 1
    end
  ensure  $\forall 1 \leq j \leq n \Rightarrow A[j] \leq \text{Result}$ 
```

Maximum value of an array (2nd version)



```
max_v2 (A: ARRAY [T]; n: INTEGER): T
  require A.length = n ≥ 1
  ensure ∀ 1 ≤ j ≤ n ⇒ A[j] ≤ Result
```

- **Constant relaxation:** replace "constant" n by "variable" i

$$\forall 1 \leq j \leq i \Rightarrow A[j] \leq \text{Result}$$

- **Variable aging:**

use expression representing the previous value of i : $i - 1$

Invariant: $\forall 1 \leq j \leq i - 1 \Rightarrow A[j] \leq \text{Result}$

Array Partitioning



```
partition (A: ARRAY [T]; n: INTEGER; pivot: T): INTEGER
  require A.length = n ≥ 1
  local l, h: INTEGER
  do
    from l := 1 ; h := n      until l = h
    loop
      from until l = h or A[l] > pivot loop l := l + 1 end
      from until l = h or pivot > A[h] loop h := h - 1 end
      A.swap (l, h)
    end
    if pivot ≤ A[l] then l := l - 1 end ; h := l ; Result := h
  ensure (∀ 1 ≤ k ≤ Result ⇒ A[k] ≤ pivot) and
         (∀ Result < k ≤ n ⇒ A[k] ≥ pivot)
```

Array Partitioning



```
partition (A: ARRAY [T]; n: INTEGER; pivot: T): INTEGER  
require A.length = n ≥ 1  
ensure (∀ 1 ≤ k ≤ Result ⇒ A[k] ≤ pivot) and  
       (∀ Result < k ≤ n ⇒ A[k] ≥ pivot)
```

- **Uncoupling**: replace first occurrence of **Result** by **l**
and second by **h**
(∀ 1 ≤ k ≤ l ⇒ A[k] ≤ pivot) and (∀ h < k ≤ n ⇒ A[k] ≥ pivot)
- **Variable aging**: use expression representing the previous
value of **l**: **l - 1**

Invariant:

(∀ 1 ≤ k ≤ l - 1 ⇒ A[k] ≤ pivot) and (∀ h < k ≤ n ⇒ A[k] ≥ pivot)

Array Partitioning



```
partition (A: ARRAY [T]; n: INTEGER; pivot: T): INTEGER  
require A.length = n ≥ 1  
ensure (∀ 1 ≤ k ≤ Result ⇒ A[k] ≤ pivot) and  
       (∀ Result < k ≤ n ⇒ A[k] ≥ pivot)
```

- **Term dropping:** remove first conjunct
$$\forall \text{Result} < k \leq n \Rightarrow A[k] \geq \text{pivot}$$
- **Constant relaxation:** replace "constant" **Result** by "variable" **h**

Invariant: $\forall h < k \leq n \Rightarrow A[k] \geq \text{pivot}$

Implementation: gin-pink



gin-pink: Generation of INvariants by
PostcondITioN weakening

- written in Eiffel
- **command-line** tool
 - Boogie in / Boogie out
- works with any **high-level language** that can be translated to Boogie
- available for **download** from
<http://se.inf.ethz.ch/people/furia/>

Limitations of the approach

Some invariants are **not** mutations of the postcondition

- “completeness” of the postcondition
- integration with other techniques
- more heuristics

Combinatorial explosion

- user guidance

Dependencies

- especially with nested loops
- user guidance

Limitations of automated reasoning techniques

- they progress quickly



Exact Static Techniques for Invariant Inference:

Constraint-based Approach

Constraint-based Invariant Inference



- In a nutshell:
 - encode semantics of iteration as constraints on a template invariant**
 - 1. Choose a **template invariant expression**
 - template defines a (infinite) set of assertions
 - 2. Encode the **loop semantics** as a set of **constraints** on the template
 - initiation
 - consecution
 - 3. **Solve** the constraints
 - this is usually the complex part
 - 4. Any solution is an **invariant**
- E.g.: 2003 -- Henny Sipma et al.; 2004 -- Zohar Manna et al. ;
2007 -- Tom Henzinger et al.

Constraint-based Inv. Inference: Example



```
dummy_routine (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end
```

- **Template** invariant expression:

$$T = c \cdot x + d \cdot n + e \leq 0$$

- Constraints encoding **loop semantics**:

- **Initiation**: "T holds for the initial values of **x** and **n**"

$$T [0/x; n_0/n] \equiv c \cdot 0 + d \cdot n_0 + e \leq 0 \equiv d \cdot n_0 + e \leq 0$$

Constraint-based Inv. Inference: Example



```
dummy_routine (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end
```

- Constraints encoding **loop semantics**:
 - **Consecution**: “if T holds and one iteration of the loop is executed, T still holds”
$$T[x/x; n/n] \wedge (\neg(x \geq n) \wedge x' = x + 1 \wedge n' = n) \Rightarrow T[x'/x; n'/n]$$
- **Solving** the constraints requires to eliminate occurrences of x, x', n, n'
 - For linear constraints we can use **Farkas's Lemma**

Farkas's Lemma (1902)



Let S be a system of linear inequalities over n real variables:

$$S \triangleq \begin{bmatrix} a_{11}x_1 + \cdots + a_{1n}x_n + b_1 & \leq & 0 \\ \vdots & & \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n + b_m & \leq & 0 \end{bmatrix}$$

and let Ψ be a linear inequality:

$$\psi \triangleq c_1x_1 + \cdots + c_nx_n + d \leq 0$$

Then $S \Rightarrow \Psi$ is valid iff S is unsatisfiable or there exist $m+1$ real nonnegative coefficients $\lambda_0, \lambda_1, \dots, \lambda_m$ such that:

$$c_j = \sum_{i=1}^m \lambda_i a_{ij} \quad (1 \leq j \leq m) \quad d = -\lambda_0 + \sum_{i=1}^m \lambda_i b_i$$

Constraint-based Inv. Inference: Example



Use Farkas's lemma to turn the consecution constraint:

$$\begin{aligned} \top [x/x; n/n] \wedge x < n \wedge x' = x + 1 \wedge n' = n \\ \Rightarrow \top [x'/x; n'/n] \end{aligned}$$

into a constraint over c , d , and e only.

μ_1	cx	$+dn$		$+e$	≤ 0
λ_1	x	$-n$		$+1$	≤ 0
λ_2	$-x$		$+x'$	-1	≤ 0
λ_3	x		$-x'$	$+1$	≤ 0
λ_4		$-n$		$+n'$	≤ 0
λ_5		n		$-n'$	≤ 0
.....					
			cx'	$+dn'$	$+e \leq 0$

Constraint-based Inv. Inference: Example



μ_1	cx	$+dn$		$+e$	≤ 0
λ_1	x	$-n$		$+1$	≤ 0
λ_2	$-x$		$+x'$	-1	≤ 0
λ_3	x		$-x'$	$+1$	≤ 0
λ_4		$-n$		$+n'$	≤ 0
λ_5		n		$-n'$	≤ 0
.....					
			cx'	$+dn'$	$+e \leq 0$

$$\Phi \triangleq \exists \mu_1, \lambda_0, \dots, \lambda_5 \left[\begin{array}{l} \mu_1, \lambda_0, \dots, \lambda_5 \geq 0 \\ \mu_1 c + \lambda_1 - \lambda_2 + \lambda_3 = 0 \\ \mu_1 d - \lambda_1 - \lambda_4 + \lambda_5 = 0 \\ \lambda_2 - \lambda_3 = c \\ \lambda_4 - \lambda_5 = d \\ -\lambda_0 + \mu_1 e + \lambda_1 - \lambda_2 + \lambda_3 = e \end{array} \right]$$

Constraint-based Inv. Inference: Example



$$\Phi \triangleq \exists \mu_1, \lambda_0, \dots, \lambda_5 \left[\begin{array}{l} \mu_1, \lambda_0, \dots, \lambda_5 \geq 0 \\ \mu_1 c + \lambda_1 - \lambda_2 + \lambda_3 = 0 \\ \mu_1 d - \lambda_1 - \lambda_4 + \lambda_5 = 0 \\ \lambda_2 - \lambda_3 = c \\ \lambda_4 - \lambda_5 = d \\ -\lambda_0 + \mu_1 e + \lambda_1 - \lambda_2 + \lambda_3 = e \end{array} \right]$$

Finally, **eliminate existential quantifiers** from Φ to get the constraint:

$$c \leq 0 \vee (c + d = 0 \wedge e \leq 0)$$

- (Quantifier elimination is also quite technical)

Constraint-based Inv. Inference: Example



```
dummy_routine (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end
```

- Any solution $[c, d, e]$ to:

- Initiation and Consecution:

$$(d \cdot n_0 + e \leq 0) \wedge (c \leq 0 \vee (c + d = 0 \wedge e \leq 0))$$

determines an **invariant** of the loop.

- $[0, -1, 0]$ ----> $n \geq 0$
- $[1, 0, 0]$ ----> $x \geq 0$
- $[1, -1, 0]$ ----> $x - n \leq 0$

Constraint-based Inv. Inference: Summary



- **Main issues:**
 - choice of invariant templates for which effective decision procedures exist
 - interesting research topic per se, on the brink of undecidability
 - heuristics to extract the “best” invariants from the set of solutions
- **Advantages:**
 - sound
 - complete (w.r.t. the template)
 - exploit heterogeneous decision procedures syncretically
 - fully automated (possibly except for providing the template)
 - providing the template introduces a “natural” form of interaction with the user
- **Disadvantages:**
 - suitable mathematical decision theories are usually quite sophisticated
 - hence, hard to extend and customize
 - exact constraint solving is usually quite expensive
 - mostly suitable for “algebraic” invariants
 - requires integration with other techniques to achieve full functional correctness proofs



Dynamic Techniques for Invariant Inference

Dynamic Invariant Inference



- In a nutshell:

testing of candidate invariants

1. Choose a set of test cases
2. Perform runtime monitoring of candidate invariants
3. If some test run violates a candidate, discard the candidate
4. The surviving candidates are guessed invariant

- Daikon tool, 1999 -- Mike Ernst et al.
- CITADEL: Daikon for Eiffel, 2008 -- Nadia Polikarpova
- AutoInfer for Eiffel

Dynamic Invariant Inference: Example

```
dummy_routine (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end
```

- Test cases: $\{ n = k \mid 0 \leq k \leq 1000 \}$
- Candidate invariants:
 - $\{ x \geq c \mid -1000 \leq c \leq 1000 \}$,
 $\{ n \geq c \mid -1000 \leq c \leq 1000 \}$
 - $\{ x = c \cdot n + d \mid -500 \leq c, d \leq 500 \}$
 - $\{ x < n, x \leq n, x = n, x \neq n, x \geq n, x > n \}$
 - $\{ x \pm n \geq c \mid -500 \leq c \leq 500 \}$
 - ...

Dynamic Invariant Inference: Example



```
dummy_routine (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end
```

- **Survivors** (after loop iterations) :

- $\{ x \geq -c \mid 0 \leq c \leq 1000 \},$
 $\{ n \geq -c \mid 0 \leq c \leq 1000 \}$
- $x \leq n$
- $\{ x + n \geq c \mid -500 \leq c \leq 500 \}$
- ...

Dynamic Invariant Inference: Summary



- **Main issues:**
 - choose suitable test cases
 - handle huge sets of candidate invariants (runtime overhead)
 - estimate soundness/quality of survivor predicates
 - select heuristically the “best” survivor predicates
- **Advantages:**
 - straightforward to implement (at least compared to other techniques)
 - guessing is often rather accurate in practice (possibly with some heuristics)
 - customizable and rather flexible:
in principle, whatever you can test you can check for invariance
- **Disadvantages:**
 - unsound (educated guessing)
 - without heuristics, large amount of useless, redundant predicates
 - sensitive to choice of test cases
 - some complex candidate invariants are difficult to implement efficiently