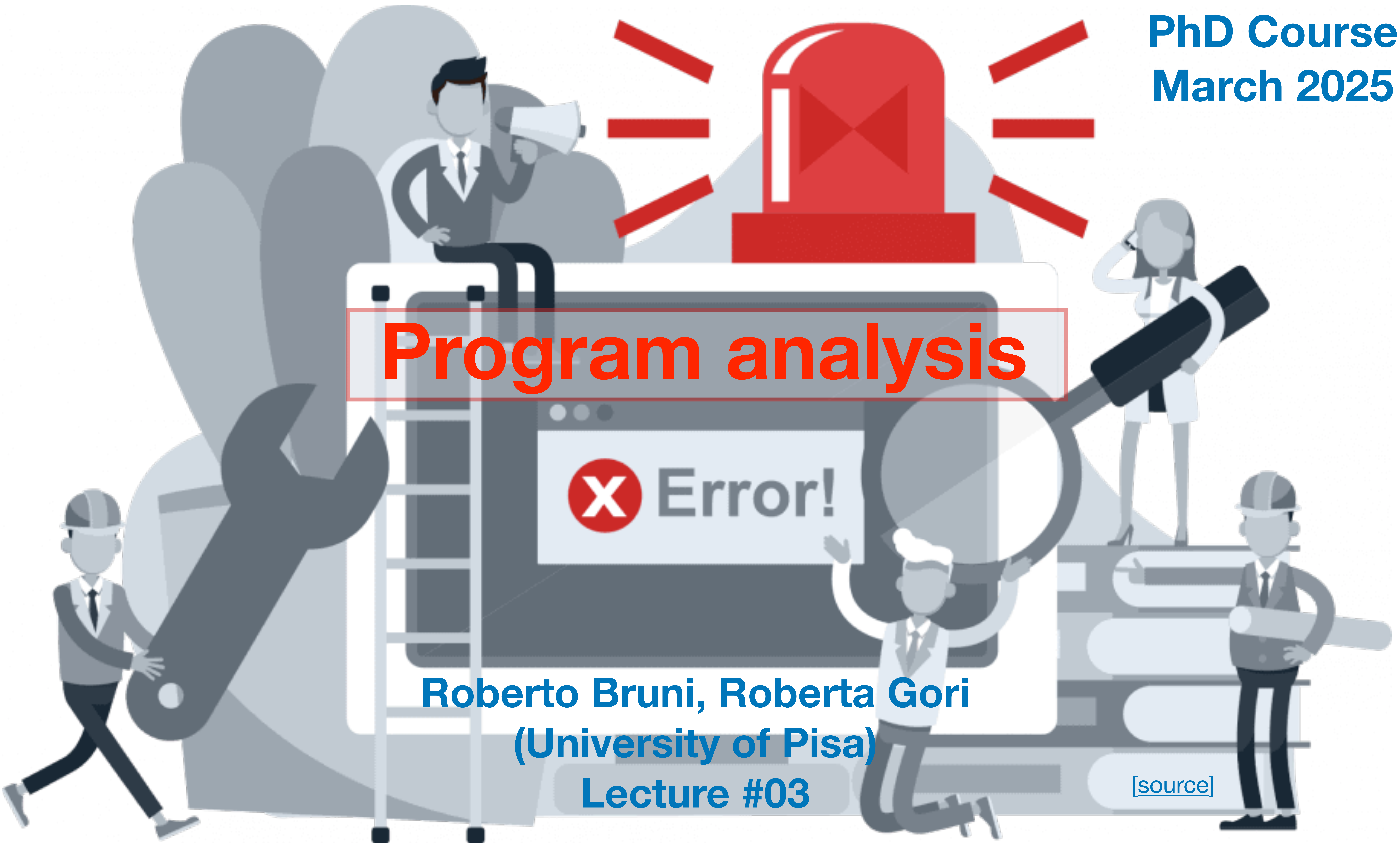
Program analysis



PhD Course
March 2025

Roberto Bruni, Roberta Gori
(University of Pisa)

Lecture #03

[source]

Unexpected result?

Behind any programming activity there is an expectation

What the program should do
Sort an array, find the maximum …

What the program should not do
Read private data, divide by zero …
Loop forever, Irresponsive UI …

The expectation is the program specification

What is programming about?

Specification

What my code should do and should not do

Hacking

Write the code

Debugging/Verifying

“Prove” my code follows its specification

Specification

Can be informal

No hang, no infinite loops, no crashes, no wrong output …

Can be formal

Simplest: Test cases

What I expect in some finite cases

More complex: Formal specification language

Debugging/Verifying

Runtime analysis

Run the program, observe the behavior for some specific runs
Check if the behavior violates the specification

Static analysis/verification

Do not run the program, observe the properties for all runs
Check if the behavior meets the specification

Example: Abs

Which is the specification for this code?

```
public int32 Abs(int32 x)
{
 if (x < 0)
   return -x;
 else
   return x;
}
```

Reminder:
int32 = $[-2^{31}, 2^{31} - 1]$
$-(-2^{31}) = -2^{31}$

Overview

Problem: Automatic inference of preconditions

Which preconditions?

Sufficient Precondition: if it holds, the code is correct

Necessary Precondition: if it does not hold the code is never correct

Sufficient preconditions

```
int Example1(int x,
             object[] a)
{
if (x >= 0)
   {
      return a.length;
   }
return -1;
}
```

Sufficient precondition: `a != null`

Too strong for the caller

No runtime errors when
`x < 0` and `a == null`

Users of verification tools complained about it
"wrong preconditions"

Necessary preconditions

```
int Example2 (object[ ]  a)
{  for (int i=0; i<=a.length;  i++)
       {
           a[i]=f(a[i]);
           if ( nondet() )
              return;
       }
}
```

Sufficient precondition: `false`

The function may fail
So eliminate all runs!

Necessary precondition: `0 < a.length`

If `0==a.length` then it will always fail!

Necessary preconditions

When automatic inference is considered, only necessary preconditions make sense

Sufficient preconditions impose too large a burden to callers

Necessary preconditions are easy to explain to users

Implemented in the contract checker verifier Clousot (Microsoft)
Precision improvements 9% to 21%
Extremely low false positive ratio

Necessary conditions (NC)

VMCAI 2013

Automatic Inference of Necessary Preconditions

Patrick Cousot[1], Radhia Cousot[2], Manuel Fähndrich[3], and Francesco Logozzo[3]

[1] NYU, ENS, CNRS, INRIA
pcousot@cims.nyu.edu
[2] CNRS, ENS, INRIA
rcousot@ens.fr
[3] Microsoft Research
{maf,logozzo}@microsoft.com

Abstract. We consider the problem of *automatic* precondition inference. We argue that the common notion of *sufficient* precondition inference (*i.e.*, under which precondition is the program correct?) imposes too large a burden on callers, and hence it is unfit for automatic program analysis. Therefore, we define the problem of *necessary* precondition inference (*i.e.*, under which precondition, if violated, will the program *always* be incorrect?). We designed and implemented several new abstract interpretation-based analyses to infer atomic, disjunctive, universally and existentially quantified necessary preconditions.

We experimentally validated the analyses on large scale industrial code. For unannotated code, the inference algorithms find necessary preconditions for almost 64% of methods which contained warnings. In 27% of these cases the inferred preconditions were also *sufficient*, meaning all warnings within the method body disappeared. For annotated code, the inference algorithms find necessary preconditions for over 68% of methods with warnings. In almost 50% of these cases the preconditions were also sufficient. Overall, the precision improvement obtained by precondition inference (counted as the additional number of methods with no warnings) ranged between 9% and 21%.

1 Introduction

Design by Contract [28] is a programming methodology which systematically requires the programmer to provide the preconditions, postconditions and object invariants (collectively called contracts) at design time. Contracts allow automatic generation of documentation, amplify the testing process, and naturally enable assume/guarantee reasoning for divide and conquer static program analysis and verification. In the real world, relatively few methods have contracts that are sufficient to prove the method correct. Typically, the precondition of a method is weaker than necessary, resulting in unproven assertions within the method, but making it easier to prove the precondition at call-sites. Inference has been advocated as the holy grail to solve this problem.

In this paper we focus on the problem of computing *necessary preconditions* which are *inevitable checks* from within the method that are hoisted to the

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 128–148, 2013.

"Under which precondition, if violated, will the program always be incorrect?"




Backward Analysis

Regular commands

regular command

atomic command

choice

Kleene star

$$r ::= e \mid r_1; r_2 \mid r_1 + r_2 \mid r^\star$$

$$e ::= \text{skip} \mid x := a \mid b? \mid \ldots$$

Syntactic sugar

if b then c_1 else c_2 $\triangleq (b?; c_1) + (\neg b?; c_2)$

while b do c $\triangleq (b?; c)^\star; \neg b?$

Backward analysis

Forward Analysis

```
int Simple (bool b)
 {   int z;
      if (b)
            z := 12; z∈ [12,12]
 else
            z := -12;  z∈ [-12,-12]
             z∈ [-12,12]
      return  1/z;
 }
```

Possible division by 0

Backward Analysis

```
int Simple (bool b)
 {   int z;
      if (b)
        z := 12;
        else
        z := -12;

      return  1/z;
 }
```

$z \neq 0$

$z \neq 0$

$z \neq 0$

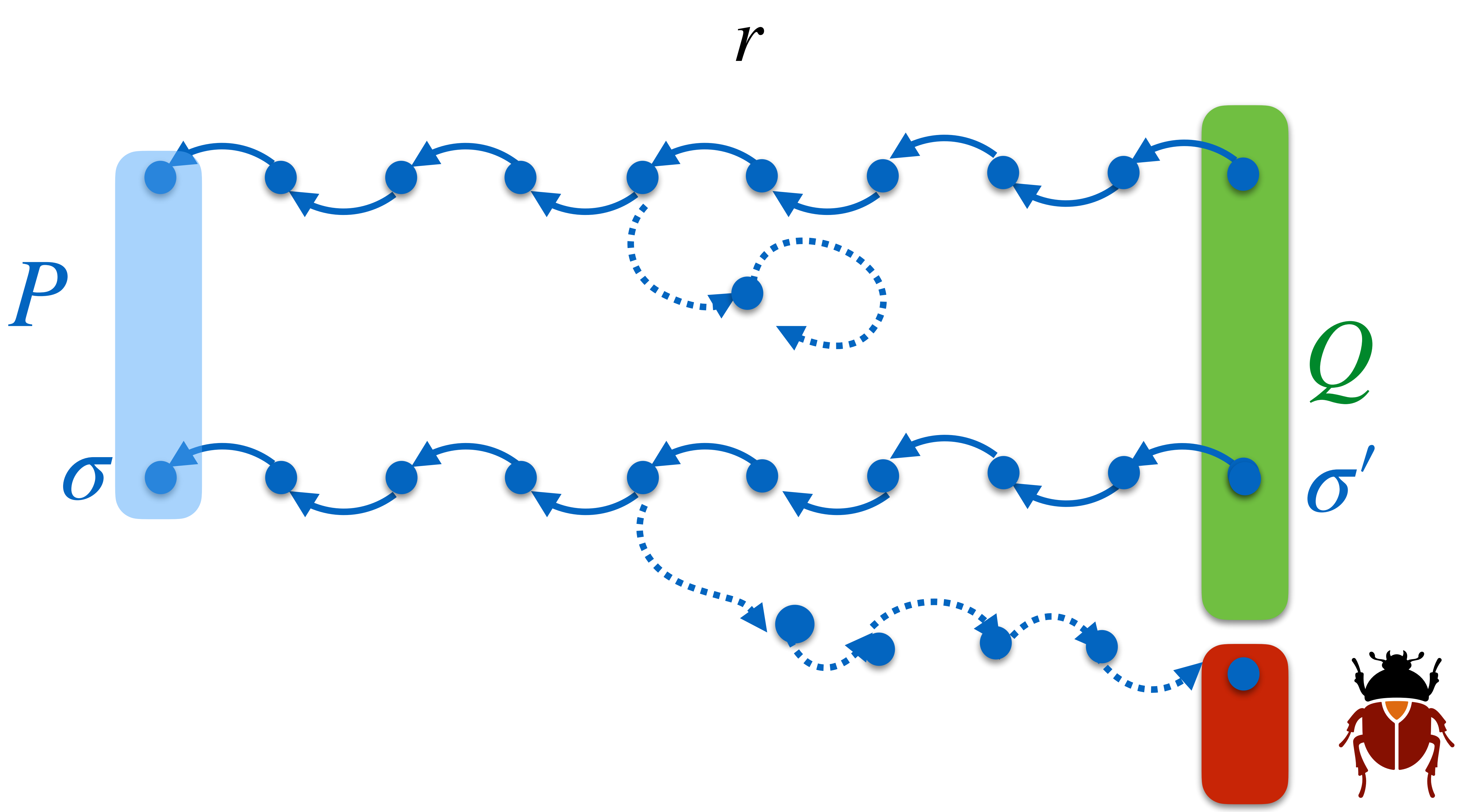
$z \neq 0$

Backward semantics

$$[\![\overleftarrow{r}]\!]\sigma' \triangleq \{\sigma \mid \sigma' \in [\![r]\!]\sigma\}$$

$$\sigma \in [\![\overleftarrow{r}]\!]\sigma' \Leftrightarrow \sigma' \in [\![r]\!]\sigma$$

Different from WLP!



As before we can extend it to sets

$$[\![\overleftarrow{r}]\!]Q = P$$

Example

$c \triangleq$

```
Divisor_of(x)
{
…
s := nondet[2..x/2];
if (x%s=0)
    skip
else
    while true do skip
}
```

$$wlp(c, Q) \triangleq \{\sigma \mid [\![c]\!]\{\sigma\} \subseteq Q\}$$

$$wlp(c, [s = 5, x = 17]) = \text{???}\ \{x\ \text{prime}\} \cup \{0,1\}$$

$$wlp(c, [s = 5, x = 15]) = \text{???}\ \{x\ \text{prime}\} \cup \{0,1\}$$

$$[\![\overleftarrow{c}]\!]Q \triangleq \{\sigma \mid \sigma' \in [\![c]\!]\sigma, \sigma' \in Q\}$$

$$[\![\overleftarrow{c}]\!][s = 5, x = 17] = \text{???}\ \emptyset$$

$$[\![\overleftarrow{c}]\!][s = 5, x = 15] = \text{???}\ (x = 15)$$

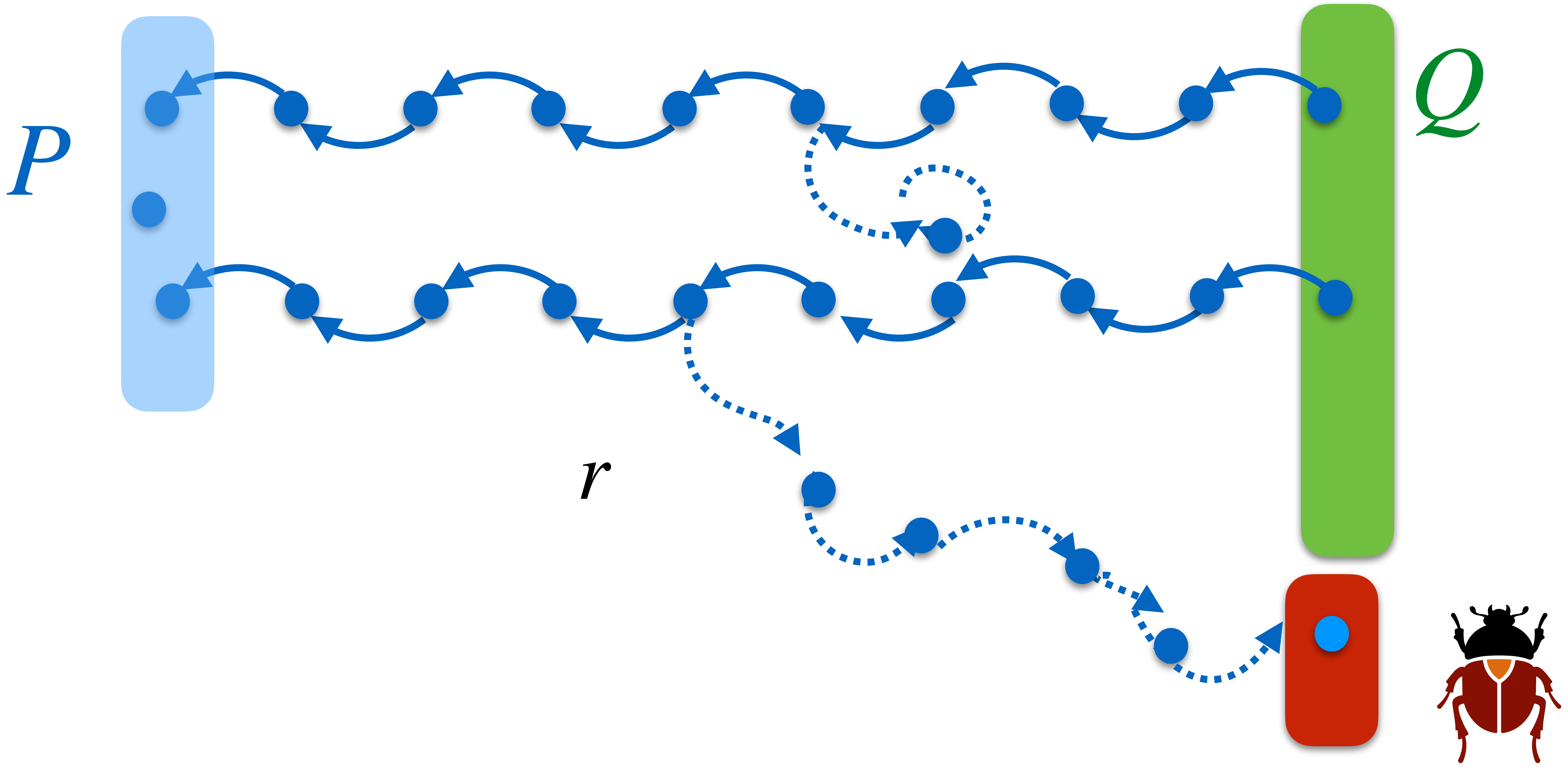
Necessary preconditions

The idea of NC is to prevent the invocation of the function with arguments that will inevitably lead to some error

Given Q the set of good final states, the NC triple

$$(P)\, r\, (Q)$$

means that any state $\sigma \in P$ may admit at least one non-erroneous execution of r.



$$[\![\overleftarrow{r}]\!] Q \subseteq P$$

It is an over-approximation!

Compare over approximation logics

	Forward	Backward
Over	{HL} $[\![r]\!]P \subseteq Q$	(NC) $[\![\overleftarrow{r}]\!]Q \subseteq P$

HL vs NC: Consequence rules

{HL}

$$[\![r]\!]P \subseteq Q$$

(NC)

$$[\![\overleftarrow{r}]\!]Q \subseteq P$$

There's not a proof system but..

$$\frac{P \Rightarrow P' \quad \{P'\}\, r\, \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\, r\, \{Q\}}$$

$$\frac{P' \Rightarrow P \quad (P')\, r\, (Q') \quad Q \Rightarrow Q'}{(P)\, r\, (Q)}$$

HL vs NC: weakest/strongest pre and post

Given P, strongest Q

{HL} $[\![r]\!]P \subseteq Q$

Given Q, weakest P

Given Q, strongest P

(NC) $[\![\overleftarrow{r}]\!]Q \subseteq P$

Given P , weakest Q

HL vs NC: relation

$$\{P\}r\{Q\} \iff (\neg P)r(\neg Q)$$

That means

$$[\![r]\!]P \subseteq Q \iff [\![\overleftarrow{r}]\!]\neg Q \subseteq \neg P$$

The proof

One implication, the other is similar

$$[\![r]\!]P \subseteq Q \implies [\![\overleftarrow{r}]\!]\neg Q \subseteq \neg P$$

$\sigma' \in \neg Q \implies \sigma' \notin Q$

Hence, $\sigma' \notin [\![r]\!]P \iff \forall \sigma \in P.\sigma' \notin [\![r]\!]\sigma$

Since $\sigma \in [\![\overleftarrow{r}]\!]\sigma' \Leftrightarrow \sigma' \in [\![r]\!]\sigma$ $\iff \forall \sigma \in P.\sigma \notin [\![\overleftarrow{r}]\!]\sigma'$

$\iff P \cap [\![\overleftarrow{r}]\!]\sigma' = \emptyset$

$\iff [\![\overleftarrow{r}]\!]\sigma' \subseteq \neg P$

Since it holds for all $\sigma' \in Q$ $\iff [\![\overleftarrow{r}]\!]\neg Q \subseteq \neg P$

Questions

Question 1

Let $c \triangleq (z := x) + (z := y)$
and let $Q \triangleq (z = 0)$

What is $wlp(c, Q)$? $(x = y = 0)$

What is $[\![\overleftarrow{c}]\!]Q$? $(x = 0 \vee y = 0)$

Question 2

Recalling that both $\{\mathsf{false}\}\ c\ \{Q\}$ and $\{P\}\ c\ \{\mathsf{true}\}$
are valid HL triples (for any P, Q and c)
can we claim something about the validity of NC triples such as

$$(\mathsf{false})\ c\ (Q) \iff \{\mathsf{true}\}\ c\ \{\neg Q\}$$

$$(P)\ c\ (\mathsf{true}) \iff \{\neg P\}\ c\ \{\mathsf{false}\}$$

$$(\mathsf{true})\ c\ (Q) \iff \{\mathsf{false}\}\ c\ \{\neg Q\}$$

$$(P)\ c\ (\mathsf{false}) \iff \{\neg P\}\ c\ \{\mathsf{true}\}$$