Abstract Execution

23rd Symposium on Formal Methods, Porto, Portugal

Dominic Steinhöfel and Reiner Hähnle
steinhoefel@cs.tu-darmstadt.de

October 10th, 2019

Software Engineering Group, Computer Science Department, TU Darmstadt

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

Abstract Execution

Properties of Concrete Programs: Functional Program Verification

```
//@ ensures \result >= 0;
public int abs(int a, int b) {
  if (a < b) {
    int tmp = a;
    a = b;
    b = tmp;
  }

  return a - b;
}
```

Properties of Many Programs: Refactorings

```
if (b) {
  P
  Q1
} else {
  P
  Q2
}
```

Properties of Many Programs:
Refactorings

```
if (b) {
   P
   Q1
} else {
   P
   Q2
}
```

$$\xrightarrow[\text{to}]{\text{refactored}}$$

Properties of Many Programs: Refactorings

```
if (b) {
   P
   Q1
} else {
   P
   Q2
}
```

$\xrightarrow[\text{to}]{\text{refactored}}$

```
P
if (b) {
   Q1
} else {
   Q2
}
```

Hierarchy of Verification Approaches

Testing	Show correctness of **one** program for **one** set of inputs

Hierarchy of Verification Approaches

Testing	Show correctness of **one** program for **one** set of inputs
Program Proving	Show correctness of **one** program for **all** possible inputs

Hierarchy of Verification Approaches

Testing	Show correctness of **one** program for **one** set of inputs
Program Proving	Show correctness of **one** program for **all** possible inputs
Abstract Program Proving	Show correctness of **all** programs for **all** possible inputs (matching a pattern).

Hierarchy of Verification Approaches

Testing	Show correctness of **one** program for **one** set of inputs
Program Proving	Show correctness of **one** program for **all** possible inputs
Abstract Program Proving	Show correctness of **all** programs for **all** possible inputs (matching a pattern).

Abstract Programs =
Programs with **Abstract Placeholder Statements (APSs)**

Abstract Execution

How does one show the correctness of an abstract program?

```
Inductive com : Type :=
  | CSkip : com
  | CAss : pvs → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.

Inductive ceval : com → state → state → Prop :=
  | E_Skip : forall st,
      SKIP / st \\ st
  | E_Ass : forall st a1 n x,
      aeval st a1 = n →
      (x =! a1) / st \\ (s_update st x n)
  (* ... *)
```

How does one show the correctness of an abstract program?

```
Theorem evaluation_deterministic:
  ∀ c st st1 st2,
    c / st \\ st1  → c / st \\ st2 → st1 = st2.
Proof.
  intros c st st1 st2 H1 H2.
  generalize dependent st2.
  induction H1.
  - (* E_Skip *) reflexivity.
  - (* E_Ass *)  reflexivity.
  - (* ... *)
```

Abstract Program Proofs by Structural Induction

- Frequently practiced in
  - **pen-and-paper** proofs and
  - **interactive theorem provers** like Isabelle and Coq (e.g., **CompCert** [Ler09] and **CakeML** [TMK+16])
- Precise **second-order reasoning** over program properties
- ...but very **hard to automate**!

Goal:

Goal: Automatic Reasoning

Goal: Automatic Reasoning about Universal Properties of Abstract Programs

Goal: Automatic Reasoning about Universal Properties of Abstract Programs in an Industrial Programming Language

Goal: **Automatic** Reasoning about **Universal** Properties of **Abstract** Programs in an **Industrial Programming Language**

- Use **Symbolic Execution** with **abstract state changes**

Goal: Automatic Reasoning about Universal Properties of Abstract Programs in an Industrial Programming Language

- Use **Symbolic Execution** with **abstract state changes**
- Model **irregular termination**
  (exceptions, (labeled) breaks, (labeled) continues, returns)

Goal: Automatic Reasoning about Universal Properties of Abstract Programs in an Industrial Programming Language

- Use **Symbolic Execution** with **abstract state changes**
- Model **irregular termination**
  (exceptions, (labeled) breaks, (labeled) continues, returns)
- Retain **sufficient precision** due to fine-grained **specification language**

Goal: Automatic Reasoning about Universal Properties of Abstract Programs in an Industrial Programming Language

- Use **Symbolic Execution** with **abstract state changes**
- Model **irregular termination**
  (exceptions, (labeled) breaks, (labeled) continues, returns)
- Retain **sufficient precision** due to fine-grained **specification language**
- Case study: Correctness of **refactoring techniques**

Specification of APSs +
Symbolic Execution of APSs +
Simplification of Abstract State Changes

Abstract Execution

Specification of APSs +
Symbolic Execution of APSs +
Simplification of Abstract State Changes

Example: Extract Prefix Refactoring

```
if (b) {
   P
   Q1
} else {
   P
   Q2
}
```

$\xrightarrow[\text{to}]{\text{refactored}}$

```
P
if (b) {
   Q1
} else {
   Q2
}
```

Martin Fowler: Refactoring - Improving the Design of Existing Code. Addison-Wesley 1999

Declaring a Program with Abstract Placeholders

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

Declaring a Program with Abstract Placeholders

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

Declaring a Program with Abstract Placeholders

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

Declaring a Program with Abstract Placeholders

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

Declaring a Program with Abstract Placeholders

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
abstract_statement P;

abstract_statement Init;
if (b) {

    abstract_statement Q1;
} else {

    abstract_statement Q2;
}
```

Declaring a Program with Abstract Placeholders

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
abstract_statement P;

abstract_statement Init;
if (b) {

    abstract_statement Q1;
} else {

    abstract_statement Q2;
}
```

Add Specifications to Constrain Represented Programs

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

Add Specifications to Constrain Represented Programs

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
if (b) {

} else {

}
```

Add Specifications to Constrain Represented Programs

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
b = x < 0;
if (b) {


} else {


}
```

Add Specifications to Constrain Represented Programs

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
b = x < 0;
if (b) {
    result = y/2;

} else {
    result = y/2;

}
```

Add Specifications to Constrain Represented Programs

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
b = x < 0; x = 42;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ assignable b;
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ assignable b;
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ assignable hasTo(b);
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ assignable hasTo(b);
abstract_statement Init;
if (b) {
    abstract_statement P;
    abstract_statement Q1;
} else {
    abstract_statement P;
    abstract_statement Q2;
}
```

```
b = x < 0;
if (b) {
    x = -1;
    x = -x + result;
} else {
    x = -1;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
Object abstractMethod() {
    // ...

    //@ assignable hasTo(b);

    abstract_statement Init;
    if (b) {


        abstract_statement P;


        abstract_statement Q1;
    } else {


        abstract_statement P;


        abstract_statement Q2;
    }

    // ...
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ declares final(args);
Object abstractMethod() {
    // ...

    //@ assignable hasTo(b);

    abstract_statement Init;
    if (b) {


        abstract_statement P;


        abstract_statement Q1;
    } else {


        abstract_statement P;


        abstract_statement Q2;
    }

    // ...
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ declares final(args);
Object abstractMethod() {
    // ...

    //@ assignable hasTo(b);
    //@ accessible args;
    abstract_statement Init;
    if (b) {
        //@ assignable result;

        abstract_statement P;


        abstract_statement Q1;
    } else {
        //@ assignable result;

        abstract_statement P;


        abstract_statement Q2;
    }

    // ...
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ declares final(args);
Object abstractMethod() {
    // ...

    //@ assignable hasTo(b);
    //@ accessible args;
    abstract_statement Init;
    if (b) {
        //@ assignable result;
        //@ accessible result, args;
        abstract_statement P;


        abstract_statement Q1;
    } else {
        //@ assignable result;
        //@ accessible result, args;
        abstract_statement P;


        abstract_statement Q2;
    }

    // ...
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ declares final(args);
Object abstractMethod() {
    // ...

    //@ assignable hasTo(b);
    //@ accessible args;
    abstract_statement Init;
    if (b) {
        //@ assignable result;
        //@ accessible result, args;
        abstract_statement P;
        //@ assignable \everything;
        //@ accessible \everything;
        abstract_statement Q1;
    } else {
        //@ assignable result;
        //@ accessible result, args;
        abstract_statement P;
        //@ assignable \everything;
        //@ accessible \everything;
        abstract_statement Q2;
    }

    // ...
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

```
//@ declares final(args);
Object abstractMethod() {
    // ...

    //@ assignable hasTo(b);
    //@ accessible args;
    abstract_statement Init;
    if (b) {
        //@ assignable result;
        //@ accessible result, args;
        abstract_statement P;
        //@ assignable \everything;
        //@ accessible \everything;
        abstract_statement Q1;
    } else {
        //@ assignable result;
        //@ accessible result, args;
        abstract_statement P;
        //@ assignable \everything;
        //@ accessible \everything;
        abstract_statement Q2;
    }

    // ...
}
```

```
b = x < 0;
if (b) {
    result = y/2;
    x = -x + result;
} else {
    result = y/2;
    x = x + result;
}
```

Add Specifications to Constrain Represented Programs

Prohibit Abrupt Completion Behavior

```
//@ return_behavior requires false;
//@ exceptional_behavior requires false;
//@ continue_behavior requires false;
//@ break_behavior requires false;
...
```

Add Specifications to Constrain Represented Programs

Bind Abrupt Completion Behavior to Formula

```
//@ return_behavior requires returnsSpec;
//@ exceptional_behavior requires excSpec;
//@ continue_behavior requires contSpec;
//@ break_behavior requires breaksSpec;
...
```

Specification Constructs for APSs

Spec. Construct	Explanation
`locals``(P)`	Refers to the Skolem (abstract) location set of local variables of an APS with symbol `P` visible from outside.
`declares` *skLocs*`;`	Specifies that an APS/method declares a list *skLocs* of Skolem location set specifiers **`locals`**`(·)`, opt. wrapped in **`final`**`(·)` modifiers, which can be used in APSs in the visible scope afterwards.
`assignable` *locs*`;`	Declares the location set *locs* to be assignable by the APS. *locs* is a list of variables, fields, and Skolem location set specifiers, optionally wrapped in a **`hasTo`**`(·)` modifier.
`accessible` *locs*`;`	Declares *locs* to be accessible by the APS.
`return_behavior requires` φ`;`	Specifies that the APS returns iff φ holds.
`exceptional_behavior requires` φ`;`	Spec. that the APS throws an exc. iff φ holds.
`break_behavior requires` φ`;` **`continue_behavior requires`** φ`;`	Specifies that the APS breaks/continues during loop execution iff φ holds.
`break_behavior` `(`*lbl*`)` **`requires`** φ`;` **`continue_behavior`** `(`*lbl*`)` **`requires`** φ`;`	Specifies that the APS breaks/continues to the (loop) label *lbl* iff φ holds.

Abstract Execution

Specification of APSs +

Symbolic Execution of APSs +

Simplification of Abstract State Changes

Symbolic Execution of an Assignment (in JavaDL)

`x=`e`;`

Symbolic Execution of an Assignment (in JavaDL)

$$[\ \ \texttt{x=}e\texttt{;}\ \ ]\phi$$

Symbolic Execution of an Assignment (in JavaDL)

$$\frac{\{\mathtt{x} := e\}[\quad]\phi}{[\quad \mathtt{x}\texttt{=}e\texttt{;} \quad]\phi}$$

Symbolic Execution of an Assignment (in JavaDL)

$$\frac{\{\mathsf{x} := e\}[\pi\ \omega]\phi}{[\pi\ \mathtt{x=}e\mathtt{;}\ \omega]\phi}$$

Symbolic Execution of an Assignment (in JavaDL)

$$\text{assignment} \quad \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{x} := e\}[\pi\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x}\texttt{=}e\texttt{;}\ \omega]\phi, \Delta}$$

Symbolic Execution of a Conditional Statement (in JavaDL)

$$[\ \texttt{if}\ (e)\ p_1\ \texttt{else}\ p_2\ ]\varphi$$

Symbolic Execution of a Conditional Statement (in JavaDL)

$$\frac{e \doteq \mathrm{TRUE} \Longrightarrow \quad [\ p_1\ ]\varphi}{[\ \texttt{if}\ (e)\ p_1\ \texttt{else}\ p_2\ ]\varphi}$$

Symbolic Execution of a Conditional Statement (in JavaDL)

$$\frac{e \doteq \mathrm{FALSE} \Longrightarrow [\ p_2\ ]\varphi}{[\ \texttt{if}\ (e)\ p_1\ \texttt{else}\ p_2\ ]\varphi}$$

IfElseSplit

$$\frac{\Gamma, e \doteq \mathrm{TRUE} \Longrightarrow \{\mathcal{U}\}[\pi\ p_1\ \omega]\varphi, \Delta \qquad \Gamma, e \doteq \mathrm{FALSE} \Longrightarrow \{\mathcal{U}\}[\pi\ p_2\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{if (}e\texttt{)}\ p_1\ \texttt{else}\ p_2\ \omega]\varphi, \Delta}$$

A Very Simple Symbolic Execution Rule for Abstract Execution

$[$ `abstract_statement P;` $]\phi$

Dominic Steinhöfel, Reiner Hähnle: Modular, Correct Compilation with Automatic Soundness Proofs. ISoLA 2018

A Very Simple Symbolic Execution Rule for Abstract Execution

$$\frac{\{\mathcal{U}_{\mathsf{P}} \qquad\qquad\qquad\qquad \} \qquad\qquad\qquad [\quad]\phi}{[\ \texttt{abstract\_statement P;}\ ]\phi}$$

Dominic Steinhöfel, Reiner Hähnle: Modular, Correct Compilation with Automatic Soundness Proofs. ISoLA 2018

A Very Simple Symbolic Execution Rule for Abstract Execution

$$\frac{\{\mathcal{U}_{\mathsf{P}}(allLocs :\approx allLocs)\} \qquad [\quad]\phi}{[\ \texttt{abstract\_statement P;}\ ]\phi}$$

Dominic Steinhöfel, Reiner Hähnle: Modular, Correct Compilation with Automatic Soundness Proofs. ISoLA 2018

A Very Simple Symbolic Execution Rule for Abstract Execution

$$\frac{\{\mathcal{U}_{\mathsf{P}}(allLocs :\approx allLocs)\} \qquad [\quad]\phi}{[\ \texttt{abstract\_statement P;}\ ]\phi}$$

Dominic Steinhöfel, Reiner Hähnle: Modular, Correct Compilation with Automatic Soundness Proofs. ISoLA 2018

A Very Simple Symbolic Execution Rule for Abstract Execution

$$\frac{\{\mathcal{U}_{\mathsf{P}}(allLocs :\approx allLocs)\} \qquad\qquad [\quad]\phi}{[\ \texttt{abstract\_statement P;}\ ]\phi}$$

Dominic Steinhöfel, Reiner Hähnle: Modular, Correct Compilation with Automatic Soundness Proofs. ISoLA 2018

A Very Simple Symbolic Execution Rule for Abstract Execution

$$\frac{\{\mathcal{U}_{\mathsf{P}}(allLocs :\approx allLocs)\}(\mathsf{C}_{\mathsf{P}}(allLocs) \rightarrow [\quad]\phi)}{[\ \texttt{abstract\_statement P;}\ ]\phi}$$

Dominic Steinhöfel, Reiner Hähnle: Modular, Correct Compilation with Automatic Soundness Proofs. ISoLA 2018

A Very Simple Symbolic Execution Rule for Abstract Execution

$$\frac{\{\mathcal{U}_{\mathsf{P}}(allLocs :\approx allLocs)\}(\mathsf{C}_{\mathsf{P}}(allLocs) \rightarrow [\quad]\phi)}{[\ \texttt{abstract\_statement P;}\ ]\phi}$$

Dominic Steinhöfel, Reiner Hähnle: Modular, Correct Compilation with Automatic Soundness Proofs. ISoLA 2018

A Very Simple Symbolic Execution Rule for Abstract Execution

simpleAERule

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathcal{U}_{\mathsf{P}}(allLocs :\approx allLocs)\}(\mathsf{C}_{\mathsf{P}}(allLocs) \rightarrow [\pi\ \omega]\phi), \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{abstract\_statement P;}\ \omega]\phi, \Delta}$$

Dominic Steinhöfel, Reiner Hähnle: Modular, Correct Compilation with Automatic Soundness Proofs. ISoLA 2018

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathrm{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$		

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathrm{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	“Illegal”
$\mathcal{U}_{\mathrm{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^{!}, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$		

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$		

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$		

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	—

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	—
$C_{\mathtt{P}}(\mathtt{x}, \mathtt{y}, \mathtt{z})$		

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	—
$C_{\mathtt{P}}(\mathtt{x}, \mathtt{y}, \mathtt{z})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	—
$C_{\mathtt{P}}(\mathtt{x}, \mathtt{y}, \mathtt{z})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	x¡w

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^{!}, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \parallel \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^{!}, \mathtt{y} :\approx \ )$	$\mathtt{x} := 1 \parallel \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \parallel \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	—
$C_{\mathtt{P}}(\mathtt{x}, \mathtt{y}, \mathtt{z})$	$\mathtt{x} > 0 \land \mathtt{x} < \mathtt{y}$	x¡w
$C_{\mathtt{P}}()$		

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^!, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^!, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$	$\mathtt{x} > 0 \wedge \mathtt{x} < \mathtt{y}$	—
$C_{\mathtt{P}}(\mathtt{x}, \mathtt{y}, \mathtt{z})$	$\mathtt{x} > 0 \wedge \mathtt{x} < \mathtt{y}$	x¡w
$C_{\mathtt{P}}()$	true	

Towards a Soundness Notion: Instantiating Abstract Updates and Path Conditions

Abstract Symbol	Example Instantiation	"Illegal"
$\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})$	$\mathtt{x} := \mathtt{y} + 1$	—
$\mathcal{U}_{\mathtt{Q}}(\mathtt{x}^{!}, \mathtt{y} :\approx \mathtt{x}, \mathtt{z})$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$	$\mathtt{y} := 12$
$\mathcal{U}_{\mathtt{R}}(\mathtt{x}^{!}, \mathtt{y} :\approx\ )$	$\mathtt{x} := 1 \,\|\, \mathtt{y} := 12$	$\mathtt{x} := \mathtt{x} + 1 \,\|\, \mathtt{y} := 12$
$C_{\mathtt{P}}(\mathit{allLocs})$	$\mathtt{x} > 0 \wedge \mathtt{x} < \mathtt{y}$	—
$C_{\mathtt{P}}(\mathtt{x}, \mathtt{y}, \mathtt{z})$	$\mathtt{x} > 0 \wedge \mathtt{x} < \mathtt{y}$	x¡w
$C_{\mathtt{P}}()$	true	x¿0

Soundness of Abstract Execution Rules

Definition (Legal Instantiations of Sequents)
A **sequent** is a **legal instantiation** if it results from substituting all updates $\mathcal{U}_\mathrm{P}$, path conditions C_P and APS symbols with legal instantiations.

It is **valid** iff **all its legal instantiations are valid**.

Soundness of Abstract Execution Rules

Definition (Legal Instantiations of Sequents)
A **sequent** is a **legal instantiation** if it results from substituting all updates $\mathcal{U}_{\mathrm{P}}$, path conditions C_{P} and APS symbols with legal instantiations.

It is **valid** iff **all its legal instantiations are valid**.

Definition (Standard Sequent Calculus Rule Validity)
A sequent calculus rule is **valid** if the validity of the **conclusion** is **implied by** the validity of the **premisses**.

The simple Abstract Execution rule...

simpleAERule

$$\frac{\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathtt{P}}(allLocs :\approx allLocs)\}(C_{\mathtt{P}}(allLocs) \rightarrow [\pi\ \omega]\varphi), \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathbf{\mathtt{abstract\_statement}}\ \mathtt{P};\ \omega]\varphi, \Delta}$$

The simple Abstract Execution rule is insufficient...

Too restrictive

Does not allow instantiations with **irregular termination**

simpleAERule

$$\frac{\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathrm{P}}(\mathit{allLocs} :\approx \mathit{allLocs})\}(C_{\mathrm{P}}(\mathit{allLocs}) \rightarrow [\pi\ \omega]\varphi), \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{\textbf{abstract\_statement}}\ \texttt{P;}\ \omega]\varphi, \Delta}$$

The simple Abstract Execution rule is insufficient...

Too restrictive

Does not allow instantiations with **irregular termination**

Too abstract

Abstract updates/path conditions may **read/write from any location**, no "has-to" assignables

simpleAERule

$$\frac{\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathtt{P}}(allLocs :\approx allLocs)\}(C_{\mathtt{P}}(allLocs) \rightarrow [\pi\ \omega]\varphi), \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{abstract\_statement}}\ \mathtt{P};\ \omega]\varphi, \Delta}$$

A More Complex AE Rule

$\vdash$ $[$ `abstract_statement P;` $]\phi$

A More Complex AE Rule

nonLoopNonVoidAERule

$$\frac{}{\vdash \quad [\pi\ \texttt{abstract\_statement P;}\ \omega]\phi}\ (*)$$

nonLoopNonVoidAERule

$\vdash$

$($

$[\pi$

$\omega]\phi)$

$$\frac{}{\vdash \quad [\pi\, \texttt{abstract\_statement P;}\ \omega]\phi}\ (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$\vdash \quad \{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$

$($

$[\pi$

$\omega]\phi)$

$$\frac{}{\vdash \quad [\pi\ \texttt{abstract\_statement P;}\ \omega]\phi}\ (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$\vdash \quad \{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$

$($

$[\pi$

$\omega]\phi)$

$$\frac{}{\vdash \quad [\pi\, \texttt{abstract\_statement P;}\ \omega]\phi} \ (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$$\vdash \quad \{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$$

$$(\quad \mathsf{C}_{\mathsf{P}}(accessibles)$$

$$\rightarrow [\pi$$

$$\frac{\omega]\phi)}{\vdash \quad [\pi\ \texttt{abstract\_statement P;}\ \omega]\phi} \quad (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$$\vdash \quad \{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$$

$$(\quad \mathsf{C}_{\mathsf{P}}(accessibles)$$

$$\to [\pi$$

$$\omega]\phi)$$

$$\frac{\qquad\qquad}{\vdash \quad [\pi\ \texttt{abstract\_statement P;}\ \omega]\phi}\ (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$\vdash \quad \{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$

$(\quad C_{\mathsf{P}}(accessibles)$

$\rightarrow [\pi$

```
if (returns) return result;
```

$\omega]\phi)$

$$\frac{}{\vdash \quad [\pi\ \texttt{abstract\_statement P;}\ \omega]\phi} \ (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$$\vdash \quad \{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$$

$$\left(\quad \mathsf{C}_{\mathsf{P}}(accessibles) \right.$$

$\to [\pi$

```
if (returns) return result;
if (exc != null) throw exc;
```

$\left. \omega]\phi \right)$

$$\frac{}{\vdash \quad [\pi\ \texttt{abstract\_statement P;}\ \omega]\phi} \quad (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$$\vdash \quad \{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$$

$$\Big(\quad \mathsf{C}_{\mathsf{P}}(accessibles)$$

$$\wedge \; (\mathtt{returns} \doteq \mathrm{TRUE} \leftrightarrow returnsSpec)^?$$

$$\wedge \; (\mathtt{exc} \neq \mathtt{null} \leftrightarrow excSpec)^?$$

$$\to [\pi$$

```
if (returns) return result;
if (exc != null) throw exc;
```

$$\omega]\phi\Big)$$

$$\frac{}{\vdash \quad [\pi\ \mathtt{abstract\_statement\ P;}\ \omega]\phi} \; (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$$\frac{\begin{array}{l}\vdash \quad \{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\} \\ \quad (\quad \mathsf{C}_{\mathsf{P}}(accessibles) \\ \quad\ \wedge \neg(\texttt{returns} \wedge \texttt{exc} \neq \texttt{null}) \\ \quad\ \wedge (\texttt{returns} \doteq \mathrm{TRUE} \leftrightarrow returnsSpec)^{?} \\ \quad\ \wedge (\texttt{exc} \neq \texttt{null} \leftrightarrow excSpec)^{?} \\ \quad \rightarrow [\pi \\ \qquad \texttt{if (returns) return result;} \\ \qquad \texttt{if (exc != null) throw exc;} \\ \qquad \omega]\phi)\end{array}}{\vdash \quad [\pi\ \texttt{abstract\_statement P;}\ \omega]\phi} \ (*)$$

nonLoopNonVoidAERule

$$\frac{\begin{array}{l}\vdash \quad \{\mathcal{U}_{\mathsf{P}}(\mathit{assignables} :\approx \mathit{accessibles})\} \\ \quad\quad \{\mathtt{returns} := \mathit{returns}_0 \,||\, \mathtt{result} := \mathit{result}_0 \,||\, \mathtt{exc} := \mathit{exc}_0\} \\ (\quad \mathsf{C}_{\mathsf{P}}(\mathit{accessibles}) \\ \quad \land \neg(\mathtt{returns} \land \mathtt{exc} \neq \mathtt{null}) \\ \quad \land (\mathtt{returns} \doteq \mathrm{TRUE} \leftrightarrow \mathit{returnsSpec})^? \\ \quad \land (\mathtt{exc} \neq \mathtt{null} \leftrightarrow \mathit{excSpec})^? \\ \to [\pi \\ \quad\quad \mathtt{if\ (returns)\ return\ result;} \\ \quad\quad \mathtt{if\ (exc\ !=\ null)\ throw\ exc;} \\ \quad \omega]\phi)\end{array}}{\vdash \quad [\pi\ \mathtt{abstract\_statement\ P;}\ \omega]\phi} \quad (*)$$

A More Complex AE Rule

nonLoopNonVoidAERule

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathsf{P}}(\mathit{assignables} :\approx \mathit{accessibles})\} \\ \qquad \{\mathtt{returns} := \mathit{returns}_0 \,||\, \mathtt{result} := \mathit{result}_0 \,||\, \mathtt{exc} := \mathit{exc}_0\} \\ (\quad \mathsf{C}_{\mathsf{P}}(\mathit{accessibles}) \\ \ \wedge\ \neg(\mathtt{returns} \wedge \mathtt{exc} \neq \mathtt{null}) \\ \ \wedge\ (\mathtt{returns} \doteq \mathrm{TRUE} \leftrightarrow \mathit{returnsSpec})^? \\ \ \wedge\ (\mathtt{exc} \neq \mathtt{null} \leftrightarrow \mathit{excSpec})^? \\ \rightarrow [\pi \\ \qquad \mathtt{if\ (returns)\ return\ result;} \\ \qquad \mathtt{if\ (exc\ !=\ null)\ throw\ exc;} \\ \quad \omega]\phi), \Delta \end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{abstract\_statement\ P;}\ \omega]\phi, \Delta}\ (*)$$

Specification of APSs +

Symbolic Execution of APSs +

Simplification of Abstract State Changes

Three Categories of Abstract Update Simplification Rules

Three Categories of Abstract Update Simplification Rules

1. Removal of **ineffective** (assignables in) updates (1 rule)

Three Categories of Abstract Update Simplification Rules

1. Removal of **ineffective** (assignables in) updates (1 rule)
2. Interplay between **concrete and abstract** updates (2 rules)

Three Categories of Abstract Update Simplification Rules

1. Removal of **ineffective** (assignables in) updates (1 rule)
2. Interplay between **concrete and abstract** updates (2 rules)
3. Abstract update **concatenation** and **permutation** (2 rules)

Case Study: Correctness of Refactoring Rules

Consolidate Duplicate Conditional Fragments

The same fragment of code is in all branches of a conditional expression.

Move it outside of the expression.

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

⇓

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```


Motivation

Sometimes you find the same code executed in all legs of a conditional. In that case you should move the code to outside the conditional. This makes clearer what varies and what stays the same.

Mechanics

- Identify code that is executed the same way regardless of the condition.
- If the common code is at the beginning, move it to before the conditional.
- If the common code is at the end, move it to after the conditional.

Consolidate Duplicate Conditional Fragments

The same fragment of code is in all branches of a conditional expression.

Move it outside of the expression.

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

⇓

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

UNSOUND

Motivation

Sometimes you find the same code executed in all legs of a conditional. In that case you should move the code to outside the conditional. This makes clearer what varies and what stays the same.

Mechanics

- Identify code that is executed the same way regardless of the condition.
- If the common code is at the beginning, move it to before the conditional.
- If the common code is at the end, move it to after the conditional.

Analyzing and Proving Refactoring Techniques with Abstract Execution: Methodology

Analyzing and **Proving Refactoring Techniques** with Abstract Execution: Methodology

1. Create **refactoring model:** Two **abstract programs** (before / after refactoring) with minimal specification

Analyzing and Proving Refactoring Techniques with Abstract Execution: Methodology

1. Create **refactoring model:** Two **abstract programs** (before / after refactoring) with minimal specification
2. Load **proof obligation** (“before refactoring $\leftrightarrow$ after refactoring”) into **KeY**

Analyzing and Proving Refactoring Techniques with Abstract Execution: Methodology

1. Create **refactoring model:** Two **abstract programs** (before / after refactoring) with minimal specification
2. Load **proof obligation** (“before refactoring $\leftrightarrow$ after refactoring”) into **KeY**
3. Start **automatic** proof

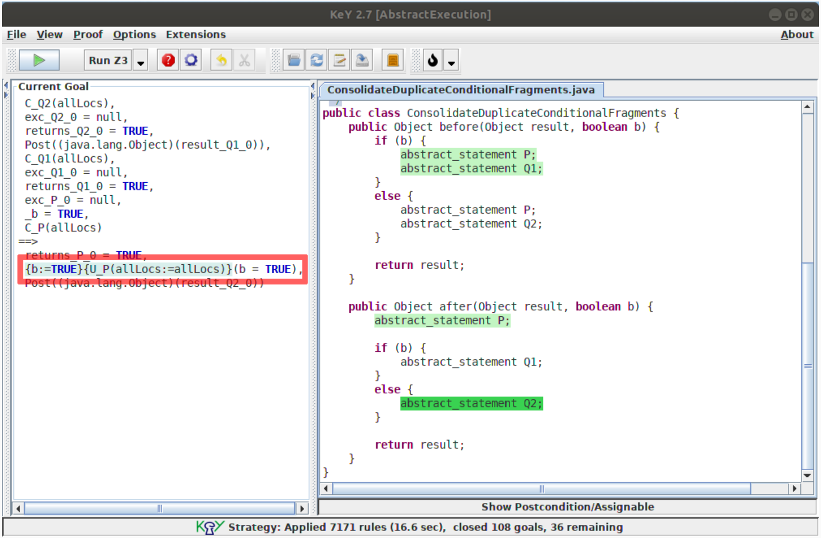
Analyzing and Proving Refactoring Techniques with Abstract Execution: Methodology

1. Create **refactoring model:** Two **abstract programs** (before / after refactoring) with minimal specification
2. Load **proof obligation** ("before refactoring $\leftrightarrow$ after refactoring") into **KeY**
3. Start **automatic** proof
   - Proof closed $\Longrightarrow$ Modeled **refactoring correct**

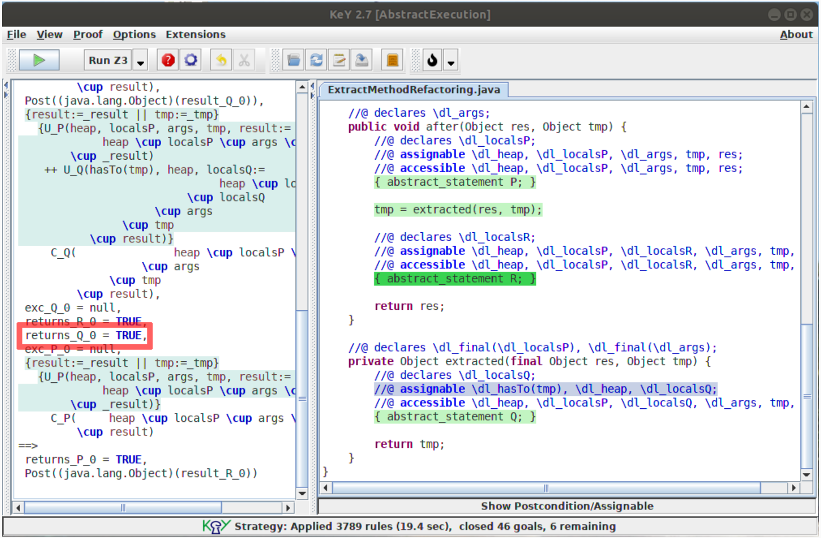
Analyzing and Proving Refactoring Techniques with Abstract Execution: Methodology

1. Create **refactoring model:** Two **abstract programs** (before / after refactoring) with minimal specification
2. Load **proof obligation** ("before refactoring $\leftrightarrow$ after refactoring") into **KeY**
3. Start **automatic** proof
   - Proof closed $\Longrightarrow$ Modeled **refactoring correct**
   - Open goals $\Longrightarrow$ Inspect proof, maybe **adapt model**

Proof Inspection: Imprecise I/O Specifications



Proof Inspection: Missing Irregular Termination Specifications



Proving Refactoring Techniques: Results

Proving Refactoring Techniques: Results

- Proved correctness of **models for 8 refactorings**:
  (1) Consolidate Duplicate Conditional Fragments (four variants), (2) Decompose Conditional, (3) Extract Method, (4) Replace Exception with Test, (5) Move Statements to Callers, (6) Slide Statements, (7) Split Loop, (8) Remove Control Flag

Proving Refactoring Techniques: Results

- Proved correctness of **models for 8 refactorings**:

  (1) Consolidate Duplicate Conditional Fragments (four variants), (2) Decompose Conditional, (3) Extract Method, (4) Replace Exception with Test, (5) Move Statements to Callers, (6) Slide Statements, (7) Split Loop, (8) Remove Control Flag

- Elicitation of **non-trivial behavioral restrictions** not mentioned in literature for 10 out of 11 studied models

Proving Refactoring Techniques: Results

- Proved correctness of **models for 8 refactorings**:
  (1) Consolidate Duplicate Conditional Fragments (four variants), (2) Decompose Conditional, (3) Extract Method, (4) Replace Exception with Test, (5) Move Statements to Callers, (6) Slide Statements, (7) Split Loop, (8) Remove Control Flag
- Elicitation of **non-trivial behavioral restrictions** not mentioned in literature for 10 out of 11 studied models
- **Automatic proofs** for loop-free problems,
  small **proof scripts** for problems with loops (coupling)

Example: Replace Exception with Test
Don't use exceptions...

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Don't use exceptions...

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Don't use exceptions...

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Don't use exceptions...

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

...as a substitute for conditional tests.

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

...as a substitute for conditional tests.

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

…as a substitute for conditional tests.

```java
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```java
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

...as a substitute for conditional tests.

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Example run for $y = 0$

```java
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```java
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test
Example run for $y = 0$

```java
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```java
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Example run for $y = 0$

```java
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```java
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Example run for $y = 0$

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Example run for $y = 0$

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Example run for $y = 0$

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

Example run for $y = 0$

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
// z == 42
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
// z == 0
```

Example: Replace Exception with Test
Lets “fix” the refactoring!

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
// z == 42
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
// z == 0
```

UNSOUND

Example: Replace Exception with Test

"Roll back" to a common program state.

```java
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
// z == 42
```

```java
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
// z == 0
```

Example: Replace Exception with Test

"Roll back" to a common program state.

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    z = 0; x = 0;
    x = Integer.MAX_VALUE;
}
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    z = 0; x = 0;
    x = Integer.MAX_VALUE;
}
```

Example: Replace Exception with Test

"Roll back" to a common program state.

```
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    z = 0; x = 0;
    x = Integer.MAX_VALUE;
}
```

```
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    z = 0; x = 0;
    x = Integer.MAX_VALUE;
}
```

Future Work & Conclusion

Future and Work

Future and Work

- Increase **support** for **heap-related properties** (*ongoing*)

Future and Work

- Increase **support** for **heap-related properties** (*ongoing*)
- Better **automation** for **problems with loops**

Future and Work

- Increase **support** for **heap-related properties** (*ongoing*)
- Better **automation** for **problems with loops**
- Apply to **structurally different** (e.g., iterative vs. recursive) & **concurrent** programs

Future and Work

- Increase **support** for **heap-related properties** (*ongoing*)
- Better **automation** for **problems with loops**
- Apply to **structurally different** (e.g., iterative vs. recursive) & **concurrent** programs
- Apply to different **target areas**:

Future and Work

- Increase **support** for **heap-related properties** (*ongoing*)
- Better **automation** for **problems with loops**
- Apply to **structurally different** (e.g., iterative vs. recursive) & **concurrent** programs
- Apply to different **target areas**:
  - **Correctness-by-construction** (*cooperation ongoing*)

Future and Work

- Increase **support** for **heap-related properties** (*ongoing*)
- Better **automation** for **problems with loops**
- Apply to **structurally different** (e.g., iterative vs. recursive) & **concurrent** programs
- Apply to different **target areas**:
  - **Correctness-by-construction** (*cooperation ongoing*)
  - **Compilation** (*formal foundations already established*)

Future and Work

- Increase **support** for **heap-related properties** (*ongoing*)
- Better **automation** for **problems with loops**
- Apply to **structurally different** (e.g., iterative vs. recursive) & **concurrent** programs
- Apply to different **target areas**:
  - **Correctness-by-construction** (*cooperation ongoing*)
  - **Compilation** (*formal foundations already established*)
  - **Optimization / Parallelization** (*cooperation started*)

- **Abstract Execution**: `abstract_program P;`
  Automatic proofs of **abstract** programs

- **Abstract Execution**: `abstract_program P;`
  Automatic proofs of **abstract** programs

- **Precise specification** of input/output `//@ assignable x;`
  and irregular termination behavior

- **Abstract Execution**: `abstract_program P;`
  Automatic proofs of **abstract** programs
- **Precise specification** of input/output `//@ assignable x;`
  and irregular termination behavior
- Core idea: **2nd-order Skolemization** $\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx \mathtt{y}, \mathtt{z})$

- **Abstract Execution**: `abstract_program P;`
  Automatic proofs of **abstract** programs
- **Precise specification** of input/output `//@ assignable x;`
  and irregular termination behavior
- Core idea: **2nd-order Skolemization** $\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx \mathtt{y}, \mathtt{z})$
- **Implemented** for the KeY framework

- **Abstract Execution**: `abstract_program P;`
  Automatic proofs of **abstract** programs
- **Precise specification** of input/output and irregular termination behavior `//@ assignable x;`
- Core idea: **2nd-order Skolemization** $\mathcal{U}_{\mathrm{P}}(\mathrm{x} :\approx \mathrm{y}, \mathrm{z})$
- **Implemented** for the KeY framework
- Case Study: Correctness of **Java refactoring techniques** ✓

References

- Anna Maria Eilertsen, Anya Helene Bagge, and Volker Stolz, **Safer Refactorings**, Proc. 7th Intern. Symp. on Leveraging Applications of Formal Methods, ISoLA, 2016, pp. 517–531.
- Martin Fowler, **Refactoring: Improving the Design of Existing Code**, Object Technology Series, Addison-Wesley, June 1999.
- Xavier Leroy, **Formal Verification of a Realistic Compiler**, Communications of the ACM **52** (2009), no. 7, 107–115.
- Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish, **A New Verified Compiler Backend for CakeML**, Proc. 21st Intern. Conf. on Functional Programming, ACM, 2016, pp. 60–73.

Properties of Concrete Programs: Relational Verification

```
//@ requires a != 0 && b != 0;
public int abs1(int a, int b) {
  if (a < b) {
    int tmp = a;
    a = b;
    b = tmp;
  }

  return a - b;
}
```

Properties of Concrete Programs: Relational Verification

```
//@ requires a != 0 && b != 0;
public int abs1(int a, int b) {
  if (a < b) {
    int tmp = a;
    a = b;
    b = tmp;
  }

  return a - b;
}
```

```
//@ requires a != 0 && b != 0;
public int abs2(int a, int b) {
  if (a < b) {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
  }

  return a - b;
}
```

Properties of Concrete Programs: Relational Verification

```
//@ requires a != 0 && b != 0;
public int abs1(int a, int b) {
  if (a < b) {
    int tmp = a;
    a = b;
    b = tmp;
  }

  return a - b;
}
```

```
//@ requires a != 0 && b != 0;
public int abs2(int a, int b) {
  if (a < b) {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
  }

  return a - b;
}
```

Properties of Concrete Programs: Information Flow Security

```
// low: OK, userInput | high: pin
public void checkPIN(int userInput) {
  if (pin == userInput) {
    OK = true;
  } else {
    OK = false;
  }
}
```

Properties of Many Programs: Compilation Rules

```
if (b
   )
  P₁
else
  P₂
```

Properties of Many Programs: Compilation Rules

```
if (b
    )
  P1
else
  P2
```

$\xrightarrow[\text{to}]{\text{compiles}}$

Properties of Many Programs: Compilation Rules

```
if (b
   )
  P1
else
  P2
```

$\xrightarrow[\text{to}]{\text{compiles}}$

```
%1 = load i1, i1* %b
br i1 %1, label %2, label
    %3
P1   ; <label>:%2
br label %4
P2   ; <label>:%3
br label %4
     ; <label>:%4
```

Properties of Many Programs: Compilation Rules

```
if (b
    )
  P1
else
  P2
```

$\xrightarrow[\text{to}]{\text{compiles}}$

```
%1 = load i1, i1* %b
br i1 %1, label %2, label
    %3
P1   ; <label>:%2
br label %4
P2   ; <label>:%3
br label %4
     ; <label>:%4
```

Properties of Many Programs: Correctness-by-Construction (CbC)

```
//@ ensures x >= 0;
{
  P
}
```

Properties of Many Programs: Correctness-by-Construction (CbC)

```
//@ ensures x >= 0;
{
  P
}
```

$\xrightarrow[\text{to}]{\text{refines}}$

Properties of Many Programs: Correctness-by-Construction (CbC)

```
//@ ensures x >= 0;
{
  P
}
```

$\xrightarrow[\text{to}]{\text{refines}}$

```
//@ ensures x >= 0;
{
  if (x < 0)
    P1
  else
    P2
}
```

Properties of Many Programs: General Security Properties

```
// low: OK, userInput | high: pin
public void checkPIN(int userInput) {
  P

  OK = false;
  userInput = null;
}
```

A Complex AE Rule in a Loop Context

```
[
abstract_statement P;
]ϕ
```

A Complex AE Rule in a Loop Context

[

$\circlearrowleft_{\mathsf{x}}$ `abstract_statement P;` $Rest_1$ $_{\mathsf{x}}\circlearrowleft$

$]\phi$

A Complex AE Rule in a Loop Context

$[\quad l_1 : \{ \cdots \{ l_n : \{$

$\circlearrowright_{\mathsf{x}}$ `abstract_statement P;` $Rest_1 \ {}_{\mathsf{x}}\!\circlearrowleft Rest_2$

$\}\} \cdots \} \quad]\phi$

A Complex AE Rule in a Loop Context

$$\frac{[\ \ l_1 : \{ \cdots \{\ \ l_n : \{\ \ \circlearrowright_{\mathsf{x}} \quad Rest_1\ {}_{\mathsf{x}}\circlearrowright \quad Rest_2\ \}\} \cdots \}\ \ ]\phi}{[\ \ l_1 : \{ \cdots \{ l_n : \{\ \ \circlearrowright_{\mathsf{x}}\ \texttt{abstract\_statement P;}\ Rest_1\ {}_{\mathsf{x}}\circlearrowright\ Rest_2\ \}\} \cdots \}\ \ ]\phi}$$

A Complex AE Rule in a Loop Context

$\{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$

$[\quad l_1 : \{ \cdots \{ \;\; l_n : \{$

$\circlearrowleft_{\mathsf{x}}$

$Rest_1 \;\; {}_{\mathsf{x}}\circlearrowleft$

$Rest_2 \; \}\} \cdots \} \quad]\phi$

$[\quad l_1 : \{ \cdots \{ l_n : \{$

$\circlearrowleft_{\mathsf{x}}$ `abstract_statement P;` $Rest_1 \;\; {}_{\mathsf{x}}\circlearrowleft \; Rest_2$

$\}\} \cdots \} \quad]\phi$

A Complex AE Rule in a Loop Context

$\{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$

$($ $\mathsf{C}_{\mathsf{P}}(accessibles)$

$\rightarrow [$ $l_1 : \{ \cdots \{ \; l_n : \{$
$\circlearrowright_{\mathsf{x}}$

$Rest_1 \; {}_{\mathsf{x}}\circlearrowleft$
$Rest_2 \; \}\} \cdots \}$ $]\phi)$

$[$ $l_1 : \{ \cdots \{ l_n : \{$
$\circlearrowright_{\mathsf{x}}$ `abstract_statement P;` $Rest_1 \; {}_{\mathsf{x}}\circlearrowleft \; Rest_2$
$\}\} \cdots \}$ $]\phi$

A Complex AE Rule in a Loop Context

$\{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$

$($ $\mathsf{C}_{\mathsf{P}}(accessibles)$

$\rightarrow [$ $l_1 : \{ \cdots \{$ $l_n : \{$

```
⟲x if (returns) return result; if (exc != null) throw exc;
   if (breaks) break;           if (continues) continue;
   if (breaksToLbl_1) break l1; ··· if (breaksToLbl_n) break ln;
   Rest1 x⟲
Rest2 }}···}   ]ϕ)
```

$[$ $l_1 : \{ \cdots \{ l_n : \{$

```
⟲x abstract_statement P; Rest1 x⟲ Rest2
}}···}   ]ϕ
```

A Complex AE Rule in a Loop Context

$$\{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\}$$

$$
\begin{array}{l}
(\quad C_{\mathsf{P}}(accessibles) \\
\quad \wedge\ (\texttt{returns} = \mathrm{TRUE} \leftrightarrow returnsSpec)^{?} \ \wedge\ (\texttt{exc} \neq \texttt{null} \leftrightarrow excSpec)^{?} \\
\quad \wedge\ (\texttt{breaks} \doteq \mathrm{TRUE} \leftrightarrow breaksSpec)^{?} \\
\quad \wedge\ (\texttt{continues} \doteq \mathrm{TRUE} \leftrightarrow continuesSpec)^{?} \\
\quad \wedge\ (\texttt{breaksToLbl\_1} \doteq \mathrm{TRUE} \leftrightarrow breaksLbl1Spec)^{?} \wedge \cdots \\
\quad \wedge\ (\texttt{breaksToLbl\_n} \doteq \mathrm{TRUE} \leftrightarrow breaksLblnSpec)^{?} \\
\rightarrow [\quad l_1 : \{ \cdots \{\ l_n : \{ \\
\qquad \circlearrowleft_{\mathsf{x}} \texttt{if (returns) return result; if (exc != null) throw exc;} \\
\qquad\quad \texttt{if (breaks) break; if (continues) continue;} \\
\qquad\quad \texttt{if (breaksToLbl\_1) break } l_1\texttt{; } \cdots \texttt{ if (breaksToLbl\_n) break } l_n\texttt{;} \\
\qquad\quad Rest_1 \ {}_{\mathsf{x}}\circlearrowleft \\
\quad Rest_2 \ \}\} \cdots \} \quad]\phi)
\end{array}
$$

$$
\begin{array}{c}
[\quad l_1 : \{ \cdots \{ l_n : \{ \\
\circlearrowleft_{\mathsf{x}} \texttt{abstract\_statement P;}\ Rest_1 \ {}_{\mathsf{x}}\circlearrowleft \ Rest_2 \\
\}\} \cdots \} \quad]\phi
\end{array}
$$

A Complex AE Rule in a Loop Context

$$\{\mathcal{U}_{\mathsf{P}}(\mathit{assignables} :\approx \mathit{accessibles})\}$$

$$
\begin{array}{l}
(\quad C_{\mathsf{P}}(\mathit{accessibles}) \\
\;\wedge\; \mathit{mutex}\,(\texttt{returns}, \texttt{exc} \neq \texttt{null}, \texttt{breaksToLbl\_1}, \cdots, \texttt{breaksToLbl\_n}) \\
\;\wedge\; (\texttt{returns} \doteq \mathrm{TRUE} \leftrightarrow \mathit{returnsSpec})^? \;\wedge\; (\texttt{exc} \neq \texttt{null} \leftrightarrow \mathit{excSpec})^? \\
\;\wedge\; (\texttt{breaks} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksSpec})^? \\
\;\wedge\; (\texttt{continues} \doteq \mathrm{TRUE} \leftrightarrow \mathit{continuesSpec})^? \\
\;\wedge\; (\texttt{breaksToLbl\_1} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksLbl1Spec})^? \wedge \cdots \\
\;\wedge\; (\texttt{breaksToLbl\_n} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksLblnSpec})^? \\
\rightarrow [\quad l_1 : \{ \cdots \{\; l_n : \{ \\
\qquad ⟲_{\mathsf{x}}\; \texttt{\textbf{if} (returns) \textbf{return} result; \textbf{if} (exc != \textbf{null}) \textbf{throw} exc;} \\
\qquad\quad \texttt{\textbf{if} (breaks) \textbf{break}; \textbf{if} (continues) \textbf{continue};} \\
\qquad\quad \texttt{\textbf{if} (breaksToLbl\_1) \textbf{break}}\; l_1\texttt{;} \cdots \texttt{\textbf{if} (breaksToLbl\_n) \textbf{break}}\; l_n\texttt{;} \\
\qquad\quad \mathit{Rest}_1 \;{}_{\mathsf{x}}⟲ \\
\quad \mathit{Rest}_2 \;\}\}\cdots\} \quad]\phi)
\end{array}
$$

$$
\begin{array}{c}
[\quad l_1 : \{ \cdots \{ l_n : \{ \\
⟲_{\mathsf{x}}\; \texttt{\textbf{abstract\_statement} P;}\; \mathit{Rest}_1 \;{}_{\mathsf{x}}⟲\; \mathit{Rest}_2 \\
\}\}\cdots\} \quad]\phi
\end{array}
$$

A Complex AE Rule in a Loop Context

$$
\frac{
\begin{array}{l}
\quad\quad \{\mathcal{U}_{\mathsf{P}}(\mathit{assignables} :\approx \mathit{accessibles})\} \\
\quad\quad \{\texttt{returns} := \mathit{returns}_0 \,||\, \texttt{result} := \mathit{result}_0 \,||\, \texttt{exc} := \mathit{exc}_0 \,|| \\
\quad\quad\; \texttt{breaks} := \mathit{breaks}_0 \,||\, \texttt{continues} := \mathit{continues}_0 \,|| \\
\quad\quad\; \texttt{breaksToLbl\_1} := \mathit{breaksToLabel1}_0 \,||\, \cdots \,|| \\
\quad\quad\; \texttt{breaksToLbl\_n} := \mathit{breaksToLabeln}_0\} \\
(\quad C_{\mathsf{P}}(\mathit{accessibles}) \\
\;\wedge\; \mathit{mutex}\,(\texttt{returns}, \texttt{exc} \neq \texttt{null}, \texttt{breaksToLbl\_1}, \cdots, \texttt{breaksToLbl\_n}) \\
\;\wedge\; (\texttt{returns} \doteq \mathrm{TRUE} \leftrightarrow \mathit{returnsSpec})^{?} \;\wedge\; (\texttt{exc} \neq \texttt{null} \leftrightarrow \mathit{excSpec})^{?} \\
\;\wedge\; (\texttt{breaks} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksSpec})^{?} \\
\;\wedge\; (\texttt{continues} \doteq \mathrm{TRUE} \leftrightarrow \mathit{continuesSpec})^{?} \\
\;\wedge\; (\texttt{breaksToLbl\_1} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksLbl1Spec})^{?} \wedge \cdots \\
\;\wedge\; (\texttt{breaksToLbl\_n} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksLblnSpec})^{?} \\
\to [\quad l_1 : \{ \cdots \{ \; l_n : \{ \\
\quad\quad \circlearrowright_{x} \; \textbf{if } (\texttt{returns}) \textbf{ return } \texttt{result}; \; \textbf{if } (\texttt{exc != } \textbf{null}) \textbf{ throw } \texttt{exc}; \\
\quad\quad\quad \textbf{if } (\texttt{breaks}) \textbf{ break}; \quad \textbf{if } (\texttt{continues}) \textbf{ continue}; \\
\quad\quad\quad \textbf{if } (\texttt{breaksToLbl\_1}) \textbf{ break } l_1; \; \cdots \; \textbf{if } (\texttt{breaksToLbl\_n}) \textbf{ break } l_n; \\
\quad\quad\quad \mathit{Rest}_1 \;{}_{x}\!\circlearrowleft \\
\quad \mathit{Rest}_2 \; \}\} \cdots \} \quad]\phi)
\end{array}
}{
\begin{array}{c}
[\quad l_1 : \{ \cdots \{ l_n : \{ \\
\circlearrowright_{x} \; \texttt{abstract\_statement P}; \; \mathit{Rest}_1 \;{}_{x}\!\circlearrowleft \; \mathit{Rest}_2 \\
\}\} \cdots \} \quad]\phi
\end{array}
}
$$

A Complex AE Rule in a Loop Context

nonVoidLoopAERule

$$
\begin{array}{l}
\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)\} \\
\quad \{\texttt{returns} := returns_0 \,||\, \texttt{result} := result_0 \,||\, \texttt{exc} := exc_0 \,|| \\
\quad\; \texttt{breaks} := breaks_0 \,||\, \texttt{continues} := continues_0 \,|| \\
\quad\; \texttt{breaksToLbl\_1} := breaksToLabel1_0 \,||\, \cdots \,|| \\
\quad\; \texttt{breaksToLbl\_n} := breaksToLabeln_0\} \\
(\quad C_{\mathsf{P}}(accessibles) \\
\;\wedge\; mutex\,(\texttt{returns}, \texttt{exc} \neq \texttt{null}, \texttt{breaksToLbl\_1}, \cdots, \texttt{breaksToLbl\_n}) \\
\;\wedge\; (\texttt{returns} \doteq \mathrm{TRUE} \leftrightarrow returnsSpec)^? \;\wedge\; (\texttt{exc} \neq \texttt{null} \leftrightarrow excSpec)^? \\
\;\wedge\; (\texttt{breaks} \doteq \mathrm{TRUE} \leftrightarrow breaksSpec)^? \\
\;\wedge\; (\texttt{continues} \doteq \mathrm{TRUE} \leftrightarrow continuesSpec)^? \\
\;\wedge\; (\texttt{breaksToLbl\_1} \doteq \mathrm{TRUE} \leftrightarrow breaksLbl1Spec)^? \wedge \cdots \\
\;\wedge\; (\texttt{breaksToLbl\_n} \doteq \mathrm{TRUE} \leftrightarrow breaksLblnSpec)^? \\
\rightarrow [\pi \quad l_1 : \{ \cdots \{ \; l_n : \{ \\
\quad \circlearrowleft_{\mathsf{x}} \texttt{if (returns) }\textbf{return}\texttt{ result; if (exc != }\textbf{null}\texttt{) }\textbf{throw}\texttt{ exc;} \\
\quad\quad \texttt{if (breaks) }\textbf{break}\texttt{;} \qquad \texttt{if (continues) }\textbf{continue}\texttt{;} \\
\quad\quad \texttt{if (breaksToLbl\_1) }\textbf{break}\; l_1\texttt{;} \;\cdots\; \texttt{if (breaksToLbl\_n) }\textbf{break}\; l_n\texttt{;} \\
\quad\quad Rest_1 \;{}_{\mathsf{x}}\!\circlearrowleft \\
\quad Rest_2 \;\}\}\cdots\}\;\omega]\phi), \Delta \\
\hline
\qquad \Gamma \vdash \{\mathcal{U}\}[\pi \; l_1 : \{ \cdots \{ l_n : \{ \\
\qquad\quad \circlearrowleft_{\mathsf{x}} \textbf{abstract\_statement}\;\texttt{P;}\; Rest_1 \;{}_{\mathsf{x}}\!\circlearrowleft\; Rest_2 \\
\qquad \}\}\cdots\}\;\omega]\phi, \Delta
\end{array}
$$

Handling Programs with Loops:

$$\{\mathcal{U}\}[\texttt{while(}\, expr\, \texttt{)}\ body]\ \varphi$$

Handling Programs with Loops: Use Loop Invariant Reasoning

loopInvariantAE

$$\frac{}{\vdash \{\mathcal{U}\}[\texttt{while(}\, expr\, \texttt{)}\ body]\ \varphi}$$

Handling Programs with Loops:
Prove that the invariant holds in the initial state...

loopInvariantAE

$$\vdash \{\mathcal{U}\}Inv \qquad \text{(initially valid)}$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$\vdash \{\mathcal{U}\}[\mathtt{while(}\ expr\ \mathtt{)}\ body]\ \varphi$$

Handling Programs with Loops:
Prove that the invariant holds in the initial state...

loopInvariantAE

$$\vdash \{\mathcal{U}\}Inv \qquad \text{(initially valid)}$$

$$\overline{\vdash \{\mathcal{U}\}[\texttt{while(}\, expr\, \texttt{)}\ body]\ \varphi}$$

Handling Programs with Loops:
Prove that the invariant holds in the initial state...

loopInvariantAE

$$\vdash \{\mathcal{U}\} Inv \qquad \text{(initially valid)}$$

$$\overline{\vdash \{\mathcal{U}\}[\texttt{while(}\ expr\ \texttt{)}\ body]\ \varphi}$$

Handling Programs with Loops:
...and is inductive and strong enough for the post condition

loopInvariantAE

$\vdash \{\mathcal{U}\}Inv$ (initially valid)

$\vdash$ (preserved & use case)

$$\frac{}{\vdash \{\mathcal{U}\}[\texttt{while(}\ expr\ \texttt{)}\ body]\ \varphi}$$

Handling Programs with Loops:
Reason about an arbitrary iteration by anonymization

loopInvariantAE

$\vdash \{\mathcal{U}\} Inv$ (initially valid)

$\vdash \{\mathcal{U}'\} \Big($ (preserved & use case)

$\Big)$

$$\overline{\vdash \{\mathcal{U}\}[\texttt{while(}\,expr\,\texttt{)}\ body]\ \varphi}$$

Handling Programs with Loops: Assume the invariant holds before an arbitrary run...

loopInvariantAE

$\vdash \{\mathcal{U}\} Inv$ (initially valid)

$\vdash \{\mathcal{U}'\} \Big(Inv \rightarrow \qquad \Big)$ (preserved & use case)

$\vdash \{\mathcal{U}\}[\mathtt{while(}\ expr\ \mathtt{)}\ body]\ \varphi$

Handling Programs with Loops:
...and that it is preserved by that run

loopInvariantAE

$\vdash \{\mathcal{U}\}Inv$ (initially valid)

$\vdash \{\mathcal{U}'\}\Big(Inv \rightarrow [\ldots]$ (preserved & use case)

$($

$)\Big)$

$$\vdash \{\mathcal{U}\}[\texttt{while(}\,expr\,\texttt{)}\ body]\ \varphi$$

Handling Programs with Loops:
...and that it is preserved by that run

loopInvariantAE

$$\vdash \{\mathcal{U}\} Inv \qquad \text{(initially valid)}$$
$$\vdash \{\mathcal{U}'\} \Big(Inv \rightarrow [\ldots] \qquad \text{(preserved \& use case)}$$
$$($$
$$(loopContinues \rightarrow (Inv \qquad)))\Big)$$

$$\vdash \{\mathcal{U}\}[\texttt{while(}\, expr \,\texttt{)}\ body]\ \varphi$$

Handling Programs with Loops:
...and that it is preserved by that run

loopInvariantAE

$$\vdash \{\mathcal{U}\} Inv \qquad \text{(initially valid)}$$
$$\vdash \{\mathcal{U}'\} \Big(Inv \rightarrow [\ldots] \qquad \text{(preserved \& use case)}$$
$$($$
$$(loopContinues \rightarrow (Inv \qquad))) \Big)$$

$$\vdash \{\mathcal{U}\}[\mathtt{while(}\ expr\ \mathtt{)}\ body]\ \varphi$$

Handling Programs with Loops:
...and that it is preserved by that run

loopInvariantAE

$$\vdash \{\mathcal{U}\}Inv \qquad \text{(initially valid)}$$

$$\vdash \{\mathcal{U}'\}\Big(Inv \rightarrow [\ldots] \qquad \text{(preserved \& use case)}$$

$$($$

$$(loopContinues \rightarrow (Inv \qquad)))\Big)$$

$$\vdash \{\mathcal{U}\}[\texttt{while(}\, expr \,\texttt{)}\ body]\ \varphi$$

Handling Programs with Loops: Use the invariant when proving the post condition ("use case")

loopInvariantAE

$$\frac{\begin{array}{ll} \vdash \{\mathcal{U}\}Inv & \text{(initially valid)} \\ \vdash \{\mathcal{U}'\}\Big(Inv \rightarrow [\ldots] & \text{(preserved \& use case)} \\ \quad ((loopExited \quad \rightarrow \varphi \qquad) \wedge & \\ \quad (loopContinues \rightarrow (Inv \qquad)))\Big) & \end{array}}{\vdash \{\mathcal{U}\}[\texttt{while(}\ expr\ \texttt{)}\ body]\ \varphi}$$

Handling Programs with Loops: Post records results, separates continuing & breaking runs

loopInvariantAE

$$\dfrac{\begin{array}{ll} \vdash \{\mathcal{U}\} Inv & \text{(initially valid)} \\ \vdash \{\mathcal{U}'\} \Big(Inv \rightarrow [\ldots] & \text{(preserved \& use case)} \\ \quad \big((loopExited \quad \rightarrow \varphi[Post(\texttt{result}, \mathrm{TRUE})]) \wedge & \\ \quad (loopContinues \rightarrow (Inv &))) \Big) \end{array}}{\vdash \{\mathcal{U}\}[\texttt{while(} \; expr \; \texttt{)} \; body](\varphi[Post(\texttt{result}, \mathrm{TRUE})])}$$

Handling Programs with Loops: Post records results, separates continuing & breaking runs

loopInvariantAE

$$\vdash \{\mathcal{U}\}Inv \qquad \text{(initially valid)}$$

$$\vdash \{\mathcal{U}'\}\Big(Inv \rightarrow [\ldots] \qquad \text{(preserved \& use case)}$$

$$\big((loopExited \rightarrow \varphi[Post(\mathtt{result}, \mathrm{TRUE})]) \wedge$$

$$(loopContinues \rightarrow (Inv)))\Big)$$

$$\overline{\vdash \{\mathcal{U}\}[\mathtt{while(}\ expr\ \mathtt{)}\ body](\varphi[Post(\mathtt{result}, \mathrm{TRUE})])}$$

Handling Programs with Loops:
Post records results, separates continuing & breaking runs

loopInvariantAE

$$\vdash \{\mathcal{U}\}Inv \qquad \text{(initially valid)}$$

$$\vdash \{\mathcal{U}'\}\Big(Inv \rightarrow [\ldots] \qquad \text{(preserved \& use case)}$$

$$\big((loopExited \quad \rightarrow \varphi[Post(\texttt{result}, \mathrm{TRUE})]) \wedge$$

$$(loopContinues \rightarrow (Inv \qquad)))\Big)$$

$$\overline{\vdash \{\mathcal{U}\}[\texttt{while(}\ expr\ \texttt{)}\ body](\varphi[Post(\texttt{result}, \mathrm{TRUE})])}$$

Handling Programs with Loops: Post records results, separates continuing & breaking runs

loopInvariantAE

$$
\frac{
\begin{array}{ll}
\vdash \{\mathcal{U}\}Inv & \text{(initially valid)} \\
\vdash \{\mathcal{U}'\}\Big(Inv \rightarrow [\ldots] & \text{(preserved \& use case)} \\
\quad \Big((loopExited \quad \rightarrow \varphi[Post(\mathtt{result}, \mathrm{TRUE})]) \wedge & \\
\quad (loopContinues \rightarrow (Inv \quad)))\Big) &
\end{array}
}{
\vdash \{\mathcal{U}\}[\mathtt{while(}\ expr\ \mathtt{)}\ body](\varphi[Post(\mathtt{result}, \mathrm{TRUE})])
}
$$

Handling Programs with Loops:
Post records results, separates continuing & breaking runs

loopInvariantAE

$$\frac{\begin{array}{lr}\vdash \{\mathcal{U}\}Inv & \text{(initially valid)}\\ \vdash \{\mathcal{U}'\}\Big(Inv \rightarrow [\ldots] & \text{(preserved \& use case)}\\ \quad\Big((loopExited \quad\quad \rightarrow \varphi[Post(\texttt{result}, \mathrm{TRUE})]) \land & \\ \quad\;(loopContinues \rightarrow (Inv \land \varphi[Post(\texttt{result}, \mathrm{FALSE})]))\Big) & \end{array}}{\vdash \{\mathcal{U}\}[\texttt{while(}\,expr\,\texttt{)}\ body](\varphi[Post(\texttt{result}, \mathrm{TRUE})])}$$

Handling Programs with Loops: Post records results, separates continuing & breaking runs

loopInvariantAE

$$\vdash \{\mathcal{U}\}Inv \qquad \text{(initially valid)}$$

$$\vdash \{\mathcal{U}'\}\Big(Inv \rightarrow [\ldots] \qquad \text{(preserved \& use case)}$$

$$\Big((loopExited \rightarrow \varphi[Post(\texttt{result}, \mathrm{TRUE})]) \wedge$$

$$(loopContinues \rightarrow (Inv \wedge \varphi[Post(\texttt{result}, \mathrm{FALSE})]))\Big)\Big)$$

$$\overline{\vdash \{\mathcal{U}\}[\texttt{while(}\ expr\ \texttt{)}\ body](\varphi[Post(\texttt{result}, \mathrm{TRUE})])}$$

Handling Programs with Loops:
+ Scripted Loop Coupling, Iteration Structure Harmonization

loopInvariantAE

$$\begin{array}{ll}
\vdash \{\mathcal{U}\} Inv & \text{(initially valid)} \\
\vdash \{\mathcal{U}'\} \Big(Inv \rightarrow [\ldots] & \text{(preserved \& use case)} \\
\quad \Big((loopExited \quad \rightarrow \varphi[Post(\texttt{result}, \mathrm{TRUE})]) \wedge & \\
\quad\; (loopContinues \rightarrow (Inv \wedge \varphi[Post(\texttt{result}, \mathrm{FALSE})])) \Big) \Big) & \\
\hline
\vdash \{\mathcal{U}\}[\texttt{while(}\, expr \,\texttt{)}\ body](\varphi[Post(\texttt{result}, \mathrm{TRUE})]) &
\end{array}$$

Update Simplification in JavaDL

$$\{\mathtt{x} := \mathtt{y}\}[\mathtt{z=x;}](\mathtt{z} \doteq \mathtt{y})$$

Update Simplification in JavaDL

$$\{x := y\}\{z := x\}[\,](z \doteq y)$$

Update Simplification in JavaDL

$$\{\mathsf{x} := \mathsf{y}\}\{\mathsf{z} := \mathsf{x}\}(\mathsf{z} \doteq \mathsf{y})$$

Update Simplification in JavaDL

$$\{x := y \,||\, \{x := y\}z := x\}(z \doteq y)$$

Update Simplification in JavaDL

$$\{\mathsf{x} := \mathsf{y} \,\|\, \mathsf{z} := \{\mathsf{x} := \mathsf{y}\}\mathsf{x}\}(\mathsf{z} \doteq \mathsf{y})$$

Update Simplification in JavaDL

$$\{x := y \,||\, z := y\}(z \doteq y)$$

Update Simplification in JavaDL

$$\{\mathsf{z} := \mathsf{y}\}(\mathsf{z} \doteq \mathsf{y})$$

Update Simplification in JavaDL

$$\{z := y\}z \doteq \{z := y\}y$$

Update Simplification in JavaDL

$$y \doteq \{z := y\}y$$

Update Simplification in JavaDL

$$y \doteq y$$

Update Simplification in JavaDL

$$\mathtt{y} \doteq \mathtt{y} \quad \checkmark$$

(1) Removal of Ineffective "Assignables"

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{x}, \mathtt{y} :\approx \mathtt{x})\}(\mathtt{x} > 17)$$

(1) Removal of Ineffective "Assignables"

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx \mathtt{x})\}(\mathtt{x} > 17)$$

(1) Removal of Ineffective Abstract Updates

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{x}, \mathtt{y} :\approx \mathtt{x})\}(\mathtt{z} > 17)$$

(1) Removal of Ineffective Abstract Updates

$\texttt{z} > 17$

(2.1) Application of Concrete on Abstract Updates

(2.1) Application of Concrete on Abstract Updates

$$\{\mathrm{x} := 17 \,||\, \mathrm{y} := \mathrm{z}\}$$

(2.1) Application of Concrete on Abstract Updates

$$\{\mathrm{x} := 17 \,||\, \mathrm{y} := \mathrm{z}\}\{\mathcal{U}_{\mathrm{P}}(\mathrm{x} :\approx \mathrm{x}, \mathrm{y})\}$$

(2.1) Application of Concrete on Abstract Updates

$$\{\mathtt{x} := 17 \,||\, \mathtt{y} := \mathtt{z}\}\{\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx \mathtt{x}, \mathtt{y})\}(\mathtt{z} > 0)$$

(2.1) Application of Concrete on Abstract Updates

$$\{\mathtt{x} := 17\}\{\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx 17, \mathtt{z})\} \qquad (\mathtt{z} > 0)$$

(2.1) Application of Concrete on Abstract Updates

$$\{\mathtt{x} := 17\}\{\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx 17, \mathtt{z})\}\{\mathtt{y} := \mathtt{z}\}(\mathtt{z} > 0)$$

(2.1) Application of Concrete on Abstract Updates

$$\{\mathtt{x} := 17\}\{\mathcal{U}_{\mathtt{P}}(\mathtt{x}^{!} :\approx 17, \mathtt{z})\}\{\mathtt{y} := \mathtt{z}\}(\mathtt{z} > 0)$$

(2.1) Application of Concrete on Abstract Updates

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{x}^! :\approx 17, \mathtt{z})\}\{\mathtt{y} := \mathtt{z}\}(\mathtt{z} > 0)$$

Three categories of Abstract Update Simplification Rules (2.2) Application of Abstract on Concrete Updates

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{y}^{!} :\approx \mathtt{z})\}\{\mathtt{x} := \mathtt{y}\}\varphi(\mathtt{x})$$

Three categories of Abstract Update Simplification Rules (2.2) Application of Abstract on Concrete Updates

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{x}^! :\approx \mathtt{z})\}\varphi(\mathtt{x})$$

Three categories of Abstract Update Simplification Rules (3.1) Application of Abstract on Abstract Updates

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx \mathtt{y})\}\{\mathcal{U}_{\mathtt{Q}}(\mathtt{z} :\approx \mathtt{w})\}\varphi$$

Three categories of Abstract Update Simplification Rules
(3.1) Application of Abstract on Abstract Updates

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx \mathtt{y}) \circ \mathcal{U}_{\mathtt{Q}}(\mathtt{z} :\approx \mathtt{w})\}\varphi$$

Three categories of Abstract Update Simplification Rules (3.2) Permutation of Abstract Updates in Concatenations

$$\{\mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx \mathtt{y}) \circ \mathcal{U}_{\mathtt{Q}}(\mathtt{z} :\approx \mathtt{w})\}\varphi$$

Three categories of Abstract Update Simplification Rules (3.2) Permutation of Abstract Updates in Concatenations

$$\{\mathcal{U}_{\mathtt{Q}}(\mathtt{z} :\approx \mathtt{w}) \circ \mathcal{U}_{\mathtt{P}}(\mathtt{x} :\approx \mathtt{y})\}\varphi$$