# Software analysis: the very idea

Software Analysis
Topic 3

---

Carlo A. Furia

USI – Università della Svizzera Italiana

# Today's menu

- Software analysis concepts
  - Soundness and completeness
  - Expressiveness and automation
  - Trade-offs
- A simple imperative language
  - Operational semantics

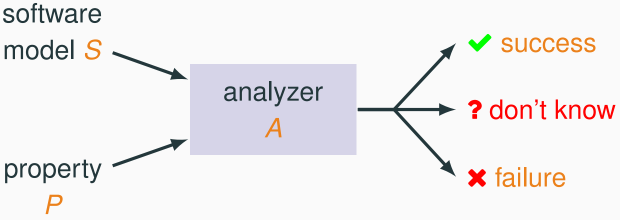
# Software analysis concepts

# Software analysis: the very idea

"Software analysis" denotes techniques, methods, and tools useful to establish that some software behaves according to some properties.

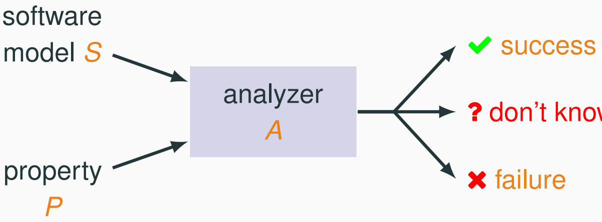
# Software analysis: the very idea

"Software analysis" denotes techniques, methods, and tools useful to establish that some software behaves according to some properties.



# Software analysis: the very idea

"Software analysis" denotes techniques, methods, and tools useful to establish that some software behaves according to some properties.

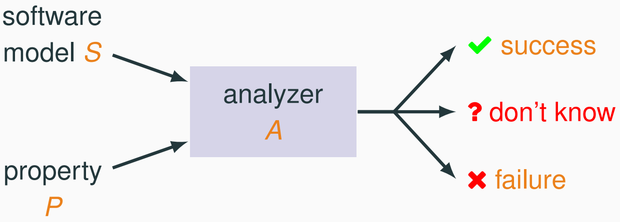



Software model $S$ (often called implementation):

- source code, byte code, binaries, automaton model, logic formula, ...
- possibly auxiliary annotations (loop invariants, proof scripts, type annotations, ...)

# Software analysis: the very idea

"Software analysis" denotes techniques, methods, and tools useful to establish that some software behaves according to some properties.

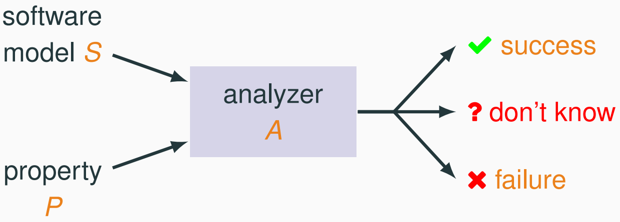



Property $P$ (often called specification):

- logic formula, automaton model, reference implementation, implicit property, sampled expected outputs (oracle), ...

# Software analysis: the very idea

"Software analysis" denotes techniques, methods, and tools useful to establish that some software behaves according to some properties.

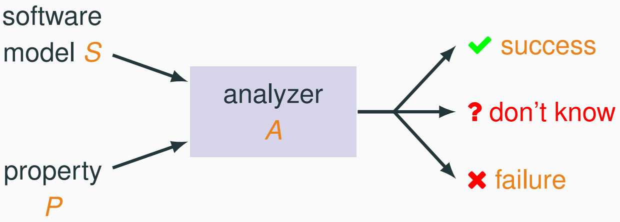



Analyzer $A$ (verifier, verification tool, analysis tool):

- automated tool, interactive system, human, . . .

# Software analysis: the very idea

"Software analysis" denotes techniques, methods, and tools useful to establish that some software behaves according to some properties.




Outcome $A(S, P)$ (verification output):

- ✔ **success**: $S$ has property $P$
- ✖ **failure**: $S$ does not have property $P$
- ? **don't know**: inconclusive analysis: doesn't terminate, timeout, out of memory, crash, forced termination, ...

# Software analysis concepts

## Soundness and completeness

# Soundness

An analyzer $A$ is sound if:

$$A(S, P) = ✔ \quad \text{implies} \quad S \models P$$

With a sound verifier: we can trust the analysis when it is successful.

# Soundness

An analyzer $A$ is sound if:

$$A(S, P) = \checkmark \quad \text{implies} \quad S \models P$$

With a sound verifier: we can trust the analysis when it is successful.

Example of sound analyzer: the typechecker of a strongly typed programming language:

**system** $S$**:** program (with type declarations)

**property** $P$**:** (implicit) the program has no type mismatch errors

When typechecking is successful, there will be no type errors at runtime (for any input).

# Soundness of typechecking in Java

# Soundness of typechecking in Java

The Java type system is mostly sound.

```
static void typecheck_OK(int n, LinkedList<String> list) {
  long m = n;              // OK: widening
  List<String> l = list; // OK: covariant
}
```

# Soundness of typechecking in Java

The Java type system is mostly sound.

```java
static void typecheck_OK(int n, LinkedList<String> list) {
  long m = n;             // OK: widening
  List<String> l = list; // OK: covariant
}
```

```
$ javac Sound.java
# [no errors]
```

# Soundness of typechecking in Java

The Java type system is mostly sound.

```java
static void typecheck_OK(int n, LinkedList<String> list) {
  long m = n;             // OK: widening
  List<String> l = list; // OK: covariant
}
```

```
$ javac Sound.java
# [no errors]
```

```java
static void typecheck_KO(long m, List<String> list) {
   int n = m;                   // NO: possibly lossy
   LinkedList<String> l = list; // NO: contravariant
}
```

# Soundness of typechecking in Java

The Java type system is mostly sound.

```java
static void typecheck_OK(int n, LinkedList<String> list) {
  long m = n;             // OK: widening
  List<String> l = list; // OK: covariant
}
```

```
$ javac Sound.java
# [no errors]
```

```java
static void typecheck_KO(long m, List<String> list) {
   int n = m;                       // NO: possibly lossy
   LinkedList<String> l = list; // NO: contravariant
}
```

```
$ javac Sound.java
Sound.java:12: error: incompatible types: possible lossy conversion from long to int
     int n = m;                  // NO: possibly lossy
             ^
Sound.java:13: error: incompatible types: List<String> cannot be converted to LinkedList<String>
     LinkedList<String> l = list; // NO: contravariant
```

# Loopholes in Java typechecking

The Java type system is mostly sound, provided we do not use certain loophole features that typechecking effectively ignores.

```
static void loopholes(LinkedList<String> list) {
     // raw type List
   List l = list;
    // add integer to list of strings!
   l.add(10);
   l.add("hello");
}
```

# Loopholes in Java typechecking

The Java type system is mostly sound, provided we do not use certain loophole features that typechecking effectively ignores.

```java
static void loopholes(LinkedList<String> list) {
    // raw type List
    List l = list;
    // add integer to list of strings!
    l.add(10);
    l.add("hello");
}
```

```
$ javac Sound.java
# [no errors]
Note: Sound.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ javac -Xlint:unchecked Sound.java
Sound.java:9: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
     l.add("hello");
  where E is a type-variable: E extends Object declared in interface List
Sound.java:11: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
     l.add(10);
  where E is a type-variable: E extends Object declared in interface List
```

# Loopholes in Java typechecking

The Java type system is mostly sound, provided we do not use certain loophole features that typechecking effectively ignores.

```java
static void loopholes(LinkedList<String> list) {
    // raw type List
    List l = list;
    // add integer to list of strings!
    l.add(10);
    l.add("hello");
}
```

```
$ javac Sound.java
# [no errors]
Note: Sound.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ javac -Xlint:unchecked Sound.java
Sound.java:9: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
     l.add("hello");
  where E is a type-variable: E extends Object declared in interface List
Sound.java:11: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
     l.add(10);
  where E is a type-variable: E extends Object declared in interface List
```

Annotation `@SuppressWarnings("unchecked")` suppresses the warning.

# Unsoundness of typechecking in Java

It has been recently discovered that the Java type system is (non-deliberately) unsound when we use some features of constrained genericity.

*OOPSLA'16*, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984004

**Java and Scala's Type Systems are Unsound** *

The Existential Crisis of Null Pointers

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch

Ross Tate
Cornell University, USA
ross@cs.cornell.edu

# Unsoundness of typechecking in Java

It has been recently discovered that the Java type system is (non-deliberately) unsound when we use some features of constrained genericity.

```
class Unsound {
   static class Constrain<A, B extends A> { }

   static class Bind<A> {
      <B extends A> A upcast(Constrain<A,B> constrain, B b)
       { return b; }
   }

   static<T,U> U coerce(T t) {
      Constrain<U,? super T> constrain = null;
      Bind<U> bind = new Bind<U>();
      return bind.upcast(constrain, t);
   }

   public static void main(String[] args) {
      String zero = Unsound.<Integer,String>coerce(0);
   }
}
```

```
$ javac Unsound.java  # no errors
$ java Unsound
Exception in thread "main"
  java.lang.ClassCastException:
    java.base/java.lang.Integer
      cannot be cast to
    java.base/java.lang.String
```

The typechecker outputs ✔ but the program has a type error!

# Soundness in practice

An analyzer $A$ is sound if $A(S, P) = ✔$ implies $S \models P$

Software analysis is mostly interested in techniques that are sound.

However, soundness is typically traded-off against other properties:

# Soundness in practice

An analyzer $A$ is sound if $A(S, P) = ✔$ implies $S \models P$

Software analysis is mostly interested in techniques that are sound.

However, soundness is typically traded-off against other properties:

- trivially sound analyzer: always return ✖

# Soundness in practice

An analyzer $A$ is sound if $A(S, P) = ✔$ implies $S \models P$

Software analysis is mostly interested in techniques that are sound.

However, soundness is typically traded-off against other properties:

- trivially sound analyzer: always return ✖

Unsoundness on some input language features is acceptable (even desirable) as long as it enables achieving other properties (completeness, scalability, etc.).

# Soundiness

VIEWPOINT

## In Defense of Soundiness: A Manifesto

By Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, Dimitrios Vardoulakis

Communications of the ACM, February 2015, Vol. 58 No. 2, Pages 44-46
10.1145/2644805

# Soundiness

VIEWPOINT

## In Defense of Soundiness: A Manifesto

By Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, Dimitrios Vardoulakis
Communications of the ACM, February 2015, Vol. 58 No. 2, Pages 44-46
10.1145/2644805

# Completeness

An analyzer $A$ is complete if:

$A(S, P) \neq$ ✔ implies $S \not\models P$

With a complete verifier: when the analysis is unsuccessful there are real errors (violations of the property).

also: precise

# Completeness

$A(S, P) = ✖$ or $A(S, P) = ?$

An analyzer $A$ is complete if:

$A(S, P) \neq ✔$ implies $S \not\models P$

With a complete verifier: when the analysis is unsuccessful there are real errors (violations of the property).

also: precise

# Completeness

$A(S,P) = ✖$ or $A(S,P) = ?$

An analyzer $A$ is complete if:

$A(S,P) \neq ✔$ implies $S \not\models P$

With a complete verifier: when the analysis is unsuccessful there are real errors (violations of the property).

also: precise

Example of complete analyzer: a test-case generator:

**system** $S$**:** program

**property** $P$**:** oracle defining the expected behavior (e.g. assertions)

When a generated test triggers an assertion violation, we found an input that leads to error.

# Completeness of testing

Generated test:

```
@Test
public void testMySort() {
    String[] input = {"c", "a", "b"};           // input
    String[] output = mySort(input);            // run test
    assertEquals(sorted(input)[0], output[0]);  // compare oracle
                                                // and output
}
```

# Completeness of testing

Generated test:

```java
@Test
public void testMySort() {
    String[] input = {"c", "a", "b"};              // input
    String[] output = mySort(input);               // run test
    assertEquals(sorted(input)[0], output[0]);  // compare oracle
                                               // and output
}
```

```
$ java -cp /usr/share/java/junit4.jar:. org.junit.runner.JUnitCore MySortTest
There was 1 failure:
1) testMySort(MySortTest)
org.junit.ComparisonFailure: expected:<[a]> but was:<[c]>
```

# Completeness of testing

Generated test:

```java
@Test
public void testMySort() {
    String[] input = {"c", "a", "b"};              // input
    String[] output = mySort(input);               // run test
    assertEquals(sorted(input)[0], output[0]);  // compare oracle
                                                   // and output
}
```

```
$ java -cp /usr/share/java/junit4.jar:. org.junit.runner.JUnitCore MySortTest
There was 1 failure:
1) testMySort(MySortTest)
org.junit.ComparisonFailure: expected:<[a]> but was:<[c]>
```

This output conclusively indicates that `mySort` does not work correctly on input `{"c", "a", "b"}`.

# Soundness of testing

Testing is complete: a failing test conclusively indicates that there is an error – that is $S \not\models P$.

# Soundness of testing

Testing is complete: a failing test conclusively indicates that there is an error – that is $S \not\models P$.

```
JUnit version 4.12
OK (1000000 tests)
```

## Soundness of testing

Testing is complete: a failing test conclusively indicates that there is an error – that is $S \not\models P$.

```
JUnit version 4.12
OK (1000000 tests)
```

Testing is unsound: even if all tests pass there may still be other inputs that trigger an error.

## Soundness of testing

Testing is complete: a failing test conclusively indicates that there is an error – that is $S \not\models P$.

```
JUnit version 4.12
OK (1000000 tests)
```

Testing is unsound: even if all tests pass there may still be other inputs that trigger an error.

However, if you narrowly define the property $P$ you're testing for as limited to the inputs being tested, then testing becomes sound and complete – except possibly in case of non termination of the program under test.

```
@Test
public void testMySort() {
   String[] input = {"c", "a", "b"};
   String[] output = mySort(input);
   assertEquals("a", output[0]);
}
```

$P$:

```
mySort({"c", "a", "b"})[0] == "a"
```

# Completeness in practice

An analyzer $A$ is complete if $A(S, P) \neq$ ✔ implies $S \not\models P$

Any analyzer that tackles an undecidable problem cannot be both sound and complete.

# Completeness in practice

An analyzer $A$ is complete if $A(S, P) \neq$ ✔ implies $S \not\models P$

Any analyzer that tackles an undecidable problem cannot be both sound and complete.

- trivially sound, incomplete analyzer: always return ✖

# Completeness in practice

An analyzer $A$ is complete if $A(S, P) \neq$ ✔ implies $S \not\models P$

Any analyzer that tackles an undecidable problem cannot be both sound and complete.

- trivially sound, incomplete analyzer: always return ✖
- trivially complete, unsound analyzer: always return ✔

# Completeness in practice

An analyzer $A$ is complete if $A(S, P) \neq$ ✔ implies $S \not\models P$

Any analyzer that tackles an undecidable problem cannot be both sound and complete.

- trivially sound, incomplete analyzer: always return ✖
- trivially complete, unsound analyzer: always return ✔

Thus, sound analyzers are generally incomplete: a verification failure just means “don’t know” (not “there is an error”).

# Completeness in practice

An analyzer $A$ is complete if $A(S, P) \neq$ ✔ implies $S \not\models P$

Any analyzer that tackles an undecidable problem cannot be both sound and complete.

- trivially sound, incomplete analyzer: always return ✖
- trivially complete, unsound analyzer: always return ✔

Thus, sound analyzers are generally incomplete: a verification failure just means "don't know" (not "there is an error").

Some incompleteness is inevitable and is a manifestation of the impossibility of automatically solving complex analysis problems (undecidability).

# Degrees of soundness and completeness

It is useful to think of soundness and completeness not as binary properties but as on a spectrum.

- Typechecking is sound on programs that do not use unsafe operations.
- (Bounded) model checking is sound up to a fixed memory bound.
- Data-flow analysis is normally complete on program fragments without branching.

# Positives and negatives

The analyzer checks for errors (property violations) and its output is the outcome of the check:

- ✔ is a negative outcome (no errors found, no warning)
- ✖ is a positive outcome (errors found, warning)

| $A(S, P)$ | $S \models P$ | $S \not\models P$ |
|---|---|---|
| ✔ | true negative | false negative |
| ✖ | false positive | true positive |

# Positives and negatives

The analyzer checks for errors (property violations) and its output is the outcome of the check:

✔ is a negative outcome (no errors found, no warning)

✖ is a positive outcome (errors found, warning)

| $A(S, P)$ | $S \models P$ | $S \not\models P$ |
|---|---|---|
| ✔ | true negative | false negative |
| ✖ | false positive | true positive |

- A sound analyzer never issues false negatives
- A complete analyzer never issues false positives

# Software analysis concepts

## Expressiveness and automation

# Expressiveness

also: flexibility

The expressiveness of an analyzer $A$ is a measure of the variety and extensiveness of the properties $P$ it can analyze.

# Expressiveness

also: flexibility

The expressiveness of an analyzer $A$ is a measure of the <u>variety and extensiveness</u> of the properties $P$ it can analyze.

Main kinds of properties:

**full-fledged logic:** first-order logic, higher-order logic, ...

**specialized language:** temporal logic, reachability, ...

**fixed/implicit:** null safety, type safety, termination, ...

# Automation

The level of automation of an analyzer $A$ is a measure of how much human effort it needs beyond providing system model $S$ and property $P$.

# Automation

The level of automation of an analyzer $A$ is a measure of how much human effort it needs beyond providing system model $S$ and property $P$.

Main levels of automation:

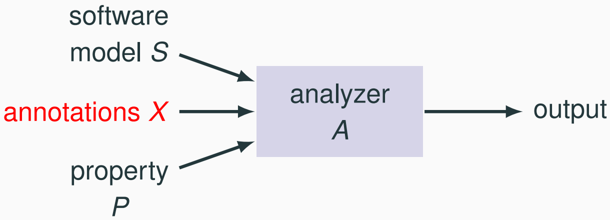
**automatic (push-button):** the analyzer works completely automatically

**auto-active:** the user interacts with the analyzer indirectly in a series of iteration by providing additional annotations to guide the analysis – which is itself automatic

**interactive:** the user guides the analyzer interactively at crucial steps

# Auto-active tools

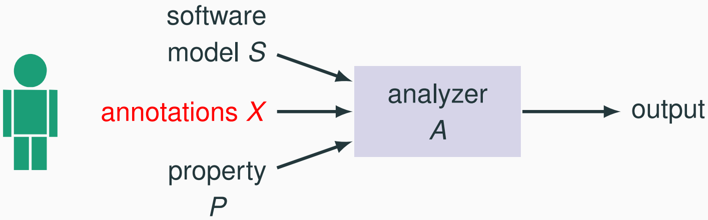
Auto-active is a portmanteau of automatic and interactive.




1. Prepare initial input
2. Run analysis
3. Inspect output
4. Revise annotations

# Auto-active tools

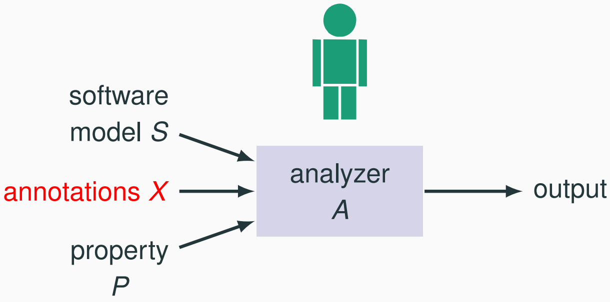
Auto-active is a portmanteau of automatic and interactive.




1. Prepare initial input
2. Run analysis
3. Inspect output
4. Revise annotations

# Auto-active tools

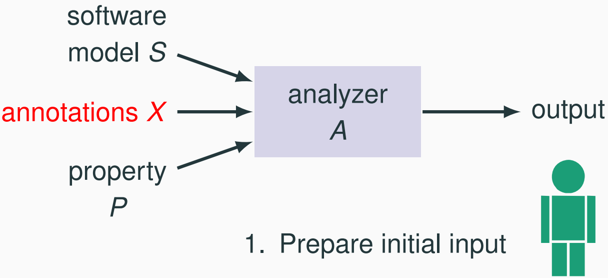
Auto-active is a portmanteau of automatic and interactive.




1. Prepare initial input
2. Run analysis
3. Inspect output
4. Revise annotations

# Auto-active tools

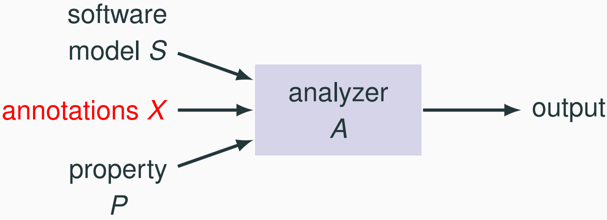
Auto-active is a portmanteau of automatic and interactive.




1. Prepare initial input
2. Run analysis
3. Inspect output
4. Revise annotations

# Auto-active tools

Auto-active is a portmanteau of automatic and interactive.




1. Prepare initial input
2. Run analysis
3. Inspect output
4. Revise annotations

# Software analysis concepts

## Trade-offs

# Trade-offs

We have already seen that there is a trade-off between soundness and completeness.

There is also a trade-off between soundness, expressiveness, and automation:

- many software analysis problems are undecidable
- a usable analyzer can only implement tractable algorithms

Soundness, expressiveness, and automation: something has to give

# Trade-offs in practice

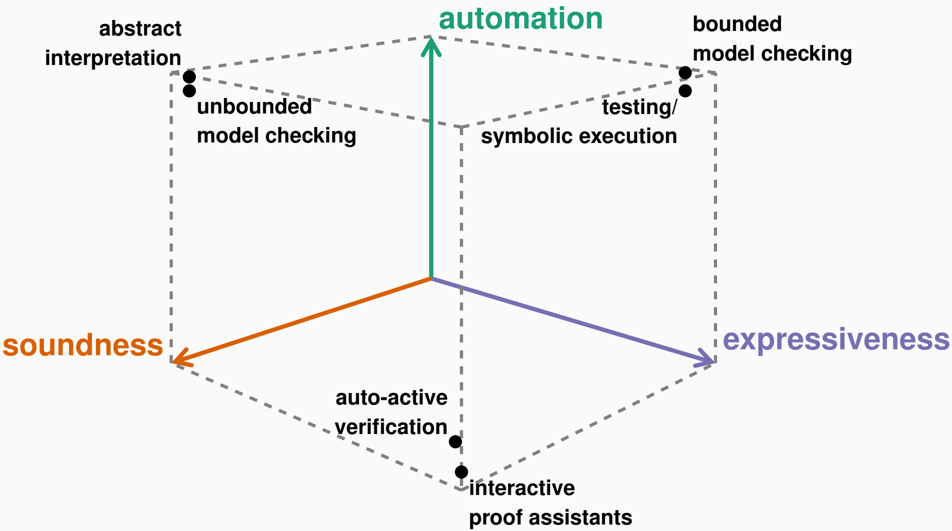
**tractable problems:** restrict the analysis to systems $S$ and properties $P$ that can be analyzed exhaustively – losing <u>expressiveness</u>

**under-approximation:** analyze a (typically finite) subset of all possible behaviors of system $S$ – losing <u>soundness</u>

**over-approximation:** analyze an abstract superset of all possible behaviors of system $S$

- manual abstraction: precise – losing <u>automation</u>
- automated abstraction: imprecise and working only with fixed properties $P$ – losing <u>expressiveness</u>

# Dimensions in program verification




From Nadia Polikarpova: Dimensions in program verification

# A simple imperative language

# The Helium language

We will use a series of simple imperative languages to write the programs we will analyze.

The smallest language is called Helium and does not have any modular constructs (procedures or functions).

$$
\begin{aligned}
\mathit{He} &::= \mathit{Statement}^* \\
\mathit{Statement} &::= \mathit{Declaration} \mid \mathit{Active} \\
\mathit{Declaration} &::= \mathit{VariableDeclaration} \\
\mathit{Active} &::= \mathit{Skip} \mid \mathit{Assignment} \mid \mathit{Conditional} \mid \mathit{Loop}
\end{aligned}
$$

# Helium: declarations and expressions

Helium variables only use scalar types: integers, booleans, and generic types (whose values can be copied and compared for equality).

Integers have infinite precision (also called: mathematical integers), and hence overflows cannot happen.

$$
\begin{aligned}
\textit{VariableDeclaration} &::= \texttt{var}\ \texttt{v}_1\texttt{, ..., v}_n\texttt{:}\ \textit{Type} \\
\textit{Type} &::= \textbf{\texttt{Integer}} \mid \textbf{\texttt{Boolean}} \mid \texttt{TypeId} \\
\textit{Expression} &::= \texttt{(}\textit{Expression}\texttt{)} \mid \texttt{v} \in \textit{Variables} \\
&\quad \mid \textit{BooleanExpression} \mid \textit{ArithmeticExpression} \\
\textit{BooleanExpression} &::= \texttt{true} \mid \texttt{false} \mid \textit{RelationalExpression} \\
&\quad \mid \textit{Expression} \land \textit{Expression} \mid \cdots \\
\textit{RelationalExpression} &::= \textit{Expression} \leq \textit{Expression} \mid \cdots \\
\textit{ArithmeticExpression} &::= \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \cdots \mid \textit{Expression} + \textit{Expression} \mid \cdots
\end{aligned}
$$

## Helium: active statements

The only unusual feature of active statements is that assignments can involve multiple variables <u>in parallel</u>.

$$\begin{aligned}
\textit{Skip} &::= \texttt{skip} \\
\textit{Assignment} &::= \texttt{v}_1\texttt{, ..., v}_n\texttt{:=}\ \textit{Expression}_1\, , \ldots ,\textit{Expression}_n \\
\textit{Conditional} &::= \texttt{if}\ \textit{Expression Statement}^{+}\ \left[\texttt{else}\ \textit{Statement}^{+}\right] \\
\textit{Loop} &::= \texttt{while}\ \textit{Expression Statement}^{+}
\end{aligned}$$

# Maximum of two integers

Helium programs leave input and output implicit.

```
var x, y, max: Integer
// inputs: x, y
if x > y
  max := x
else
  max := y
// output: max
```

# Integer power

Helium uses indentation to group statements.

```
var x, y, power: Integer // inputs: x, y
var n: Integer
n, power := y, 1         // parallel assignment
while n > 0
  pow := pow * x
  n := n - 1
```

# Integer power

Helium uses indentation to group statements.

```
var x, y, power: Integer // inputs: x, y
var n: Integer
n, power := y, 1         // parallel assignment
while n > 0
  pow := pow * x
  n := n - 1
```

Alternatively, we can group multiple statements with braces and separate them with semicolons:

```
while (n > 0) { pow := pow * x; n := n - 1 }
```

## Helium: semantics in Java

To present the semantics of Helium in a familiar way we sketch a translation $\mathcal{T}$ of Helium programs into Java.

$$\mathcal{T}\Big(\texttt{var v}_1\texttt{, ..., v}_n\texttt{: T}\Big) = \mathcal{T}(T)\ \texttt{v}_1\texttt{, ..., v}_n\texttt{;}$$

$$\mathcal{T}\Big(\textbf{Boolean}\Big) = \textbf{boolean}$$

$$\mathcal{T}\Big(\textbf{Integer}\Big) = \texttt{BigInteger}$$

$$\mathcal{T}\Big(\texttt{TypeId}\Big) = \texttt{TypeIdClass} \text{ which implements equality}$$

$$\mathcal{T}\Big(E_1 + E_2\Big) = \mathcal{T}(E_1)\texttt{.add(}\mathcal{T}(E_2)\texttt{)} \text{ and similar for } \texttt{-}, *, *, \texttt{mod}$$

$$\mathcal{T}\Big(E_1 \wedge E_2\Big) = \mathcal{T}(E_1)\ \texttt{\&\&}\ \mathcal{T}(E_2) \text{ and similar for } \vee, \Longrightarrow, \ldots$$

$$\mathcal{T}\Big(E_1 \leq E_2\Big) = \mathcal{T}(E_1)\texttt{.compareTo(}\mathcal{T}(E_2)\texttt{) <= 0}$$

$$\text{and similar for } <, =, \ldots$$

```
class TypeIdClass {
  public TypeIdClass() { }
  int compareTo(TypeIdClass other) { return (this == other) ? 0 : 1; }
}
```

# Helium: semantics in Java

A block of Helium statements is translated into a sequential block of Java statements:

$$\mathcal{T}\left(S_1\ S_2\right) = \mathcal{T}(S_1);\mathcal{T}(S_2)$$

# Helium: semantics in Java

$$\mathcal{T}\Big(\texttt{skip}\Big) = \texttt{;}$$

$$\begin{aligned}
\mathcal{T}\Big(\mathtt{v}_1, \ldots, \mathtt{v}_n := E_1, \ldots, E_n\Big) = \{\, & T_1\ t_1 = \mathtt{t}_1.\mathtt{clone}(); \ldots \\
& T_m\ t_m = \mathtt{t}_m.\mathtt{clone}(); \\
& \mathtt{v}_1 = \mathcal{T}(E_1)[\mathtt{t}_1 \mapsto t_1, \ldots, \mathtt{t}_m \mapsto t_m]; \\
& \ldots \\
& \mathtt{v}_n = \mathcal{T}(E_n)[\mathtt{t}_1 \mapsto t_1, \ldots, \mathtt{t}_m \mapsto t_m];\ \}
\end{aligned}$$

where $\{\mathtt{t}_1, \ldots, \mathtt{t}_m\} = \mathcal{V}(E_1, \ldots, E_n)$ are all variables appearing in any $E_1, \ldots, E_m$ (of types $T_1, \ldots, T_m$) whose values are copied into fresh local variables $t_1, \ldots, t_m$ before being used, so as to correctly express the semantics of parallel assignment.

# Helium: semantics in Java

$$\mathcal{T}\Big(\texttt{skip}\Big) = ;$$

$$\mathcal{T}\Big(\mathtt{v}_1, \ldots, \mathtt{v}_n := E_1, \ldots, E_n\Big) = \{ T_1\ t_1 = \mathtt{t}_1\texttt{.clone();} \ldots$$
$$T_m\ t_m = \mathtt{t}_m\texttt{.clone();}$$
$$\mathtt{v}_1 = \mathcal{T}(E_1)[\mathtt{t}_1 \mapsto t_1, \ldots, \mathtt{t}_m \mapsto t_m];$$
$$\ldots$$
$$\mathtt{v}_n = \mathcal{T}(E_n)[\mathtt{t}_1 \mapsto t_1, \ldots, \mathtt{t}_m \mapsto t_m];\ \}$$

where $\{\mathtt{t}_1, \ldots, \mathtt{t}_m\} = \mathcal{V}(E_1, \ldots, E_n)$ are all variables appearing in any $E_1, \ldots, E_m$ (of types $T_1, \ldots, T_m$) whose values are copied into fresh local variables $t_1, \ldots, t_m$ before being used, so as to correctly express the semantics of parallel assignment.

```
var x, y: Integer
x, y := y, x   // swap x and y
```

```
BigInteger x, y;
{ BigInteger _x = x.clone();
  BigInteger _y = y.clone();
  x = _y;  // BigInteger is immutable
  y = _x; }
```

# Helium: semantics in Java

$$\mathcal{T}\Big(\texttt{if}\ C\ T\ \texttt{else}\ E\Big) = \texttt{if}\ (\mathcal{T}(C))\ \{\mathcal{T}(T)\}\ \texttt{else}\ \{\mathcal{T}(E)\}$$

# Helium: semantics in Java

$$\mathcal{T}\left(\textbf{if}\ C\ T\ \textbf{else}\ E\right) = \textbf{if}\ (\mathcal{T}(C))\ \{\mathcal{T}(T)\}\ \textbf{else}\ \{\mathcal{T}(E)\}$$

```
var x, y, max: Integer
if x > y
  max := x
else
  max := y
```

```
BigInteger x, y, max;
if (x.compareTo(y) > 0)
  { max = x; }
else
  { max = y; }
```

# Helium: semantics in Java

$$\mathcal{T}\left(\texttt{while}\ C\ B\right) = \texttt{while}\ \texttt{(}\mathcal{T}(C)\texttt{)}\ \{\mathcal{T}(B)\}$$

# Helium: semantics in Java

$$\mathcal{T}\left(\texttt{while}\ C\ B\right) = \texttt{while}\ (\mathcal{T}(C))\ \{\mathcal{T}(B)\}$$

```
var x, y, power, n: Integer
n, power := y, 1
while n > 0
  pow := pow * x
  n := n - 1
```

```
BigInteger x, y, power, n;
BigInteger _y = y.clone();
n = _y;
power = new BigInteger("1");
while (n.compareTo(new BigInteger("0")) > 0)
  pow = pow.multiply(x);
  n = n.subtract(new BigInteger("1"));
}
```

# A simple imperative language

## Operational semantics

# Operational semantics

Using Java to define semantics is:

# Operational semantics

Using Java to define semantics is:

Pros (Yay! 👍):

- easy to understand

Cons (Nay! 👎):

- unsuitable for mathematical analysis
- not abstract: using a more complex language (Java) to describe a much simpler one (Helium)

# Operational semantics

Using Java to define semantics is:

Pros (Yay! 👍):

- easy to understand

Cons (Nay! 👎):

- unsuitable for mathematical analysis
- not abstract: using a more complex language (Java) to describe a much simpler one (Helium)

A better alternative is an operational semantics.

An operational semantics defines the behavior of programs in terms of how executing each statement modifies an abstract program state.

# Operational semantics

An operational semantics consists of reduction (or evaluation) rules that define transitions between states.

# Operational semantics

An operational semantics consists of reduction (or evaluation) rules that define transitions between states.

The program state (sometimes called store, stack, or environment)

$$s : \quad \mathit{Variables} \rightarrow \mathit{Values}$$

assigns a value (of suitable type) to every program variable.

# Operational semantics

An operational semantics consists of reduction (or evaluation) rules that define transitions between states.

The program state (sometimes called store, stack, or environment)

$$s : \quad \mathit{Variables} \to \mathit{Values}$$

assigns a value (of suitable type) to every program variable.

Transitions between states are defined by reduction rules:

requirements

$$\frac{R}{\langle S, s \rangle \rightsquigarrow s'}$$

program fragment current state new state

# Operational semantics

An operational semantics consists of reduction (or evaluation) rules that define transitions between states.

The program state (sometimes called store, stack, or environment)

$$s : \quad \mathit{Variables} \to \mathit{Values}$$

assigns a value (of suitable type) to every program variable.

Transitions between states are defined by reduction rules:

requirements

$$\frac{R}{\langle S, s\rangle \rightsquigarrow s'}$$

program fragment

current state

new state

When $R$ holds, executing $S$ in state $s$ leads to state $s'$.

# Helium: operational semantics

Blocks of statements lead to a sequence of transitions:

$$\frac{\langle S_1, s\rangle \rightsquigarrow s' \quad \langle S_2, s'\rangle \rightsquigarrow s''}{\langle S_1 \; ; \; S_2, s\rangle \rightsquigarrow s''}$$

# Helium: operational semantics

Declaring variables extends the state with the declared variables mapping to undefined values ?. Redeclaring a previously declared variable is undefined.

$$\frac{\forall 1 \leq k \leq n \bullet \mathsf{v}_k \notin \mathit{domain}(s)}{\langle \texttt{var v}_1\texttt{, ..., v}_n\texttt{: T}, s\rangle \rightsquigarrow s \cup \bigcup_{k=1,\dots,n}\{\mathsf{v}_k \rightarrow ?\}}$$

# Helium: operational semantics

Declaring variables extends the state with the declared variables mapping to undefined values ?. Redeclaring a previously declared variable is undefined.

$$\frac{\forall 1 \leq k \leq n \bullet \mathrm{v}_k \notin \mathit{domain}(s)}{\langle \texttt{var v}_1\texttt{, ..., v}_n\texttt{: T}, s\rangle \rightsquigarrow s \cup \bigcup_{k=1,\dots,n} \{\mathrm{v}_k \to ?\}}$$

Evaluating an expression does not change the state but depends on the state:

$$[\![E]\!]_s$$

denotes the value of expression $E$ in state $s$. We omit the evaluation rules for expressions, which should be straightforward.

# Helium: operational semantics

Executing **`skip`** does not change the program state:

$$\frac{}{\langle \texttt{skip}, s\rangle \rightsquigarrow s}$$

# Helium: operational semantics

Executing **skip** does not change the program state:

$$\frac{}{\langle \texttt{skip}, s\rangle \rightsquigarrow s}$$

Parallel assignment is only defined if the assigned variables are all different:

$$\frac{\mathsf{v}_1, \ldots, \mathsf{v}_n \text{ all different} \quad [\![E_1]\!]_s = e_1 \ \cdots \ [\![E_n]\!]_s = e_n}{\langle \mathsf{v}_1\,, \ldots\,, \mathsf{v}_n \ \texttt{:=}\ E_1\,, \ldots\,, E_n\,,\ s\rangle \rightsquigarrow s[\mathsf{v}_1 \mapsto e_1, \ldots, \mathsf{v}_n \mapsto e_n]}$$

# Helium: operational semantics

Conditional statements have two rules according to whether the condition is true or false:

$$\frac{[\![C]\!]_s = \top \quad \langle T, s\rangle \rightsquigarrow s'}{\langle \texttt{if}\ C\ T\ \texttt{else}\ E, s\rangle \rightsquigarrow s'} \qquad \frac{[\![C]\!]_s = \bot \quad \langle E, s\rangle \rightsquigarrow s'}{\langle \texttt{if}\ C\ T\ \texttt{else}\ E, s\rangle \rightsquigarrow s'}$$

The case of no `else` branch is just a shorthand for an empty `else` branch:

$$\frac{\langle \texttt{if}\ C\ T\ \texttt{else skip}, s\rangle \rightsquigarrow s'}{\langle \texttt{if}\ C\ T, s\rangle \rightsquigarrow s'}$$

# Helium: operational semantics

Loop statements have two rules according to whether the exit condition is true or false:

$$\frac{[\![C]\!]_s = \top \quad \langle B, s\rangle \rightsquigarrow s' \quad \langle \texttt{while}\; C\; B, s'\rangle \rightsquigarrow s''}{\langle \texttt{while}\; C\; B, s\rangle \rightsquigarrow s''} \qquad \frac{[\![C]\!]_s = \bot}{\langle \texttt{while}\; C\; B, s\rangle \rightsquigarrow s}$$

# Operational semantics: big-step vs. small-step

The style of operational semantics we have used is called big-step (also: natural semantics) because it defines the overall semantics (how the state changes) of each programming construct.

# Operational semantics: big-step vs. small-step

The style of operational semantics we have used is called big-step (also: natural semantics) because it defines the overall semantics (how the state changes) of each programming construct.

A different style is called small-step because it defines the individual rewriting steps which, combined, lead to the overall semantics.

Examples of small-step semantics rules:

| RECURSIVE CASES | BASE CASES |
|---|---|
| $\dfrac{\langle S_1, s\rangle \longrightarrow \langle S_1', s'\rangle}{\langle S_1\ S_2, s\rangle \longrightarrow \langle S_1'\ S_2, s'\rangle}$ | $\dfrac{\langle S_1, s\rangle \longrightarrow s'}{\langle S_1\ S_2, s\rangle \longrightarrow \langle S_2, s'\rangle}$ |
| $\dfrac{\langle E, s\rangle \longrightarrow \langle E', s\rangle}{\langle \mathtt{v}\ := E, s\rangle \longrightarrow \langle \mathtt{v}\ := E', s\rangle}$ | $\dfrac{\langle x, s\rangle \longrightarrow x}{\langle \mathtt{v}\ := x, s\rangle \longrightarrow s[v \mapsto x]}$ |
| $\dfrac{\langle C, s\rangle \longrightarrow \langle C', s\rangle}{\langle \mathbf{if}\ C\ T\ \mathbf{else}\ E, s\rangle \longrightarrow \langle \mathbf{if}\ C'\ T\ \mathbf{else}\ E, s\rangle}$ | $\dfrac{}{\langle \mathbf{if}\ \top\ T\ \mathbf{else}\ E, s\rangle \longrightarrow \langle T, s\rangle}$ |

# Summary

# Summary

Since in software analysis we are mainly interested in sound analyses, we have to give up some completeness whenever we deal with undecidable problems.

Expressiveness and automation are two other important dimensions to characterize the capabilities of a software analyzer.

As a first step towards doing formal software analysis we have defined a simple imperative language (Helium) and its formal operational semantics.

# References

This class's title is after Fetzer's paper Program verification: the very idea, Communications of the ACM, 1988.

The term auto-active was coined by Leino and Moskal in Usable auto-active verification, 2010.

# These slides' license