# Semantics and Equality

L. Thomas van Binsbergen

November 21, 2022

# Type 4 clones

**Type 4**: functionality is the same, code may be completely different.

# Type 4 clones

**Type 4**: functionality is the same, code may be completely different.

In other words, two program fragments are *equivalent*..

# Type 4 clones

**Type 4**: functionality is the same, code may be completely different.

In other words, two program fragments are *equivalent*..

Is the relation of type 4 clone pairs an equivalence relation?

# Type 4 clones

**Type 4**: functionality is the same, code may be completely different.

In other words, two program fragments are *equivalent*..

Is the relation of type 4 clone pairs an equivalence relation?

How could we determine whether program fragments are equivalent?

# Syntax vs Semantics

What is the syntax of a language?

# Syntax vs Semantics

What is the syntax of a language?

What is the semantics of a language?

# Are these the same programs?

```java
int n = 10;
int acc = 1;
int i = 2;
while (i <= n) {
  acc = acc * i;
  i++;
}
System.out.println(acc);
```

```java
int n = 10;
int acc = 1;
for (int i = 2; i <= n; i++) {
  acc *= i;
}
System.out.println(acc);
```

# Are these the same programs?

```
int n = 10;
int acc = 1;
int i = 2;
while (i <= n) {
  acc = acc * i;
  i++;
}
System.out.println(acc);
```

```
int n = 10;
int acc = 1;
for (int i = 2; i <= n; i++) {
  acc *= i;
}
System.out.println(acc);
```

This depends on our notion of equality..

# Are these the same programs?

```java
int n = 10;
int acc = 1;
int i = 2;
while (i <= n) {
  acc = acc * i;
  i++;
}
System.out.println(acc);
```

```java
int n = 10;
int acc = 1;
for (int i = 2; i <= n; i++) {
  acc *= i;
}
System.out.println(acc);
```

This depends on our notion of equality..

- **Structural equality**: programs are syntactically equal

# Are these the same programs?

```java
int n = 10;
int acc = 1;
int i = 2;
while (i <= n) {
  acc = acc * i;
  i++;
}
System.out.println(acc);
```

```java
int n = 10;
int acc = 1;
for (int i = 2; i <= n; i++) {
  acc *= i;
}
System.out.println(acc);
```

This depends on our notion of equality..

- **Structural equality**: programs are syntactically equal
- **Semantic equivalence**: programs have equal semantics (semantics need to be specified)

# Are these the same programs?

```java
int n = 10;
int acc = 1;
int i = 2;
while (i <= n) {
  acc = acc * i;
  i++;
}
System.out.println(acc);
```

```java
int n = 10;
int acc = 1;
for (int i = 2; i <= n; i++) {
  acc *= i;
}
System.out.println(acc);
```

This depends on our notion of equality..

- **Structural equality**: programs are syntactically equal
- **Semantic equivalence**: programs have equal semantics (semantics need to be specified)
  - **Mathematical equality**: programs 'denote' the same mathematical object, e.g. $1+2 = 7-4$

# Are these the same programs?

```
int n = 10;
int acc = 1;
int i = 2;
while (i <= n) {
  acc = acc * i;
  i++;
}
System.out.println(acc);
```

```
int n = 10;
int acc = 1;
for (int i = 2; i <= n; i++) {
  acc *= i;
}
System.out.println(acc);
```

This depends on our notion of equality..

- **Structural equality**: programs are syntactically equal
- **Semantic equivalence**: programs have equal semantics (semantics need to be specified)
  - **Mathematical equality**: programs 'denote' the same mathematical object,e.g. $1+2 = 7-4$
  - **Operational equivalence**: programs 'behave' the same

# Overview

1. Structural equality
   Term rewriting
2. Mathematical equality
   Denotational semantics
3. Operational equivalence
   Small-step Operational Semantics

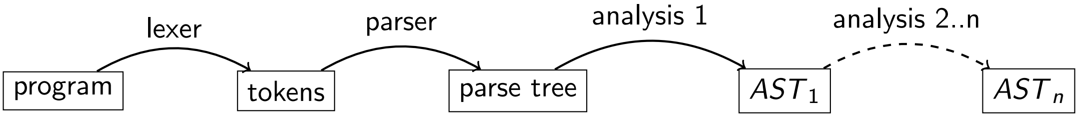
Section 1

# Structural equality

# Multiple stages of syntactic analysis

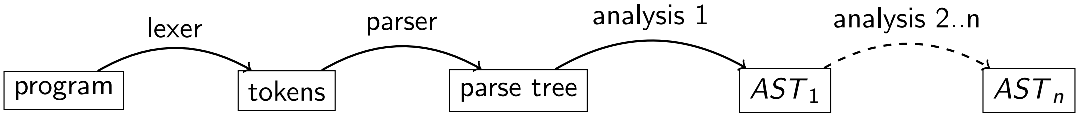
$$
\begin{tikzcd}
\boxed{\text{program}} \arrow[r, bend left, "\text{lexer}"] & \boxed{\text{tokens}} \arrow[r, bend left, "\text{parser}"] & \boxed{\text{parse tree}} \arrow[r, bend left, "\text{analysis 1}"] & \boxed{AST_1} \arrow[r, bend left, dashed, "\text{analysis 2..n}"] & \boxed{AST_n}
\end{tikzcd}
$$

# Multiple stages of syntactic analysis




- syntax analyser: a *language* is defined as the set of all sentences (where sentences are defined by a concrete grammar, such as a BNF grammar)

# Multiple stages of syntactic analysis




- syntax analyser: a *language* is defined as the set of all sentences
  (where sentences are defined by a concrete grammar, such as a BNF grammar)
- semantics analyser: a *language* is the set of all ASTs
  (where programs are defined by an abstract grammar, such as an algebraic datatype)

# Multiple stages of syntactic analysis

$$\begin{tikzcd}
\boxed{\text{program}} \arrow[r, bend left, "\text{lexer}"] & \boxed{\text{tokens}} \arrow[r, bend left, "\text{parser}"] & \boxed{\text{parse tree}} \arrow[r, bend left, "\text{analysis 1}"] & \boxed{AST_1} \arrow[r, bend left, dashed, "\text{analysis 2..n}"] & \boxed{AST_n}
\end{tikzcd}$$

- syntax analyser: a *language* is defined as the set of all sentences
  (where sentences are defined by a concrete grammar, such as a BNF grammar)
- semantics analyser: a *language* is the set of all ASTs
  (where programs are defined by an abstract grammar, such as an algebraic datatype)
- so where in the picture are “programs” according to these two definitions?

# Structurally equal?

```
int x; /* some layout here */
int y;
```

```
int x; int y;
```

# Structurally equal?

```
int x; /* some layout here */
int y;
```

```
int x; int y;
```

**lexically equal**

# Structurally equal?

```
int x; /* some layout here */
int y;
```

```
int x; int y;
```

**lexically equal**

```
if (x < 10) return x;
```

```
if (x < 10) {
  return x;
}
```

**lexically equal?**

# Structurally equal?

```
int x; /* some layout here */
int y;
```

```
int x; int y;
```

**lexically equal**

```
if (x < 10) return x;
```

```
if (x < 10) {
  return x;
}
```

**lexically equal?**

**structurally equal as ASTs**

# Structurally equal?

```
int x; /* some layout here */
int y;
```

```
int x; int y;
```

**lexically equal**

```
if (x < 10) return x;
```

```
if (x < 10) {
  return x;
}
```

**lexically equal?**

**structurally equal as ASTs**

$$
\begin{tikzcd}
\boxed{\text{program}} \arrow[r, bend left, "\text{lexer}"] & \boxed{\text{tokens}} \arrow[r, bend left, "\text{parser}"] & \boxed{\text{parse tree}} \arrow[r, bend left, "\text{analysis 1}"] & \boxed{AST_1} \arrow[r, bend left, dashed, "\text{analysis 2..n}"] & \boxed{AST_n}
\end{tikzcd}
$$

# Structurally equal?

```
System.out.println(getValue());
```

```
System.out.println(this.getValue());
```

# Structurally equal?

```
System.out.println(getValue());
```

```
System.out.println(this.getValue());
```

**depends on static information, i.e. static/instance method**

# Structurally equal?

```
System.out.println(getValue());
```

```
System.out.println(this.getValue());
```

**depends on static information, i.e. static/instance method**

```
class MyClass {
  void mymethod() {
  }
}
```

```
class MyClass {
  protected void mymethod() {
  }
}
```

# Structurally equal?

```
System.out.println(getValue());
```

```
System.out.println(this.getValue());
```

**depends on static information, i.e. static/instance method**

```
class MyClass {
  void mymethod() {
  }
}
```

```
class MyClass {
  protected void mymethod() {
  }
}
```

**cannot be structurally equal, since semantically different**

# Structurally equal?

```
System.out.println(getValue());
```

```
System.out.println(this.getValue());
```

**depends on static information, i.e. static/instance method**

```
class MyClass {
  void mymethod() {
  }
}
```

```
class MyClass {
  protected void mymethod() {
  }
}
```

**cannot be structurally equal, since semantically different**

```
class MyClass {
  void mymethod() {
  }
}
```

```
class MyClass {
  void mymethod() {
    return;
  }
}
```

# Structurally equal?

```java
System.out.println(getValue());
```

```java
System.out.println(this.getValue());
```

**depends on static information, i.e. static/instance method**

```java
class MyClass {
  void mymethod() {
  }
}
```

```java
class MyClass {
  protected void mymethod() {
  }
}
```

**cannot be structurally equal, since semantically different**

```java
class MyClass {
  void mymethod() {
  }
}
```

```java
class MyClass {
  void mymethod() {
    return;
  }
}
```

**operationally equivalent, structural equality not so clear..**

Subsection 1

## Term rewriting

# Term rewriting

Term rewriting is a simple computational paradigm based on repeated application of rules.

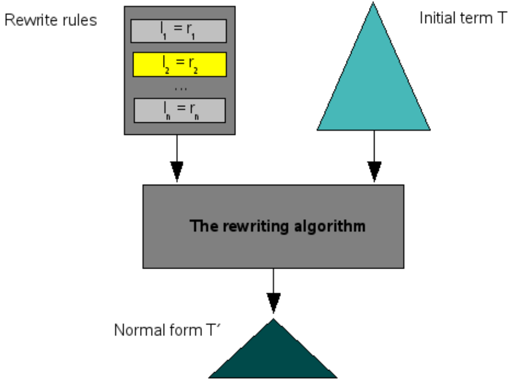
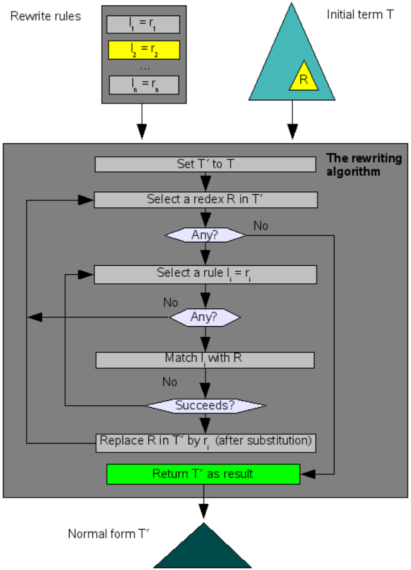


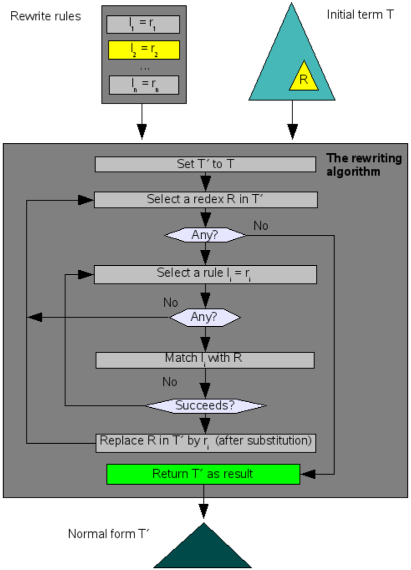

Figure: ©Paul Klint: `https://homepages.cwi.nl/~daybuild/daily-books/extraction-transformation/term-rewriting/term-rewriting.html`

# Term rewriting




$\textbf{true} \;\&\&\; X \Rightarrow X$
$X \;\&\&\; \textbf{true} \Rightarrow X$
$\textbf{false} \;||\; X \Rightarrow X$
$X \;||\; \textbf{false} \Rightarrow X$
$\textbf{true} \;?\; X : Y \Rightarrow X$
$\textbf{false} \;?\; X : Y \Rightarrow Y$

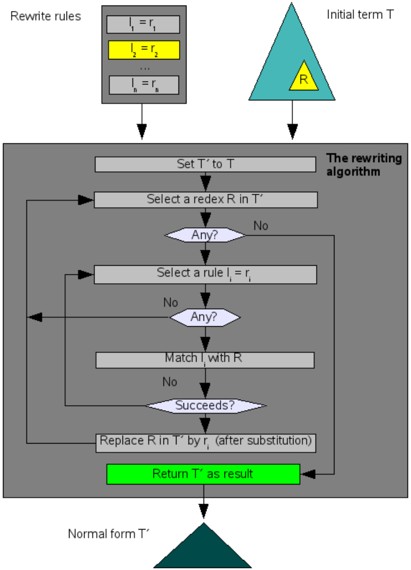
# Term rewriting




$$
\begin{array}{l}
\textbf{true} \ \&\& \ X \Rightarrow X \\
X \ \&\& \ \textbf{true} \Rightarrow X \\
\textbf{false} \ || \ X \Rightarrow X \\
X \ || \ \textbf{false} \Rightarrow X \\
\textbf{true} \ ? \ X : Y \Rightarrow X \\
\textbf{false} \ ? \ X : Y \Rightarrow Y
\end{array}
$$

- How to determine which redex to choose?

# Term rewriting




$$
\begin{array}{l}
\textbf{true} \;\&\&\; X \Rightarrow X \\
X \;\&\&\; \textbf{true} \Rightarrow X \\
\textbf{false} \;||\; X \Rightarrow X \\
X \;||\; \textbf{false} \Rightarrow X \\
\textbf{true} \;?\; X : Y \Rightarrow X \\
\textbf{false} \;?\; X : Y \Rightarrow Y
\end{array}
$$

- How to determine which redex to choose?
- How to determine order between rules?

# Term writing – simple example

Terms: Java expressions

Rules:

$$
\begin{array}{l}
\textbf{true} \ \&\& \ X \Rightarrow X \\
X \ \&\& \ \textbf{true} \Rightarrow X \\
\textbf{false} \ || \ X \Rightarrow X \\
X \ || \ \textbf{false} \Rightarrow X \\
\textbf{true} \ ? \ X \ \textbf{:} \ Y \Rightarrow X \\
\textbf{false} \ ? \ X \ \textbf{:} \ Y \Rightarrow Y
\end{array}
$$

# Term writing – simple example

Terms: Java expressions

Rules:

$$\begin{array}{l} \textbf{true}\ \&\&\ X \Rightarrow X \\ X\ \&\&\ \textbf{true} \Rightarrow X \\ \textbf{false}\ \|\ X \Rightarrow X \\ X\ \|\ \textbf{false} \Rightarrow X \\ \textbf{true}\ ?\ X : Y \Rightarrow X \\ \textbf{false}\ ?\ X : Y \Rightarrow Y \end{array}$$

What does the following expression rewrite to in this system?

$$((\textbf{true}\ \&\&\ \textbf{false})\ ?\ \textbf{true} : \textbf{false})\ \|\ \textbf{false}$$

# Rewriting

Term rewriting can be used to make programs *structurally equal*.

```
return (true ? null : new String("hello"));
return (false ? new String("world") : null);
```

# Rewriting

Term rewriting can be used to make programs *structurally equal*.

```
return (true ? null : new String("hello"));
return (false ? new String("world") : null);
```

**Normalisation**: apply rewrites to a program until it reaches some normal or canonical form.
$\hookrightarrow$ Requires *confluence* and *termination*

# Rewriting

Term rewriting can be used to make programs *structurally equal*.

```
return (true ? null : new String("hello"));
return (false ? new String("world") : null);
```

**Normalisation**: apply rewrites to a program until it reaches some normal or canonical form.
$\hookrightarrow$ Requires *confluence* and *termination*

Rewrite rules can be *semantics preserving*, but they do not have to be

# Confluence

$$
\begin{tikzcd}
& (11 + 9) \times (2 + 4) \arrow[dl, "\text{eval left}"'] \arrow[dr, "\text{eval right}"] & \\
20 \times (2 + 4) \arrow[dr, "\text{eval right}"'] & & (11 + 9) \times 6 \arrow[dl, "\text{eval left}"] \\
& 20 \times 6 \arrow[d] & \\
& 120 &
\end{tikzcd}
$$

# Causes of non-termination

## Untamed growth

Rewrites produce new subterm that envelopes the original redex:

$$\mathsf{X} \;?\; \mathsf{Y} : \mathsf{Z} \Rightarrow \textbf{true} \;?\; (\mathsf{X} \;?\; \mathsf{Y} : \mathsf{Z}) : \textbf{false}$$

# Causes of non-termination

## Untamed growth

Rewrites produce new subterm that envelopes the original redex:

X ? Y : Z $\Rightarrow$ **true** ? (X ? Y : Z) : **false**

## Cyclic rewrites

Rewrites produce the original redex:

X && Y $\Rightarrow$ Y && X

# Rewriting in rascal

**Name** Rascal/Expressions/Visit

**Synopsis** Visit the elements in a tree or value.

**Syntax**

```
Strategy visit ( Exp ) {
case PatternWithAction1;
case PatternWithAction2;
...
default: ...
}
```

The visit expression is optionally preceded by one of the following strategy indications that determine the traversal order of the subject:

- `top-down`: visit the subject from root to leaves.
- `top-down-break`: visit the subject from root to leaves, but stop at the current path when a case matches.
- `bottom-up`: visit the subject from leaves to root (this is the default).
- `bottom-up-break`: visit the subject from leaves to root, but stop at the current path when a case matches.
- `innermost`: repeat a bottom-up traversal as long as the traversal changes the resulting value (compute a fixed-point).
- `outermost`: repeat a top-down traversal as long as the traversal changes the resulting value (compute a fixed-point).

# Rewriting in Rascal – example

```
module Rewriting

import IO;
import Set;
import List;
import String;
import util::Maybe;

import lang::java::m3::Core;
import lang::java::m3::AST;

void eval(loc file) {
  if (\compilationUnit(_,[\class(_,_,_,[\method(_,_,_,_,\block(stmts))])])
       := createAstFromFile(file, true)) {
    for (\expressionStatement(\methodCall(_,_,_,[expr])) <- rewrite(stmts)) {
      println(expr);
    }
  }
}

&T rewrite(&T term) = innermost visit(term) {
       case \bracket(X0) => X0
       case \infix(\booleanLiteral(true),"&&",X1) => X1
       case \infix(X2:\booleanLiteral(false),"&&",_) => X2
       case \infix(X3,"&&",\booleanLiteral(true)) => X3
       case \infix(_,"&&",X4:\booleanLiteral(false)) => X4

       case \infix(X5:\booleanLiteral(true),"||",_) => X5
       case \infix(_,"||",X6:booleanLiteral(true)) => X6
       case \infix(\booleanLiteral(false),"||",X7) => X7
       case \infix(X8,"||",\booleanLiteral(false)) => X8

       case \conditional(\booleanLiteral(true),X9,_) => X9
       case \conditional(\booleanLiteral(false),_,X10) => X10
      };
```

```java
class Booleans {

  public static void main(String[] args) {
    System.out.println(
        true && (false && true) && true && false);
    System.out.println(
        true && (false && true) ? (false || true) : false);
    System.out.println(false||true);
    System.out.println(
        (true && (true && true)) ? (false || true) : false);
    System.out.println(
        true && (false && true) ? null : true ? null : null);
  }
}
```

Section 2

# Mathematical equality

Subsection 1

# Denotational semantics

# Denotational vs Operational Semantics

| | denotational | operational |
|---|---|---|
| origins: | **$\lambda$-calculus** | **(abstract) machines** |
| semantic assignment: | **mathematical object** | **transition system (traces)** |
| variables: | **$\lambda$-abstraction** | **configuration component** |
| effects: | monads | **auxiliary entities** |
| modular effects: | monad transformers | product category |
| advantages | more abstract | more concrete, detailed |
| $\hookrightarrow$ | **formal reasoning** | **evaluation order** |
| traditional targets langs: | (purely) functional | imperative & concurrent |
| $\hookrightarrow$ | expression languages | command languages |

# Denotational vs Operational Semantics

| | denotational | operational |
|---|---|---|
| origins: | **$\lambda$-calculus** | **(abstract) machines** |
| semantic assignment: | **mathematical object** | **transition system (traces)** |
| variables: | **$\lambda$-abstraction** | **configuration component** |
| effects: | monads | **auxiliary entities** |
| modular effects: | monad transformers | product category |
| advantages | more abstract | more concrete, detailed |
| $\hookrightarrow$ | **formal reasoning** | **evaluation order** |
| traditional targets langs: | (purely) functional | imperative & concurrent |
| $\hookrightarrow$ | expression languages | command languages |

Observation: distinction fades when implemented in a functional language using monads...

# Mathematically equal?

$(11 + 9) * (2 + 4)$ 120

# Mathematically equal?

$(11 + 9) * (2 + 4)$ $\qquad$ $120$

**depends on the meaning assigned to + and ∗**

# Mathematically equal?

$(11 + 9) * (2 + 4)$ $\qquad$ $120$

**depends on the meaning assigned to + and ***

$(11 + x) * (2 + 4)$ $\qquad$ $(11 + 9) * (2 + y)$

# Mathematically equal?

$(11 + 9) * (2 + 4)$ 120

**depends on the meaning assigned to + and ∗**

$(11 + x) * (2 + 4)$ $(11 + 9) * (2 + y)$

**depends on the bindings to x and y, e.g. x=9 and y=4 or x=-1 and y=1**

# Mathematically equal?

| | |
|---|---|
| $(11 + 9) * (2 + 4)$ | $120$ |

**depends on the meaning assigned to + and ∗**

| | |
|---|---|
| $(11 + x) * (2 + 4)$ | $(11 + 9) * (2 + y)$ |

**depends on the bindings to x and y, e.g. x=9 and y=4 or x=-1 and y=1**

| | |
|---|---|
| $0.1 + 0.1 + 0.1$ | $0.3$ |

# Mathematically equal?

| | |
|---|---|
| $(11 + 9) * (2 + 4)$ | $120$ |

**depends on the meaning assigned to + and ∗**

| | |
|---|---|
| $(11 + x) * (2 + 4)$ | $(11 + 9) * (2 + y)$ |

**depends on the bindings to x and y, e.g. x=9 and y=4 or x=-1 and y=1**

| | |
|---|---|
| $0.1 + 0.1 + 0.1$ | $0.3$ |

**mathematically yes..., but**

# Mathematically equal?

$(11 + 9) * (2 + 4)$ 120

**depends on the meaning assigned to + and ∗**

$(11 + x) * (2 + 4)$ $(11 + 9) * (2 + y)$

**depends on the bindings to x and y, e.g. x=9 and y=4 or x=-1 and y=1**

$0.1 + 0.1 + 0.1$ 0.3

**mathematically yes..., but**

**in our machine this depends on the chosen representation for reals, e.g. floating point precision**

# Mathematically equal?

(11 + 9) * (2 + 4) 120

**depends on the meaning assigned to + and ***

(11 + x) * (2 + 4) (11 + 9) * (2 + y)

**depends on the bindings to x and y, e.g. x=9 and y=4 or x=-1 and y=1**

0.1 + 0.1 + 0.1 0.3

**mathematically yes..., but**

**in our machine this depends on the chosen representation for reals, e.g. floating point precision**

(x >= 0) ? Math.sqrt(x) : 0 Math.sqrt(x)

# Mathematically equal?

$(11 + 9) * (2 + 4)$ 120

**depends on the meaning assigned to + and ∗**

$(11 + x) * (2 + 4)$ $(11 + 9) * (2 + y)$

**depends on the bindings to x and y, e.g. x=9 and y=4 or x=-1 and y=1**

$0.1 + 0.1 + 0.1$ 0.3

**mathematically yes..., but**

**in our machine this depends on the chosen representation for reals, e.g. floating point precision**

(x >= 0) ? Math.sqrt(x) : 0 Math.sqrt(x)

**depends on the** ***possible*** **values of x**

# Mathematically equal?

$(11 + 9) * (2 + 4)$ 120

**depends on the meaning assigned to + and ∗**

$(11 + x) * (2 + 4)$ $(11 + 9) * (2 + y)$

**depends on the bindings to x and y, e.g. x=9 and y=4 or x=-1 and y=1**

$0.1 + 0.1 + 0.1$ $0.3$

**mathematically yes..., but**

**in our machine this depends on the chosen representation for reals, e.g. floating point precision**

(x >= 0) ? Math.sqrt(x) : 0 Math.sqrt(x)

**depends on the *possible* values of x**

++x x+1 x = x+1

# Mathematically equal?

$(11 + 9) * (2 + 4)$ 120

**depends on the meaning assigned to + and ∗**

$(11 + x) * (2 + 4)$ $(11 + 9) * (2 + y)$

**depends on the bindings to x and y, e.g. x=9 and y=4 or x=-1 and y=1**

0.1 + 0.1 + 0.1 0.3

**mathematically yes..., but**

**in our machine this depends on the chosen representation for reals, e.g. floating point precision**

(x >= 0) ? Math.sqrt(x) : 0 Math.sqrt(x)

**depends on the *possible* values of x**

++x x+1 x = x+1

**denote the same value (in all contexts); do not have the same operational behaviour due to side-*effects***

# Denotational semantics – without variables

A denotational semantics describes a translation from expressions into a *semantic domain*. In this example from expressions into integer numbers:

$$[\![X * Y]\!] = [\![X]\!] \times [\![Y]\!]$$

$$[\![X + Y]\!] = [\![X]\!] + [\![Y]\!]$$

$$[\![X \mathbin{\&} Y]\!] = [\![X]\!] \wedge [\![Y]\!]$$

$$[\![X \mid Y]\!] = [\![X]\!] \vee [\![Y]\!]$$

$$[\![X \mathbin{?} Y : Z]\!] = \begin{cases} [\![Y]\!] & \textbf{if } [\![X]\!] = \mathbf{1} \\ [\![Z]\!] & \textbf{if } [\![X]\!] = \mathbf{0} \end{cases}$$

$$[\![\textbf{true}]\!] = \mathbf{1}$$

$$[\![\textbf{false}]\!] = \mathbf{0}$$

$$[\![X]\!] = X \quad \textbf{when } X \in \mathbb{Z}$$

# Denotational semantics – without variables

A denotational semantics describes a translation from expressions into a *semantic domain*. In this example from expressions into integer numbers:

$$[\![X * Y]\!] = [\![X]\!] \times [\![Y]\!]$$

$$[\![X + Y]\!] = [\![X]\!] + [\![Y]\!]$$

$$[\![X \mathbin{\&} Y]\!] = [\![X]\!] \wedge [\![Y]\!]$$

$$[\![X \mid Y]\!] = [\![X]\!] \vee [\![Y]\!]$$

$$[\![X \mathbin{?} Y : Z]\!] = \begin{cases} [\![Y]\!] & \textbf{if } [\![X]\!] = \mathbf{1} \\ [\![Z]\!] & \textbf{if } [\![X]\!] = \mathbf{0} \end{cases}$$

$$[\![\textbf{true}]\!] = \mathbf{1}$$

$$[\![\textbf{false}]\!] = \mathbf{0}$$

$$[\![X]\!] = X \quad \textbf{when } X \in \mathbb{Z}$$

**simplification: ignores overloading, numerical representations, ...**

# Denotational semantics – without variables

A denotational semantics describes a translation from expressions into a *semantic domain*.
In this example from expressions into integer numbers:

$$[\![X * Y]\!] = [\![X]\!] \times [\![Y]\!]$$

$$[\![X + Y]\!] = [\![X]\!] + [\![Y]\!]$$

$$[\![X \mathbin{\&} Y]\!] = [\![X]\!] \wedge [\![Y]\!]$$

$$[\![X \mid Y]\!] = [\![X]\!] \vee [\![Y]\!]$$

$$[\![X \mathbin{?} Y : Z]\!] = \begin{cases} [\![Y]\!] & \textbf{if } [\![X]\!] = \mathbf{1} \\ [\![Z]\!] & \textbf{if } [\![X]\!] = \mathbf{0} \end{cases}$$

$$[\![\textbf{true}]\!] = \mathbf{1}$$

$$[\![\textbf{false}]\!] = \mathbf{0}$$

$$[\![X]\!] = X \quad \textbf{when } X \in \mathbb{Z}$$

**simplification: ignores overloading, numerical representations, ...**

... and what is the meaning of $\wedge, \vee, \ldots$ ?

# Denotational semantics – with variables

Assume we understand the $\lambda$-calculus, and *abstract* over the *environment*.
The semantic domain is now the set of all $\lambda$-expressions over integers.

$$[\![X]\!] = (\lambda\rho.\ \rho(X)) \quad \textbf{if } X \in \textbf{identifiers},\ \rho(X) \neq \perp$$

# Denotational semantics – with variables

Assume we understand the $\lambda$-calculus, and *abstract* over the *environment*.
The semantic domain is now the set of all $\lambda$-expressions over integers.

$$[\![X]\!] = (\lambda\rho.\ \rho(X)) \quad \textbf{if } X \in \textbf{identifiers},\ \rho(X) \neq \perp$$

$$[\![\textbf{let } X = Y \textbf{ in } Z]\!] = (\lambda\rho.[\![Z]\!](\rho')) \quad \textbf{if } X \in \textbf{identifiers},\ \rho' = \rho[X \mapsto [\![Y]\!](\rho)]$$

# Denotational semantics – with variables

Assume we understand the $\lambda$-calculus, and *abstract* over the *environment*.
The semantic domain is now the set of all $\lambda$-expressions over integers.

$$[\![X]\!] = (\lambda\rho.\ \rho(X)) \quad \textbf{if } X \in \textbf{identifiers},\ \rho(X) \neq \bot$$

$$[\![\textbf{let } X = Y \textbf{ in } Z]\!] = (\lambda\rho.[\![Z]\!](\rho')) \quad \textbf{if } X \in \textbf{identifiers},\ \rho' = \rho[X \mapsto [\![Y]\!](\rho)]$$

Alternative formulation:

$$[\![X]\!](\rho) = \rho(X) \quad \textbf{if } X \in \textbf{identifiers},\ \rho(X) \neq \bot$$

# Denotational semantics – with variables

Assume we understand the $\lambda$-calculus, and *abstract* over the *environment*.
The semantic domain is now the set of all $\lambda$-expressions over integers.

$$[\![X]\!] = (\lambda\rho.\ \rho(X)) \quad \textbf{if } X \in \textbf{identifiers},\ \rho(X) \neq \bot$$

$$[\![\textbf{let } X = Y \textbf{ in } Z]\!] = (\lambda\rho.[\![Z]\!](\rho')) \quad \textbf{if } X \in \textbf{identifiers},\ \rho' = \rho[X \mapsto [\![Y]\!](\rho)]$$

Alternative formulation:

$$[\![X]\!](\rho) = \rho(X) \quad \textbf{if } X \in \textbf{identifiers},\ \rho(X) \neq \bot$$

$$[\![\textbf{let } X = Y \textbf{ in } Z]\!](\rho) = [\![Z]\!](\rho') \quad \textbf{if } X \in \textbf{identifiers},\ \rho' = \rho[X \mapsto [\![Y]\!](\rho)]$$

# Denotational semantics – additional effects?

$$[\![X \mathbin{\&\&} Y]\!] = ???$$

# Denotational semantics – additional effects?

$$[\![X \mathbin{\&\&} Y]\!] = ???$$

**the following is valid under the assumption that expressions have no additional effects:**

$$[\![X \mathbin{\&\&} Y]\!](\rho) = [\![X]\!](\rho) \wedge [\![Y]\!](\rho)$$

and note that we ignore properties such as running times and memory consumption

# Denotational semantics – additional effects?

$$[\![X \ \&\& \ Y]\!] = ???$$

**the following is valid under the assumption that expressions have no additional effects:**

$$[\![X \ \&\& \ Y]\!](\rho) = [\![X]\!](\rho) \wedge [\![Y]\!](\rho)$$

and note that we ignore properties such as running times and memory consumption

**if expressions do have effects, perhaps:**

$$[\![X \ \&\& \ Y]\!] = [\![X \ ? \ Y \ : \ \mathbf{false}]\!]$$

# Denotational semantics – additional effects?

$$[\![X \,\&\&\, Y]\!] = ???$$

**the following is valid under the assumption that expressions have no additional effects:**

$$[\![X \,\&\&\, Y]\!](\rho) = [\![X]\!](\rho) \wedge [\![Y]\!](\rho)$$

and note that we ignore properties such as running times and memory consumption

**if expressions do have effects, perhaps:**

$$[\![X \,\&\&\, Y]\!] = [\![X \;?\; Y \;:\; \mathbf{false}]\!]$$

**or:**

$$[\![X \,\&\&\, Y]\!](\rho) = \begin{cases} [\![Y]\!](\rho) & \textbf{if } [\![X]\!](\rho) = \mathbf{1} \\ \mathbf{0} & \textbf{if } [\![X]\!](\rho) = \mathbf{0} \end{cases}$$

**but where did the effects of the first operand go?**

# Denotational vs Operational Semantics

| | denotational | operational |
|---|---|---|
| origins: | **$\lambda$-calculus** | **(abstract) machines** |
| semantic assignment: | **mathematical object** | **transition system (traces)** |
| variables: | **$\lambda$-abstraction** | **configuration component** |
| effects: | monads | **auxiliary entities** |
| modular effects: | monad transformers | product category |
| advantages | more abstract | more concrete, detailed |
| $\hookrightarrow$ | **formal reasoning** | **evaluation order** |
| traditional targets langs: | (purely) functional | imperative & concurrent |
| $\hookrightarrow$ | expression languages | command languages |

# Mathematical equality?

*Two expressions $p_1$ and $p_2$ are equal in the context of $\rho$ if:*

$$[\![p_1]\!](\rho) = [\![p_2]\!](\rho)$$

Here we can choose $\rho$ to be empty for top-level expressions, or choose a context.

# Mathematical equality?

*Two expressions $p_1$ and $p_2$ are equal in the context of $\rho$ if:*

$$[\![p_1]\!](\rho) = [\![p_2]\!](\rho)$$

Here we can choose $\rho$ to be empty for top-level expressions, or choose a context.

For example, let $p_1 =$ `(11 + x) * (2 + 4)`, $p_2 =$ `(11 + 9) * (2 + y)` and $\rho = [x \mapsto -1, y \mapsto 1]$

# Mathematical equality?

*Two expressions $p_1$ and $p_2$ are equal in the context of $\rho$ if:*

$$[\![p_1]\!](\rho) = [\![p_2]\!](\rho)$$

Here we can choose $\rho$ to be empty for top-level expressions, or choose a context.

For example, let $p_1 =$ `(11 + x) * (2 + 4)`, $p_2 =$ `(11 + 9) * (2 + y)` and $\rho = [x \mapsto -1, y \mapsto 1]$

Alternatively, $p_1 =$ `(x >= 0) ? Math.sqrt(x) : 0`, $p_2 =$ `Math.sqrt(x)`, are equal for all $\rho$ with $\rho(x) \geq 0$

Section 3

# Operational equivalence

Subsection 1

## Small-step Operational Semantics

# Small-step, Operational Semantics

An SOS[1] specification defines a *transition system* as:

- A set of configurations, laying out the **terms** under evaluation and **contextual** and **mutable** *semantic entities*
- A set of labels, laying out the **input**, **output**, and **control** entities
- A labelled-transition relation over configurations $\gamma \xrightarrow{\alpha} \gamma'$
  The transition relation is defined through a collection of *inference rules*:

$$\frac{\text{zero or more premises and side conditions}}{\text{conclusion about transition relation}}$$

[1]SOS: Structural Operational Semantics. *A Structural Approach to Operational Semantics*. Plotkin 1981/2004

# Small-step, Operational Semantics

An SOS[1] specification defines a *transition system* as:

- A set of configurations, laying out the **terms** under evaluation and **contextual** and **mutable** *semantic entities*
- A set of labels, laying out the **input**, **output**, and **control** entities
- A labelled-transition relation over configurations $\gamma \xrightarrow{\alpha} \gamma'$
  The transition relation is defined through a collection of *inference rules*:

$$\frac{\text{zero or more premises and side conditions}}{\text{conclusion about transition relation}}$$

Shape of a premise or conclusion:

$$\textbf{contextual}^* \vdash \langle \textbf{term}, \textbf{mutable}^* \rangle \xrightarrow{\textbf{label}} \langle \textbf{term}', \textbf{mutables}'^* \rangle$$

[1] SOS: Structural Operational Semantics. *A Structural Approach to Operational Semantics*. Plotkin 1981/2004

# Small-step, Operational Semantics

An SOS[1] specification defines a *transition system* as:

- A set of configurations, laying out the **terms** under evaluation and **contextual** and **mutable** *semantic entities*
- A set of labels, laying out the **input**, **output**, and **control** entities
- A labelled-transition relation over configurations $\gamma \xrightarrow{\alpha} \gamma'$
  The transition relation is defined through a collection of *inference rules*:

$$\frac{\text{zero or more premises and side conditions}}{\text{conclusion about transition relation}}$$

Shape of a premise or conclusion:

$$\textbf{contextual}^* \vdash \langle \textbf{term}, \textbf{mutable}^* \rangle \xrightarrow{\textbf{label}} \langle \textbf{term}', \textbf{mutables}'^* \rangle$$

The semantics of a program are its *traces*, i.e. 'longest paths' in the transitive closure of $\rightarrow$

[1]SOS: Structural Operational Semantics. *A Structural Approach to Operational Semantics*. Plotkin 1981/2004

# Example trace

```
int x = 3 + 3;
System.out.println(x * 7);
```

# Example trace

```
int x = 3 + 3;
System.out.println(x * 7);
```

$$
\begin{aligned}
\langle X := 3 + 3; \mathit{println}(X * 7), [] \rangle &\xrightarrow{[]} \langle X := 6; \mathit{println}(X * 7), [] \rangle \\
&\xrightarrow{[]} \langle 6; \mathit{println}(X * 7), [X \mapsto 6] \rangle \\
&\xrightarrow{[]} \langle \mathit{println}(6 * 7), [X \mapsto 6] \rangle \\
&\xrightarrow{[]} \langle \mathit{println}(42), [X \mapsto 6] \rangle \\
&\xrightarrow{[42]} \langle \mathbf{void}, [X \mapsto 6] \rangle
\end{aligned}
$$

# Example trace

```
int x = 3 + 3;
System.out.println(x * 7);
```

$$
\begin{aligned}
\langle X := 3+3; \mathit{println}(X*7), []\rangle &\xrightarrow{[]} \langle X := 6; \mathit{println}(X*7), []\rangle \\
&\xrightarrow{[]} \langle 6; \mathit{println}(X*7), [X \mapsto 6]\rangle \\
&\xrightarrow{[]} \langle \mathit{println}(6*7), [X \mapsto 6]\rangle \\
&\xrightarrow{[]} \langle \mathit{println}(42), [X \mapsto 6]\rangle \\
&\xrightarrow{[42]} \langle \mathbf{void}, [X \mapsto 6]\rangle
\end{aligned}
$$

In other words, $X := 3+3; \mathit{println}(X*7)$

- evaluates to **void**
- produces output 42 and
- assigns 6 to $X$ (via 5 steps).

# SOS rules for variables

Example

- **mutable** entity store $\sigma$, representing variable assignments
- **label** entity output $\alpha$, representing printed values

$$\frac{\langle Y, \sigma\rangle \xrightarrow{\alpha} \langle Y', \sigma'\rangle}{\langle X := Y, \sigma\rangle \xrightarrow{\alpha} \langle X := Y', \sigma'\rangle}$$

$$\frac{V \in \mathbb{Z} \quad X \in \textbf{identifiers} \quad \sigma' = \sigma[X \mapsto V]}{\langle X := V, \sigma\rangle \xrightarrow{[]} \langle V, \sigma'\rangle}$$

$$\frac{X \in \textbf{identifiers} \quad V = \sigma(X)}{\langle X, \sigma\rangle \xrightarrow{[]} \langle V, \sigma\rangle}$$

# SOS rules for printing

Example

- **mutable** entity store $\sigma$, representing variable assignments
- **label** entity output $\alpha$, representing printed values

$$\frac{\langle X, \sigma \rangle \xrightarrow{\alpha} \langle X', \sigma' \rangle}{\langle \mathit{println}(X), \sigma \rangle \xrightarrow{\alpha} \langle \mathit{println}(X'), \sigma' \rangle}$$

$$\frac{V \in \mathbb{Z}}{\langle \mathit{println}(V), \sigma \rangle \xrightarrow{[V]} \langle \mathbf{void}, \sigma \rangle}$$

# SOS rules for ‘&&’

Example

- **mutable** entity store $\sigma$, representing variable assignments
- **label** entity output $\alpha$, representing printed values

$$\frac{\langle X, \sigma \rangle \xrightarrow{\alpha} \langle X', \sigma' \rangle}{\langle X \mathbin{\&\&} Y, \sigma \rangle \xrightarrow{\alpha} \langle X' \mathbin{\&\&} Y, \sigma' \rangle}$$

$$\frac{}{\langle \textbf{true} \mathbin{\&\&} Y, \sigma \rangle \xrightarrow{[]} \langle Y, \sigma' \rangle}$$

$$\frac{}{\langle \textbf{false} \mathbin{\&\&} Y, \sigma \rangle \xrightarrow{[]} \langle \textbf{false}, \sigma' \rangle}$$

# SOS rules for (other) infix operators

A left to right evaluation order on binary infix operators is specified by the following rules:

$$\frac{\langle X, \sigma \rangle \xrightarrow{\alpha} \langle X', \sigma' \rangle}{\langle X * Y, \sigma \rangle \xrightarrow{\alpha} \langle X' * Y, \sigma' \rangle}$$

$$\frac{V \in \mathbb{Z} \quad \langle Y, \sigma \rangle \xrightarrow{\alpha} \langle Y', \sigma' \rangle}{\langle V * Y, \sigma \rangle \xrightarrow{\alpha} \langle V * Y', \sigma' \rangle}$$

$$\frac{V_3 = V_1 \times V_2}{\langle V_1 * V_2, \sigma \rangle \xrightarrow{[]} \langle V_3, \sigma \rangle}$$

Other infix operators would have very similar rules.
Interesting comparison with denotational semantics of operators, e.g. $[\![X * Y]\!] = [\![X]\!] \times [\![Y]\!]$

# SOS semantics for **while**

How can we specify the semantics of the **while** construct?

# SOS semantics for **while**

How can we specify the semantics of the **while** construct?

$$\frac{\langle C, \sigma\rangle \xrightarrow{\alpha} \langle C', \sigma'\rangle}{\langle \textbf{while}(C)\ B, \sigma\rangle \xrightarrow{\alpha} \langle \textbf{while}(C')\ B, \sigma'\rangle}$$

$$\frac{}{\langle \textbf{while}(\textbf{true})\ B, \sigma\rangle \xrightarrow{[]} \langle ???, \sigma'\rangle}$$

# SOS semantics for **while**

How can we specify the semantics of the **while** construct?

$$\frac{\langle C, \sigma\rangle \xrightarrow{\alpha} \langle C', \sigma'\rangle}{\langle \textbf{while}(C)\ B, \sigma\rangle \xrightarrow{\alpha} \langle \textbf{while}(C')\ B, \sigma'\rangle}$$

$$\frac{}{\langle \textbf{while}(\textbf{true})\ B, \sigma\rangle \xrightarrow{[]} \langle ???, \sigma'\rangle}$$

Problem: we 'lost' the original condition

# SOS semantics for **while**

How can we specify the semantics of the **while** construct?

$$\frac{\langle C, \sigma\rangle \xrightarrow{\alpha} \langle C', \sigma'\rangle}{\langle \textbf{while}(C)\ B, \sigma\rangle \xrightarrow{\alpha} \langle \textbf{while}(C')\ B, \sigma'\rangle}$$

$$\frac{}{\langle \textbf{while}(\textbf{true})\ B, \sigma\rangle \xrightarrow{[]} \langle ???, \sigma'\rangle}$$

Problem: we 'lost' the original condition

Simple alternative, relying on **if-then** construct and recursive nature of transitions:

$$\frac{}{\langle \textbf{while}(C)\ B, \sigma\rangle \xrightarrow{[]} \langle \textbf{if}(C)\ \{B; \textbf{while}(C)\ B\}, \sigma\rangle}$$

# Operational equivalence

Are the following programs operationally equivalent?

```
int i = 0;
while (i <= 100) {
  if (i % 2 == 0) {
    System.out.println(i);
  }
  i = i + 1;
}
```

```
int i = 0;
while (i <= 100) {
  System.out.println(i);
  i = i + 2;
}
```

# Operational equivalence

Are the following programs operationally equivalent?

```
int i = 0;
while (i <= 100) {
  if (i % 2 == 0) {
    System.out.println(i);
  }
  i = i + 1;
}
```

```
int i = 0;
while (i <= 100) {
  System.out.println(i);
  i = i + 2;
}
```

- Approach: Define the *yield* of a trace and compare yields
- Definition 1: the yield is the concatenation of all output

# Operational equivalence

Are the following programs operationally equivalent?

```
int i = 0;
while (i <= 100) {
  if (i % 2 == 0) {
    System.out.println(i);
  }
  i = i + 1;
}
```

```
int i = 0;
while (i <= 100) {
  System.out.println(i);
  i = i + 2;
}
```

- Approach: Define the *yield* of a trace and compare yields
- Definition 1: the yield is the concatenation of all output
- Definition 2: the yield is all output and the deltas between stores

# Operational equivalence

Are the following programs operationally equivalent?

```
int i = 0;
while (i <= 100) {
  if (i % 2 == 0) {
    System.out.println(i);
  }
  i = i + 1;
}
```

```
int i = 0;
while (i <= 100) {
  System.out.println(i);
  i = i + 2;
}
```

- Approach: Define the *yield* of a trace and compare yields
- Definition 1: the yield is the concatenation of all output
- Definition 2: the yield is all output and the deltas between stores
- ...
- Definition i: the yield is the return value (a **control** entity) of a (pure) function
- ...
- Definition n: ...

# Denotational vs Operational Semantics

| | denotational | operational |
|---|---|---|
| origins: | **$\lambda$-calculus** | **(abstract) machines** |
| semantic assignment: | **mathematical object** | **transition system (traces)** |
| variables: | **$\lambda$-abstraction** | **configuration component** |
| effects: | monads | **auxiliary entities** |
| modular effects: | monad transformers | product category |
| advantages | more abstract | more concrete, detailed |
| $\hookrightarrow$ | **formal reasoning** | **evaluation order** |
| traditional targets langs: | (purely) functional | imperative & concurrent |
| $\hookrightarrow$ | expression languages | command languages |

# Small step evaluation in Rascal

**Name** Rascal/Statements/Solve

**Synopsis** Solve a set of equalities by fixed-point iteration.

**Syntax** `solve(`$Var_1$`, `$Var_2$`, ..., `$Var_n$`; `*Exp*`) `*Statement*`;`

**Description** Rascal provides a solve statement for performing arbitrary fixed-point computations. This means, repeating a certain computation as long as it causes changes. This can, for instance, be used for the solution of sets of simultaneous linear equations but has much wider applicability.

# Small step evaluation in Rascal – example

```
module SmallStep

import IO;
import String;
import util::Maybe;

import lang::java::m3::Core;
import lang::java::m3::AST;

alias source = tuple[node,store];
alias target = Maybe[tuple[value, store, output]];
alias store = map[str,value];
alias output = list[str];


void eval(loc file) {
  if (\compilationUnit(_,[\class(_,_,_,[\method(_,_,_,_,stmt)])]) := createAstFromFile(file, true)) {
     output out = [];
     store sto = ();
     solve(stmt) {
       if(just(<stmt_, sto_, out_>) := step(<stmt, sto>)) {
           out = out + out_;
           stmt = stmt_;
           sto = sto_;
       }
     }
     for (str s <- out) {print(s);}
  }
}
```

# Type 4 clones

**Type 4**: functionality is the same, code may be completely different.

In other words, two program fragments are *equivalent*..

How could we determine whether two program fragments are equivalent?

# Semantics and Equality

L. Thomas van Binsbergen

November 21, 2022