Language Semantics and Interpretation

ADPRO Fall 2024

Florian Biermann
Andrzej Wąsowski

Which language is this?

```
print (1/2)
```

- Valid in
  - Python
  - Ruby

- What does this fragment print?

  - Python → 0.5
  - Ruby → 0

Talking About Languages

A programming language is:

- **Syntax**
  - What may be recognized as a valid program?
- **Semantics**
  - How is a val evaluated?
- **Implementation**
  - Which C++ compiler do you use?
  - GCC, MinGW, MSVC…

In Scala…

In Java…

In C++…

A Tiny Arithmetic Language

$$
\begin{aligned}
e \ ::= &\ \mathrm{num}[n] \quad n \in \mathbb{Z} \\
\mid &\ e + e
\end{aligned}
$$

Abstract syntax

$$\frac{}{\mathbf{num}[n] \Downarrow \mathbf{num}[n]}\ (num)$$

$$\frac{e_1 \Downarrow \mathbf{num}[n] \quad e_2 \Downarrow \mathbf{num}[m]}{e_1 + e_2 \Downarrow \mathbf{num}[n+m]}\ (add)$$

Judgment form = $\dfrac{\text{Premise}}{\text{Judgment}}$

Operational Semantics

- “How to evaluate a program”
- We do **evaluation semantics**
  - “big-step”
  - “natural”
  - Invented by Khan
- Structural semantics
  - “ small-step”
  - Invented by Plotkin
- Reduction semantics
  - Invented by Felleisen & Hieb

A transition system between expressions.

Evaluation via Derivation

$$
\begin{aligned}
e \ ::= &\ \mathrm{num}[n] \quad n \in \mathbb{Z} \\
\mid &\ e + e
\end{aligned}
$$

$$\frac{}{\mathrm{num}[n] \Downarrow \mathrm{num}[n]}\ (num)$$

$$\frac{e_1 \Downarrow \mathrm{num}[n] \quad e_2 \Downarrow \mathrm{num}[m]}{e_1 + e_2 \Downarrow \mathrm{num}[n+m]}\ (add)$$

Algorithm: for each expression, apply a rule that matches.

$$\mathrm{num}[42] + \mathrm{num}[21] + \mathrm{num}[23] \Downarrow$$

Evaluation order is under-specified!

Semantics Is Not An Algorithm!

- Operational semantics may be under-specified
  - Different evaluation orders may be allowed
  - Especially useful for automatic parallelization
- Semantics specifies an interpreter!
  - May require some choices to evaluation order
  - As long as the interpreter lives up to the semantics, it is a good interpreter.
- There are other ways of specification: english!
  - ECMA standard
  - Java Language and Virtual Machine Specifications
  - C++ language specification
  - …

TAL + conditionals + functions

$$
\begin{aligned}
e ::= {} & \mathrm{num}[n] \\
| {} & x \\
| {} & e + e \\
| {} & \text{if } e \text{ then } e \text{ else } e \\
| {} & \lambda\, x.\, e \\
| {} & e(e)
\end{aligned}
$$

“Lambda calculus with numbers, addition and conditionals.”

$\lambda\, x. e =$ `x => e`

$$\frac{e_1 \Downarrow \mathrm{num}[0]}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow e_3}\ (ff)$$

$$\frac{e_1 \Downarrow \mathrm{num}[n]}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow e_2}\ n \neq 0\ (tt)$$

Side-condition

$$\frac{e_0 \Downarrow \mathrm{num}[n]}{\lambda\, x.\, e(e_0) \Downarrow e[x/\mathrm{num}[n]]}\ (app)$$

Substitute all occurrences of x for num[n] in e.

Derivations!!

$$\frac{e_1 \Downarrow \text{num}[0]}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow e_3}\ (ff) \qquad \frac{e_0 \Downarrow \text{num}[n]}{\lambda\, x.\, e(e_0) \Downarrow e[x/\text{num}[n]]}\ (app)$$

$$\frac{e_1 \Downarrow \text{num}[n]}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow e_2} n \neq 0\ (tt)$$

$$(\lambda\, x.\, (\lambda\, y.\, \text{if } x + y \text{ then } x \text{ else } y))(\text{num}[1])(\text{num}[-1])$$

- Five minutes (or more?)
- Write a derivation for this tree (by hand, in your favorite text editor, Latex...)
- What is the result? Which rules do you apply (write down!)

Intermission – Summary!

- Evaluation semantics define transition systems
- Really, we are defining a relation (or function) called $\Downarrow$
- This is a **specification** of the language
- It can map one-to-one to an implementation
- And that is what we do now!

TAL in Scala

$$
\begin{aligned}
e ::= \;& \mathrm{num}[n] \\
\mid\;& x \\
\mid\;& e + e \\
\mid\;& \text{if } e \text{ then } e \text{ else } e \\
\mid\;& \lambda\, x.\, e \\
\mid\;& e(e)
\end{aligned}
$$

```scala
enum E:
  case Num(value : Int)
  case Var(name : String)
  case Add(lhs : E, rhs : E)
  case If (cond : E, tt : E, ff : E)
  case Fun(param : String, body : E)
  case App(fun : E, arg : E)
```

Evaluating TAL in Scala

```
def eval (e : E) : E = e match
    case Add(lhs, rhs) =>
      (eval(lhs), eval(rhs)) match
        case (Num(n), Num(m)) => Num(n + m)
        case _ => throw Exception("Expected number")
    case If(cond, tt, ff) =>
      eval(cond) match
        case Num(0) => eval(ff)
        case Num(_) => eval(tt)
        case _ => throw Exception("Expected number")
    case App(fun, arg) =>
      eval(fun) match
        case Fun(param, body) => eval(subst(arg, param, body))
        case _ => throw Exception("Expected fun")
    case e => e
```

$$\frac{e_1 \Downarrow \mathrm{num}[n] \quad e_2 \Downarrow \mathrm{num}[m]}{e_1 + e_2 \Downarrow \mathrm{num}[n+m]}\ (add)$$

$$\frac{e_1 \Downarrow \mathrm{num}[0]}{\mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 \Downarrow e_3}\ (ff)$$

$$\frac{e_1 \Downarrow \mathrm{num}[n]}{\mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 \Downarrow e_2}\ n \neq 0\ (tt)$$

$$\frac{e_0 \Downarrow \mathrm{num}[n]}{\lambda\, x.\, e(e_0) \Downarrow e[x/\mathrm{num}[n]]}\ (app)$$

An alternative to throwing?

Extending TAL with division (and exceptions)

$$e ::= \ldots \mid e \,/\, e \mid \mathrm{fail!}$$

Signals a program error .

$$\frac{e_1 \Downarrow \mathbf{num}[n] \quad e_2 \Downarrow \mathbf{num}[d]}{e_1/e_2 \Downarrow \mathbf{num}[\frac{n}{d}]} \; d \neq 0 \quad (div)$$

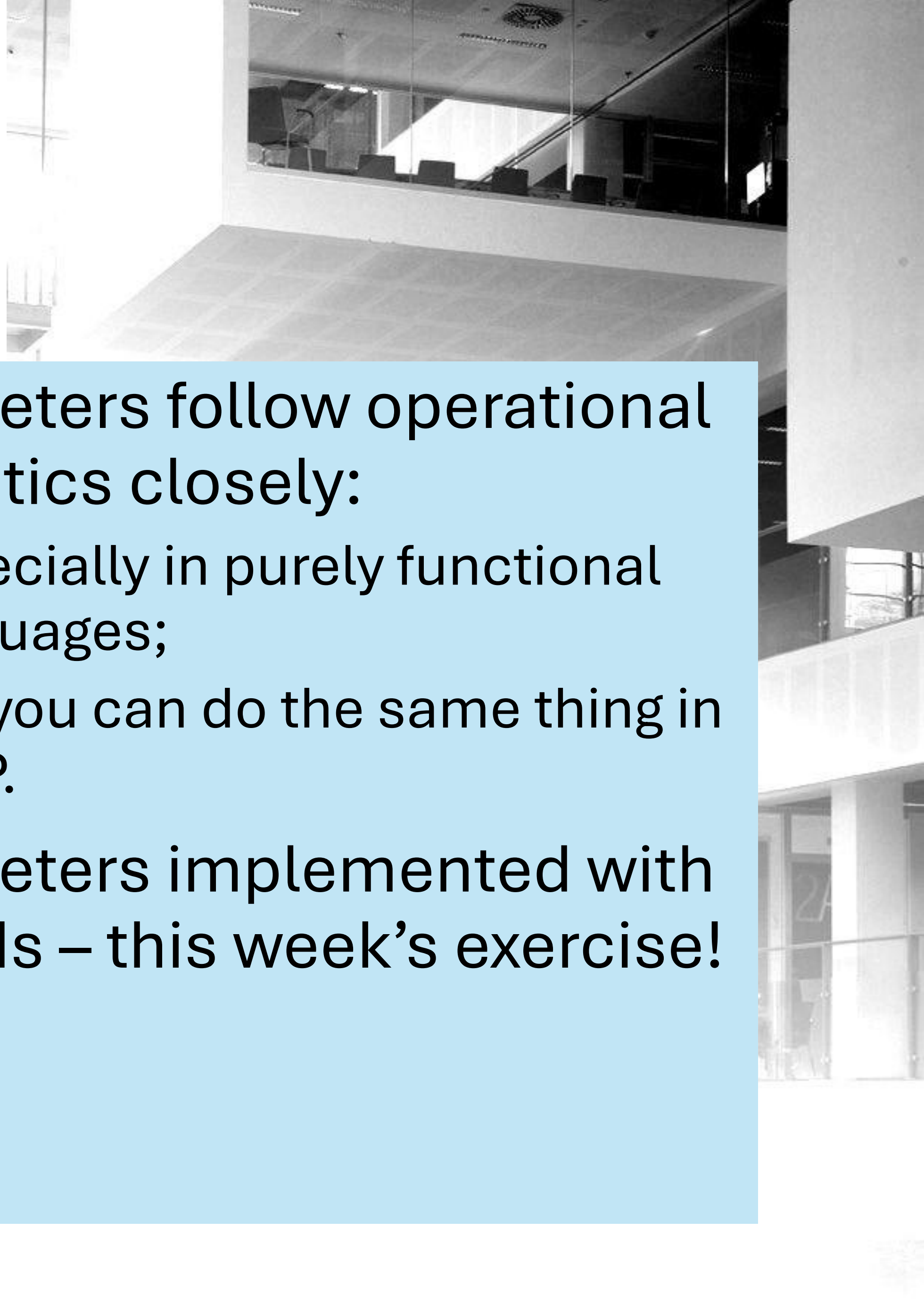
$$\frac{e_2 \Downarrow \mathbf{num}[0]}{e_1/e_2 \Downarrow \mathrm{fail!}} \quad (div_0)$$

Can we recover from an exception?

TAL with exceptions: finally some monads

```
def eval (e : E) : M[E] = e match
    ...
    case Div(lhs, rhs) =>
        eval(lhs).flatMap(lhs =>
          eval(rhs).flatMap(rhs =>
            (lhs, rhs) match
              case (Num(n), Num(0)) => M.fail ()
              case (Num(n), Num(d)) => M.unit (Num(n/d))
              case _ => throw Exception("Expected number")))
```

Summary



- Operational semantics describe:
  - what a program evaluates to;
  - not how it evaluates!
- Very research-y, but lots of practical applications, too.
- You will see this again in other courses.

- Interpreters follow operational semantics closely:
  - especially in purely functional languages;
  - but you can do the same thing in OOP.
- Interpreters implemented with monads – this week's exercise!