@AndrzejWasowski@scholar.social
Andrzej Wąsowski

Advanced Programming

Monadic Evaluators. Operational Semantics

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

What language is this?

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

This is valid in **Scala** and **Python** (and many others).

What is the outcome?

Scala: 0

Python: 0.5

Same syntax, but **different meaning**
This is why we have to talk about **semantics** of programming languages.

What is a programming language?

- **Syntax:** What may be recognized as a valid program?
- **Semantics:** How is a valid program executed?
- **Implementation:** How is the execution realized on a computer? (compiler, interpreter)

Operational Semantics with Inference Rules

Semantics rules are often present in form of binary **inference rules**

$$\frac{Premise}{Conclusion}$$

Premise (an antecedent) is what we assume. If the premise holds, then we can conclude the **conclusion** (a consequent).

If there are several premises, we separate them by spaces,
and require that all of them hold (conjunction).

Inference rules in execution: if we can execute a smaller part of the program according to the premise, then we can execute the bigger part of the program according to the conclusion.

Example: Basic evaluator (1/2) [Sect. 2.1 in Wadler]

Abstract Syntax

First: define the **abstract syntax** (syntax after parsing, already represented as trees). Abstract syntax is often specified using an ambiguous **context-free grammar**.

$$
\begin{aligned}
e ::= {} & \mathsf{num}(n) \quad n \in \mathbb{Z} \\
| {} & e \div e
\end{aligned}
$$

In Scala this becomes:

```
enum NumExpr
  case Num(n: Int)
  case Div(left: NumExpr, right: NumExpr)
```

The grammar expressions are often called **terms**. Wadler uses this word.

Example: Basic Evaluator (2/2)

Big-step Operational Semantics

In operational semantics (reduction semantics) we specify premises and conclusions using a reduction relation (judgement) from terms to values (big-step).

$$\frac{}{\textsf{num}(n) \to n}\ \textsf{NUM}$$

$$\frac{e \to n_1 \quad e' \to n_2}{e \div e' \to n_1 / n_2}\ \textsf{DIV}$$

If the premise is **empty**, the conclusion **always holds** (true premise, axiom rule).

In Scala:

```scala
def eval (expr: NumExpr): Int = expr match
  case Num(n) => n
  case Div(left, right) => eval(left) / eval(right)
```

This is so called **big-step semantics**.
Each evaluation relation (arrow) performs a **complete reduction from expression to value**.

Evaluation is Derivation

Example

Consider an expression: $(\text{num}(42) \div \text{num}(21)) \div \text{num}(2)$.

In Scala `Div( Div(Num(42),Num(21)), Num(2))`.

NB. **Precedence** and left/right-**binding** (order of evaluation) is decided by the **parser**, before the evaluator starts.

The derivation tree (read from bottom):

$$\dfrac{\dfrac{\dfrac{}{\text{num}(42) \to 42}\,\text{NUM} \qquad \dfrac{}{\text{num}(21) \to 21}\,\text{NUM}}{\text{num}(42) \div \text{num}(21) \to 2}\,\text{DIV} \qquad \dfrac{}{\text{num}(2) \to 2}\,\text{NUM}}{(\text{num}(42) \div \text{num}(21)) \div \text{num}(2) \to 1}\,\text{DIV}$$

Names are written to the right of the rules.
The tree (bottom-up) follows the recursion of our interpreter of the previous slide.

Let's add exceptions

Wadler, Section 2.2

The above semantics is **broken**. We should have really written

$$\frac{e \to n_1 \quad e' \to n_2 \quad n_2 \neq 0}{e \div e' \to n_1/n_2}\ \text{DIV-NORM}$$

For $n_2 = 0$ the program **gets stuck**. The further execution is not specified.
Such semantics would not give meanings to programs with division by zero.

Alternatively, **add exceptions** to the language. We keep the same abstract syntax

$$\begin{aligned} e ::= {} & \mathsf{num}(n) \quad n \in \mathbb{Z} \\ \mid {} & e \div e \end{aligned}$$

but change the evaluation rule (next slide).

Exceptions: The new evaluation rule

The type: `eval`: $e \to \textit{Exception} \oplus \textit{Return}\ \mathbb{Z}$

In Scala:

```scala
def eval (e: NumExpr): M[Int] = e match

case Num(n) => Return(n)
case Div(left, right) => eval(left) match
  case Raise(msg) => Raise(msg)
  case Return(lv) => eval(right) match
    case Raise(msg) => Raise(msg)
    case Return(rv) =>
      if rv == 0
        then Raise("Division by zero")
        else Return(lv/rv)
```

where

```scala
enum M[A]
  case Return (a: A)
  case Raise (msg: String)
```

$$\frac{e \to \text{Return}\ n_1 \qquad e' \to \text{Return}\ n_2 \qquad n_2 = 0}{e \div e' \to \text{Exception}\ \texttt{"Division by zero"}}\ \text{DIV-EXC}$$

$$\frac{e \to \text{Return}\ n_1 \qquad e' \to \text{Return}\ n_2 \qquad n_2 \neq 0}{e \div e' \to \text{Return}\ (n_1/n_2)}\ \text{DIV-NORM}$$

$$\frac{}{\text{num}(n) \to \text{Return}(n)}\ \text{NUM}$$

$$\frac{e \to \text{Exception}\ msg}{e \div e' \to \text{Exception}\ msg}\ \text{EXC-PROP-1}$$

$$\frac{e' \to \text{Exception}\ msg}{e \div e' \to \text{Exception}\ msg}\ \text{EXC-PROP-2}$$

- Note the correspondence between operational semantics and the implementation
- Exercise: write the operational rules for other interpreters in Sect. 2.2 (not shown in slides)

The basic evaluator vs the exception evaluator

```
1 def eval (expr: NumExpr): Int = expr match
2   case Num(n) => n
3   case Div(left, right) => eval(left) / eval(right)
```

```
1 def eval (e: NumExpr): M[Int] = e match
2 case Num(n) => Return(n)
3 case Div(left, right) => eval(left) match
4   case Raise(msg) => Raise(msg)
5   case Return(lv) => eval(right) match
6     case Raise(msg) => Raise(msg)
7     case Return(rv) =>
8       if rv == 0
9         then Raise("Division by zero")
10        else Return(lv/rv)
```

- **Problem:** They are widely different!
- In an **imperative language**, only line 3 of the first one would require a modification that looks similar to lines 7–9 of the second one
- We want functional programs to be as **maintainable** as imperative ones

Can we make interpreters modular?

Compare our two evaluators

- Wadler shows several interpreters for the same language
- Each interpreter has a different type
- Each interpreter has a different implementation
- Can we factor out the common parts?

The **common parts** are:

- The abstract syntax
- The structure of the interpreter (the recursion over the syntax tree)

The **different parts** are:

- The return type of the interpreter
- The handling of special cases (exceptions, state, output, etc)

Let's make `M` a monad

This, as we know requires two functions:

`unit`: $a \to M[a]$ and
`flatMap`: $M[a] \to (a \to M[b]) \to M[b]$ (called $\star$ by Wadler)

In Scala these become (for the exception monad):

```
def unit[A](a: A): M[A] = Return(a)
def flatMap[A,B](ma: M[A])(f: A => M[B]): M[B] = ma match
  case Raise(msg) => Raise(msg)
  case Return(a) => f(a)
```

We also need to ensure that the **monad laws** hold (left identity, right identity, associativity). See exercises.

How did we come up with this? Exception monad is the same thing as Option/Either in the previous lectures.

Implement the interpreter in Monad `M`

```scala
def eval (e: NumExpr): M[Int] = e match
case Num(n) => Return(n)
case Div(left, right) => eval(left) match
  case Raise(msg) => Raise(msg)
  case Return(lv) => eval(right) match
    case Raise(msg) => Raise(msg)
    case Return(rv) =>
      if rv == 0
        then Raise("Division by zero")
        else Return(lv/rv)
```

```scala
def eval2(e: NumExpr): M[Int] = e match
  case Num(n) => unit(n)
  case Div(left, right) =>
    flatMap(eval2(left)) { lv =>
      flatMap(eval2(right)) { rv =>
        if rv == 0
          then Raise("Division by zero")
          else unit(lv / rv)
      }
    }
```

```scala
def eval3(e: NumExpr): M[Int] = e match
  case Num(n) => unit(n)
  case Div(left, right) => for
    lv <- eval3(left)
    rv <- eval3(right)
    result <- if rv == 0
      then Raise("Division by zero")
      else unit(lv / rv)
  yield result
```

Note: Pattern matching on results is gone. This is now “done by the monad.”

Identity Monad: Make the basic interpreter monadic

```scala
type M[A] = A
def unit[A](a: A): M[A] = a
def flatMap[A,B](ma: M[A])(f: A => M[B]): M = f(ma)
def eval2(e: NumExpr): M[Int] = e match
  case Num(n) => unit(n)
  case Div(left, right) =>
    flatMap(eval2(left)) { lv =>
      flatMap(eval2(right)) { rv =>
        unit(lv / rv)
      }
    }
```

```scala
def eval3(e: NumExpr): M[Int] = e match
  case Num(n) => unit(n)
  case Div(left, right) => for
    lv <- eval3(left)
    rv <- eval3(right)
    result <- unit(lv / rv)
  yield result
```

For comparison from the previous slide (with exception handling):

```scala
def eval3(e: NumExpr): M[Int] = e match
  case Num(n) => unit(n)
  case Div(left, right) => for
    lv <- eval3(left)
    rv <- eval3(right)
    result <- if rv == 0
      then Raise("Division by zero")
      else unit(lv / rv)
  yield result
```

The **change required** to switch between the two interpreters is **minimal**, like in the imperative language: change the return type and the calculation of the special case in the monad.

In the next episode . . .

- **Reinforcement learning!** We do ML&AI in style (pure functional style!)
- Write an end-to-end pure functional program
- Reading provided via our github repository
- **Happy reading! and See you next week!**