Principles of Programming Languages

Lecture 01 Introduction

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Peter Thiemann

University of Freiburg, Germany

thiemann@informatik.uni-freiburg.de

16 Apr 2018

Contents of the course

- Building blocks of progamming languages
- Vernacular for talking about programming languages
- Tools for describing the meaning of a program
- Techniques for reasoning about a program

Why bother?

- Improved understanding of programs
- Verify program transformations
- Verify compilers
- Design and verify static analyses

Key aspect: Semantics

Dynamics — run time

- describes execution of a program
- wide variety of styles

Statics — compile time

- describes checks **before** execution
- conditions that avoid certain execution errors
- principal example: types

Requirements

UNI FREIBURG

Description of

- syntax
- execution states
- evolving execution
- static rules

Checking that

- execution preserves static rules
- static rules enable execution

Plan

Concrete syntax

- **Concrete syntax** describes valid program texts
- Description in two stages
  1. lexical syntax
  2. context-free syntax

Lexical syntax

The lexical syntax defines the “atoms” of the language in terms of regular languages. The *lexical analysis* (*scanner*, *lexer*) of a compiler partitions a program into *lexemes* and maps them into *tokens*. (Lexemes are sequences of input characters, tokens are symbolic values.)

Lexical syntax, example

Typical lexeme classes are identifiers, numeric literals, opening and closing parentheses, and keywords

class	regexp	example	token
identifier	`[A-Za-z][A-Za-z0-9]*`	`Catch22`	*ident*(`Catch22`)
numeric lit	`[-+]?[0-9]+`	`-42`	`num (42)`
opening par	`(`	`(`	*openingPar*
closing par	`)`	`)`	*closingPar*
keyword	`while`	`while`	*kwWhile*

The scanner typically ignores *whitespace*, that is sequences of spaces, tabulators, line feeds, and so on. The scanner also removes comments.

Context-free Syntax

- given by a context-free grammar $\mathcal{G}$
- symbols are tokens from lexical analysis
- a *parser* for $\mathcal{G}$ maps a token sequence to a derivation tree of $\mathcal{G}$ or fails if the token sequence is not in the language.

Context-free syntax, example

A grammar for parsing infix expressions.

$$
\begin{array}{lcl}
\langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{factor} \rangle \\
\langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{expr} \rangle - \langle \mathit{factor} \rangle \\
\langle \mathit{factor} \rangle & \rightarrow & \langle \mathit{atom} \rangle \\
\langle \mathit{factor} \rangle & \rightarrow & \langle \mathit{factor} \rangle / \langle \mathit{atom} \rangle \\
\langle \mathit{atom} \rangle & \rightarrow & \mathtt{a} \\
\langle \mathit{atom} \rangle & \rightarrow & (\langle \mathit{expr} \rangle)
\end{array}
$$

It reflects the convention that / binds tighter than - and that both associate to the left.

Context-free syntax, derivation tree

Derivation tree for `a / a - (a - a / a)`.
TODO

Abstract syntax (AST)

Much of the structure of the derivation tree of a grammar suitable for parsing is irrelevant for the meaning of an expression. For that task, a much simpler structure is sufficient, the *abstract syntax*:

$$e \quad ::= \quad e{-}e \mid e/e \mid \mathtt{a}$$

- Abstract syntax is also described by a context-free grammar
- The point of this grammar is *not* the set of strings derivable from it, but rather its set of derivation trees, the *abstract syntax trees (AST)*.

Describing AST

More precisely, an AST is a term built from a signature of operation symbols (the above grammar is a common, but sloppy way of writing that signature). Technically, the non-terminals of the grammar are considered as types and an explicit signature specifies the restrictions.

$$
\begin{array}{lcrcl}
- & : & \langle \mathit{expr} \rangle \times \langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{expr} \rangle \\
/ & : & \langle \mathit{expr} \rangle \times \langle \mathit{expr} \rangle & \rightarrow & \langle \mathit{expr} \rangle \\
\mathtt{a} & : & & \rightarrow & \langle \mathit{expr} \rangle
\end{array}
$$

Language support for AST

Functional programming languages directly support tree datatypes suitable for defining AST. For example, the type `expr` can be defined as follows in OCaml:

```
type expr = subExpr of expr * expr
          | divExpr of expr * expr
          | conExpr
```

(The lexemes / and - cannot be used because they are predefined by the OCaml language. Constructors like `subExpr`, `divExpr`, `conExpr` must be used instead.)

Language support for AST

PLT Redex is a domain specific language for *semantics engineering*. It provides extensive support for most constructions used in this course.

```
(define-language expressions
  (E ::= (- E E)
         (/ E E)
         number))
```

There are very few restrictions on lexemes in PLT Redex.

Plan

Semantics

- assign meaning to a program text
- usually by a mapping from AST to mathematical object
- different styles of semantics $\Leftrightarrow$ different mathematical objects

denotational: defines domains that capture final result of a program; the object is an element of a suitable domain

operational: defines an abstract machine which comes with a notion of (step-wise) execution; the object is a state of this machine

axiomatic: defines the meaning via logical formulas expressing pre- and postconditions; the object is a pair of pre- and postconditions

Operational Semantics

This Course

In this course, we concentrate on operational semantics because of its simple foundations.

Two styles of operational semantics

Small-step operational semantics
: describes program execution as a sequence of transformations starting from the initial state and ending with the final result (if any).

Big-step operational semantics
: describes program execution as a function from program text to the result. Often close to an interpreter.

Small-step Operational Semantics
Example

$$e ::= e - e \mid e / e \mid n$$

where $n \in \mathbf{Q}$ ranges over rational numbers.
Small-step operational semantics is specified as an abstract machine:

- **state** expression e
- **transformation** given by a *transition relation* $e \longrightarrow e'$, a binary relation on expressions
- **final state** number n: particular expressions that denote final results, often called *values*

-X

$$m\text{-}n \longrightarrow p \quad \text{where } p = m - n \in rat$$

/X

$$m/n \longrightarrow p \quad \text{where } n \neq 0 \text{ and } p = m/n \in rat$$

-L

$$\frac{e_1 \longrightarrow e_1'}{e_1\text{-}e_2 \longrightarrow e_1'\text{-}e_2}$$

-R

$$\frac{e_2 \longrightarrow e_2'}{e_1\text{-}e_2 \longrightarrow e_1\text{-}e_2'}$$

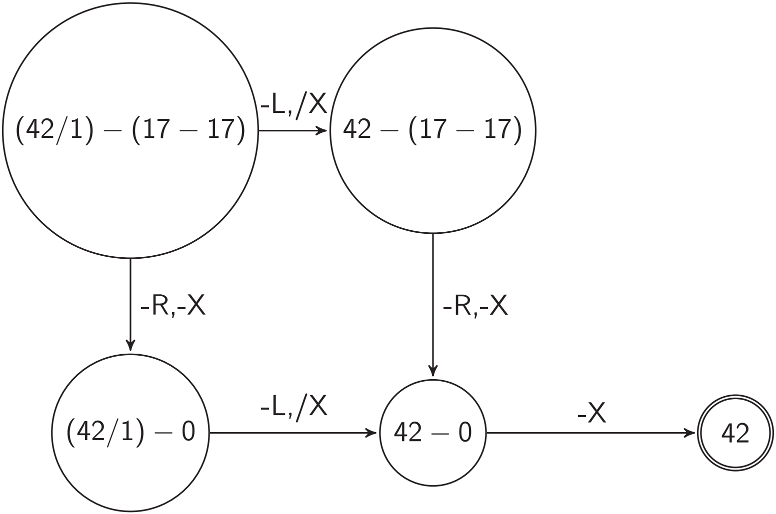
/L

$$\frac{e_1 \longrightarrow e_1'}{e_1/e_2 \longrightarrow e_1'/e_2}$$

/R

$$\frac{e_2 \longrightarrow e_2'}{e_1/e_2 \longrightarrow e_1/e_2'}$$

Small-step Operational Semantics
Example execution



Small-step Operational Semantics
Why translation **relation**?

UNI FREIBURG

Partial operations

There may be no useful transition from a (non-final) state. For example, $1/0$ has no transition, but it is not a number.

Nondeterminism

Occasionally the order in which arguments are evaluated does not matter.

Concurrency

It is the point of modeling concurrent execution that any thread can take the next step.

Small-step Operational Semantics

Deterministic rule set for evaluation of expressions

-X
$$m - n \longrightarrow p \quad \text{where } p = m - n \in \mathbf{Q}$$

/X
$$m / n \longrightarrow p \quad \text{where } n \neq 0 \text{ and } p = m / n \in \mathbf{Q}$$

-L
$$\frac{e_1 \longrightarrow e_1'}{e_1 - e_2 \longrightarrow e_1' - e_2}$$

-R
$$\frac{e_2 \longrightarrow e_2'}{m - e_2 \longrightarrow m - e_2'}$$

/L
$$\frac{e_1 \longrightarrow e_1'}{e_1 / e_2 \longrightarrow e_1' / e_2}$$

/R
$$\frac{e_2 \longrightarrow e_2'}{m / e_2 \longrightarrow m / e_2'}$$

Goal

Relate an expression to its final value.

- Need to specify binary relation, called *evaluation*, between expressions and numbers $e \hookrightarrow n$.
- Evaluation may be partial as for $1/0$
- Evaluation is usually deterministic (i.e. a partial function)

$$n \hookrightarrow n$$

$$\frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow n_2}{e_1\texttt{-}e_2 \hookrightarrow m} \quad \text{where } m = n_1 - n_2 \in \mathbf{Q}$$

$$\frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow n_2}{e_1/e_2 \hookrightarrow m} \quad \text{where } n_2 \neq 0 \text{ and } m = n_1/n_2 \in \mathbf{Q}$$

Big-step Operational Semantics
Example evaluation

$$\dfrac{\dfrac{42 \hookrightarrow 42 \qquad 1 \hookrightarrow 1}{42/1 \hookrightarrow 42} \qquad \dfrac{17 \hookrightarrow 17 \qquad 17 \hookrightarrow 17}{17 - 17 \hookrightarrow 0}}{(42/1) - (17 - 17) \hookrightarrow 42}$$

Comparison

Nontermination observable in small-step; non-obvious solution for big-step

Exceptions small-step gets stuck on exceptional subexpressions like $1/0$; big-step evaluation of every expression containing $1/0$ is *undefined*

Evaluation order reasonably easy to define in small-step; more involved for big-step

Concurrency easy in small-step; non-obvious solution for big-step

But

- big-step is close to an interpreter (i.e., more intuitive, close to implementation)
- some properties are easier to prove for big-step