Programming Language Theory

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
http://www.softlang.org/

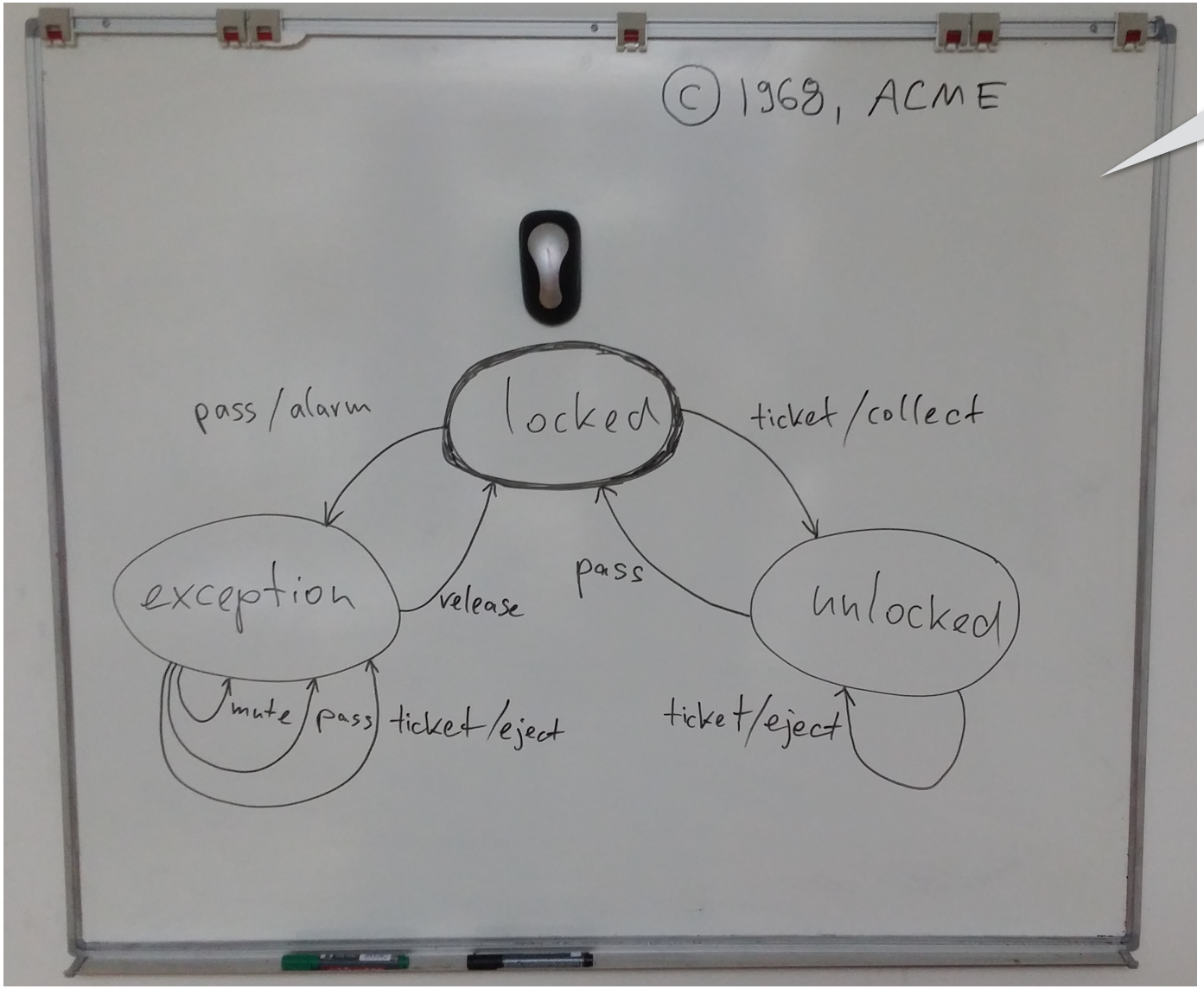
Key questions

- Syntax
  - What is the structure of language elements?
  - What *constructs* does a language provide?
  - How are these constructs organized in *categories*?
- **Semantics and types**
  - **What is the *meaning* of language elements?**
  - **What are the *types* of language phrases?**

What sorts of languages are of interest?

- General purpose programming languages
  - Functional languages
  - Imperative languages
  - …
- Domain-specific languages
  - Finite state machine language
  - …

A DSL for finite state machines (FSMs)



A FSM for a 'turnstile' in a metro system

What's the semantics of FSM? When is an FSM well-formed?

An imperative program for Euclidian division

```
{
  // Sample operands for Euclidian division
  x = 14;
  y = 4;

  // Compute quotient q=3 and remainder r=2
  q = 0;
  r = x;
  while (r >= y) {
    r = r - y;
    q = q + 1;
  }
}
```

What's the semantics of the program?
Is it well-typed?

A functional program for “!”

```
-- The factorial function
factorial :: Int -> Int
factorial x =
  if ((==) x 0)
    then 1
    else ((*) x (factorial ((-) x 1)))

-- Apply the function to 5
main = print $ factorial 5 -- Prints 120
```

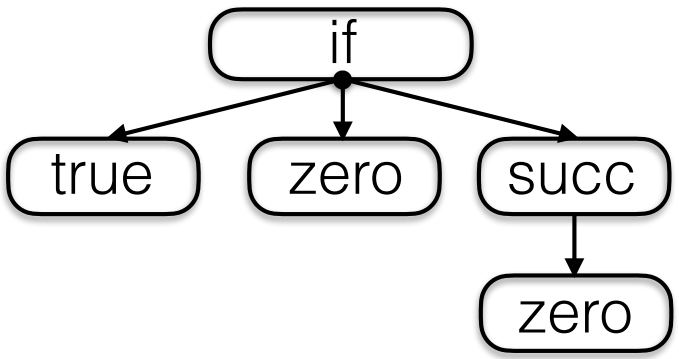
What’s the semantics of the program?
Is it well-typed?

Lectures

- **Tree-based abstract syntax**
- Basics of interpretation
- Big-step operational semantics
- Small-step operational semantics
- Type systems
- Denotational semantics
- Abstract interpretation
- Lambda calculus
- System F

Textual versus tree-based abstract syntax of a conditional expression

Concrete syntax	***Abstract syntax***	
Text	**Tree**	**Tree as term**

if true
then zero
else succ zero
		`if(true,`
		`zero,`
		`succ(zero)).`

N.B.: abstract syntax is relevant in programming language theory as foundation for defining semantics of programs and type systems.

Lectures

- Tree-based abstract syntax
- **Basics of interpretation**
- Big-step operational semantics
- Small-step operational semantics
- Type systems
- Denotational semantics
- Abstract interpretation
- Lambda calculus
- System F

Application of an interpreter of a simple imperative programming language

```
-- // Compute quotient q and remainder r for dividing x by y
-- q = 0; r = x; while (r >= y) { r = r - y; q = q + 1; }
euclideanDiv :: Stmt
euclideanDiv =
  Seq (Assign "q" (IntConst 0)) (Seq (Assign "r" (Var "x"))
      (While
        (Binary Geq (Var "r") (Var "y"))
        (Seq (Assign "r" (Binary Sub (Var "r") (Var "y")))
             (Assign "q" (Binary Add (Var "q") (IntConst 1))))))
```

```
▶ execute euclideanDivision (fromList [("x", Left 13), ("y", Left 4)])
fromList [("q",Left 3),("r",Left 2),("x",Left 14),("y",Left 4)]
```

A Haskell-based interpreter

```
-- Results of expression evaluation
type Value = Either Int Bool
-- Stores as maps from variable names to values
type Store = Map String Value
```

```
-- Execution of statements
execute :: Stmt → Store → Store
execute Skip m = m
execute (Assign x e) m = insert x (evaluate e m) m
execute (Seq s1 s2) m = execute s2 (execute s1 m)
execute (If e s1 s2) m = execute (if b then s1 else s2) m where Right b = evaluate e m
execute (While e s) m = execute (If e (Seq s (While e s)) Skip) m

-- Evaluation of expressions
evaluate :: Expr → Store → Value
evaluate (IntConst i) _ = Left i
evaluate (Var x) m = m!x
evaluate (Unary o e) m = uop o (evaluate e m)
evaluate (Binary o e1 e2) m = bop o (evaluate e1 m) (evaluate e2 m)

-- Interpretation of unary operators
uop :: UOp → Value → Value
uop Negate (Left i) = Left (negate i)
uop Not (Right b) = Right (not b)
```

Lectures

- Tree-based abstract syntax
- Basics of interpretation
- **Big-step operational semantics**
- Small-step operational semantics
- Type systems
- Denotational semantics
- Abstract interpretation
- Lambda calculus
- System F

Big-step operational semantics of a simple imperative language

$$m \vdash \mathsf{skip} \longrightarrow m \qquad \text{[SKIP]}$$

$$\frac{m \vdash e \longrightarrow v}{m \vdash \mathsf{assign}(x, e) \longrightarrow m[x \mapsto v]} \qquad \text{[ASSIGN]}$$

$$\frac{m_0 \vdash s_1 \longrightarrow m_1 \qquad m_1 \vdash s_2 \longrightarrow m_2}{m_0 \vdash \mathsf{seq}(s_1, s_2) \longrightarrow m_2} \qquad \text{[SEQ]}$$

$$\frac{m \vdash e_0 \longrightarrow \mathsf{true} \qquad m \vdash s_1 \longrightarrow m'}{m \vdash \mathsf{if}(e_0, s_1, s_2) \longrightarrow m'} \qquad \text{[IF1]}$$

$$\frac{m \vdash e_0 \longrightarrow \mathsf{false} \qquad m \vdash s_2 \longrightarrow m'}{m \vdash \mathsf{if}(e_0, s_1, s_2) \longrightarrow m'} \qquad \text{[IF2]}$$

$$\frac{m \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{skip}) \longrightarrow m'}{m \vdash \mathsf{while}(e, s) \longrightarrow m'} \qquad \text{[WHILE]}$$

Lectures

- Tree-based abstract syntax
- Basics of interpretation
- Big-step operational semantics
- **Small-step operational semantics**
- Type systems
- Denotational semantics
- Abstract interpretation
- Lambda calculus
- System F

Small-step operational semantics of a simple imperative language

$$\frac{m \vdash e \longrightarrow v}{\langle m, \mathsf{assign}(x, e)\rangle \rightarrow \langle m[x \mapsto v], \mathsf{skip}\rangle} \quad [\mathsf{assign}]$$

$$\langle m, \mathsf{seq}(\mathsf{skip}, s)\rangle \rightarrow \langle m, s\rangle \quad [\mathsf{seq1}]$$

$$\frac{\langle m, s_1\rangle \rightarrow \langle m', s_1'\rangle}{\langle m, \mathsf{seq}(s_1, s_2)\rangle \rightarrow \langle m', \mathsf{seq}(s_1', s_2)\rangle} \quad [\mathsf{seq2}]$$

$$\frac{m \vdash e_0 \longrightarrow \mathsf{true}}{\langle m, \mathsf{if}(e_0, s_1, s_2)\rangle \rightarrow \langle m, s_1\rangle} \quad [\mathsf{if1}]$$

$$\frac{m \vdash e_0 \longrightarrow \mathsf{false}}{\langle m, \mathsf{if}(e_0, s_1, s_2)\rangle \rightarrow \langle m, s_2\rangle} \quad [\mathsf{if2}]$$

$$\langle m, \mathsf{while}(e, s)\rangle \rightarrow \langle m, \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s))), \mathsf{skip})\rangle \quad [\mathsf{while}]$$

Lectures

- Tree-based abstract syntax
- Basics of interpretation
- Big-step operational semantics
- Small-step operational semantics
- **Type systems**
- Denotational semantics
- Abstract interpretation
- Lambda calculus
- System F

Well-typed versus ill-typed programs

```
▶ typeOf (Succ Zero)
Just NatType
```

The expression **Succ Zero** is of type **NatType**.

- -

```
▶ typeOf (Succ TRUE)
Nothing
```

The expression **Succ TRUE** is ill-typed.

N.B.: We hint here at how we plan to implement a type system effectively as a type checker. This is comparable to implementing an operational semantics definition as an interpreter.

Lectures

- Tree-based abstract syntax
- Basics of interpretation
- Big-step operational semantics
- Small-step operational semantics
- Type systems
- **Denotational semantics**
- Abstract interpretation
- Lambda calculus
- System F

Denotational semantics = <u>functional</u> semantics

Semantic domains = <u>function</u> types of meanings

$$storeT = store \nrightarrow store \;\; // \text{ Type of store transformation}$$
$$storeO = store \nrightarrow value \;\; // \text{ Type of store observation}$$

Semantic <u>functions</u> = mappings from syntax to semantcs

$$\mathcal{S} \;\; : stmt \rightarrow storeT \;\; // \text{ Semantics of statements}$$
$$\mathcal{E} \;\; : expr \rightarrow storeO \;\; // \text{ Semantics of expressions}$$

N.B.: The semantic functions are to be defined **compositionally**.

Denotational semantics obeys ***compositionality***:

define meaning of compound construct in terms of the meanings of constituents without reference to syntax.

For comparison:

Big-step operational semantics of imperative programs

$$\frac{m_0 \vdash s_1 \longrightarrow m_1 \qquad m_1 \vdash s_2 \longrightarrow m_2}{m_0 \vdash \mathsf{seq}(s_1, s_2) \longrightarrow m_2} \quad [\text{SEQ}]$$

$$\frac{m \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{skip}) \longrightarrow m'}{m \vdash \mathsf{while}(e, s) \longrightarrow m'} \quad [\text{WHILE}]$$

N.B.: [SEQ] is compositional, but [WHILE] is <u>not</u>, as the meaning of a while-loop is defined here in terms of a constructed phrase (which, by the way, contains the while-loop under definition).

Lectures

- Tree-based abstract syntax
- Basics of interpretation
- Big-step operational semantics
- Small-step operational semantics
- Type systems
- Denotational semantics
- **Abstract interpretation**
- Lambda calculus
- System F

Abstract interpretation

```
{
    ...
    y = x * x + 42;
    if (y < 0)
        y = −y;
    ...
}
```

A program that can be optimized.

Abstract semantic domains involved in optimization

$*$	Neg	Zero	Pos	?
Neg	Pos	Zero	Neg	?
Zero	Zero	Zero	Zero	Zero
Pos	Neg	Zero	Pos	?
?	?	Zero	?	?

$+$	Neg	Zero	Pos	?
Neg	Neg	Neg	?	?
Zero	Neg	Zero	Pos	?
Pos	?	Pos	Pos	?
?	?	?	?	?

$<$	Neg	Zero	Pos	?
Neg	?	True	True	?
Zero	False	False	True	?
Pos	False	False	?	?
?	?	?	?	?

Lectures

- Tree-based abstract syntax
- Basics of interpretation
- Big-step operational semantics
- Small-step operational semantics
- Type systems
- Denotational semantics
- Abstract interpretation
- **Lambda calculus**
- System F

Illustrative lambda expressions

Lambda abstraction, i.e., variables abstract over lambda expressions

// Identity function

$\lambda x.\ x$

Variable reference

// Function composition

$\lambda f\ g\ x.\ f\ (g\ x)$

Function application by juxtaposition

N.B.: The λ-calculus is essentially a very simple functional programming language.

Lectures

- Tree-based abstract syntax
- Basics of interpretation
- Big-step operational semantics
- Small-step operational semantics
- Type systems
- Denotational semantics
- Abstract interpretation
- Lambda calculus
- **System F**

Illustrative System F expressions

// Identity function

$\lambda x.\ x$

// Function composition

$\lambda f\ g\ x.\ f\ (g\ x)$

N.B.: These expressions are polymorphically typed in System F.

Programming Language Theory: *What is it good for?*

- A foundation for
  - implementing
    - compilers (code generators, optimizers),
    - refactoring tools,
    - bug finders,
    - and, in fact, pretty much any language-based technology.
  - proving program correctness
  - …

Online resources

YAS' GitHub repository contains all code.

YAS (Yet Another SLR (Software Language Repository))

http://www.softlang.org/yas

All lectures are supported by Haskell code.

The Software Languages Book

http://www.softlang.org/book

There is a part on programming language theory.