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
  - ...

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
A DSL for finite state machines (FSMs)

2 Story of a language

turnstile (hub or spider) and insert their valid ticket into the turnstile's card reader, when they want to reach the platform in a legal manner. The Acme architects and the customer agree on the basic functionality for turnstiles in a meeting, where they draw an FSM on the whiteboard.

Illustration 2.1 (A turnstile FSM in visual notation on the whiteboard).

FSML (its visual notation) is quickly explained with the example at hand. Finite state machines comprise of states (nodes) and transitions (edges). The initial state of the machine is indicated through the bolder border. There are these states in the example:

locked
The turnstile is locked. No passenger is allowed to pass.

unlocked
The turnstile is unlocked. A passenger may pass.

exception
A problem has occurred and metro personnel needs to intervene.

Each transition connects two states and is annotated by two parts $e/a$, an event $e$ and an action $a$, where the latter is optional. The event may be triggered by a user or a sensor. An event causes a transition. The action corresponds to functionality to be performed or an actor to be addressed upon transition. Transitions are directed. The source of a transition is the source state; the target of a transition is the target state. The turnstile FSM involves these events:

- pass/alarm
- ticket/collect
- release
- pass
- ticket/eject
- ticket/eject

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

{x
    // 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
• 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

<table>
<thead>
<tr>
<th>Concrete syntax</th>
<th>Abstract syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text</strong></td>
<td><strong>Tree</strong></td>
</tr>
<tr>
<td>if true</td>
<td>if(true, zero, succ(zero))</td>
</tr>
<tr>
<td>then zero</td>
<td>true zero succ</td>
</tr>
<tr>
<td>else succ zero</td>
<td></td>
</tr>
</tbody>
</table>

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

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
• 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

```haskell
-- // 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)
```
• 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

\[ m \vdash \text{skip} \rightarrow m \quad \text{[SKIP]} \]

\[ \frac{m \vdash e \rightarrow v}{m \vdash \text{assign}(x,e) \rightarrow m[x \mapsto v]} \quad \text{[ASSIGN]} \]

\[ \frac{m_0 \vdash s_1 \rightarrow m_1 \quad m_1 \vdash s_2 \rightarrow m_2}{m_0 \vdash \text{seq}(s_1,s_2) \rightarrow m_2} \quad \text{[SEQ]} \]

\[ \frac{m \vdash e_0 \rightarrow \text{true} \quad m \vdash s_1 \rightarrow m'}{m \vdash \text{if}(e_0,s_1,s_2) \rightarrow m'} \quad \text{[IF1]} \]

\[ \frac{m \vdash e_0 \rightarrow \text{false} \quad m \vdash s_2 \rightarrow m'}{m \vdash \text{if}(e_0,s_1,s_2) \rightarrow m'} \quad \text{[IF2]} \]

\[ m \vdash \text{if}(e, \text{seq}(s, \text{while}(e,s)), \text{skip}) \rightarrow m' \quad \text{[WHILE]} \]
• 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

\[
\begin{align*}
m &\vdash e \rightarrow v \\
\langle m, \text{assign}(x, e) \rangle &\rightarrow \langle m[x \leftarrow v], \text{skip} \rangle \\
\langle m, \text{seq}(\text{skip}, s) \rangle &\rightarrow \langle m, s \rangle \\
\langle m, s_1 \rangle &\rightarrow \langle m', s'_1 \rangle \\
\langle m, \text{seq}(s_1, s_2) \rangle &\rightarrow \langle m', \text{seq}(s'_1, s_2) \rangle \\
\langle m, \text{if}(e_0, s_1, s_2) \rangle &\rightarrow \langle m, s_1 \rangle \\
\langle m, \text{if}(e_0, s_1, s_2) \rangle &\rightarrow \langle m, s_2 \rangle \\
\langle m, \text{while}(e, s) \rangle &\rightarrow \langle m, \text{if}(e, \text{seq}(s, \text{while}(e, s)), \text{skip}) \rangle
\end{align*}
\]

[assign] [seq1] [seq2] [if1] [if2] [while]
• 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.
• 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
= functional semantics

Semantic domains = function types of meanings

\[ \text{store}T = \text{store} \rightarrow \text{store} \] // Type of store transformation
\[ \text{store}O = \text{store} \rightarrow \text{value} \] // Type of store observation

Semantic functions = mappings from syntax to semantics

\[ S : \text{stmt} \rightarrow \text{store}T \] // Semantics of statements
\[ \mathcal{E} : \text{expr} \rightarrow \text{store}O \] // 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

\[
\begin{align*}
    m_0 & \vdash s_1 \rightarrow m_1 & m_1 & \vdash s_2 \rightarrow m_2 \\
    m_0 & \vdash \text{seq}(s_1, s_2) \rightarrow m_2 \\
    m & \vdash \text{if}(e, \text{seq}(s, \text{while}(e, s)), \text{skip}) \rightarrow m' \\
    m & \vdash \text{while}(e, s) \rightarrow m'
\end{align*}
\]

[N.B.: [SEQ] is compositional, but [WHILE] is not, 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).]
• 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

A program that can be optimized.

Abstract semantic domains involved in optimization

<table>
<thead>
<tr>
<th>*</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Pos</td>
<td>Zero</td>
<td>Neg</td>
<td>?</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
</tr>
<tr>
<td>Pos</td>
<td>Neg</td>
<td>Zero</td>
<td>Pos</td>
<td>?</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>Zero</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Neg</td>
<td>Neg</td>
<td>Neg</td>
<td>?</td>
</tr>
<tr>
<td>Zero</td>
<td>Neg</td>
<td>Zero</td>
<td>Pos</td>
<td>?</td>
</tr>
<tr>
<td>Pos</td>
<td>?</td>
<td>Pos</td>
<td>Pos</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>?</td>
<td>True</td>
<td>True</td>
<td>?</td>
</tr>
<tr>
<td>Zero</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>?</td>
</tr>
<tr>
<td>Pos</td>
<td>False</td>
<td>False</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
• 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
\( \lambda x. x \)

// Function composition
\( \lambda f \ g \ x. \ f \ (g \ x) \)

N.B.: The \( \lambda \)-calculus is essentially a very simple functional programming language.
• 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.