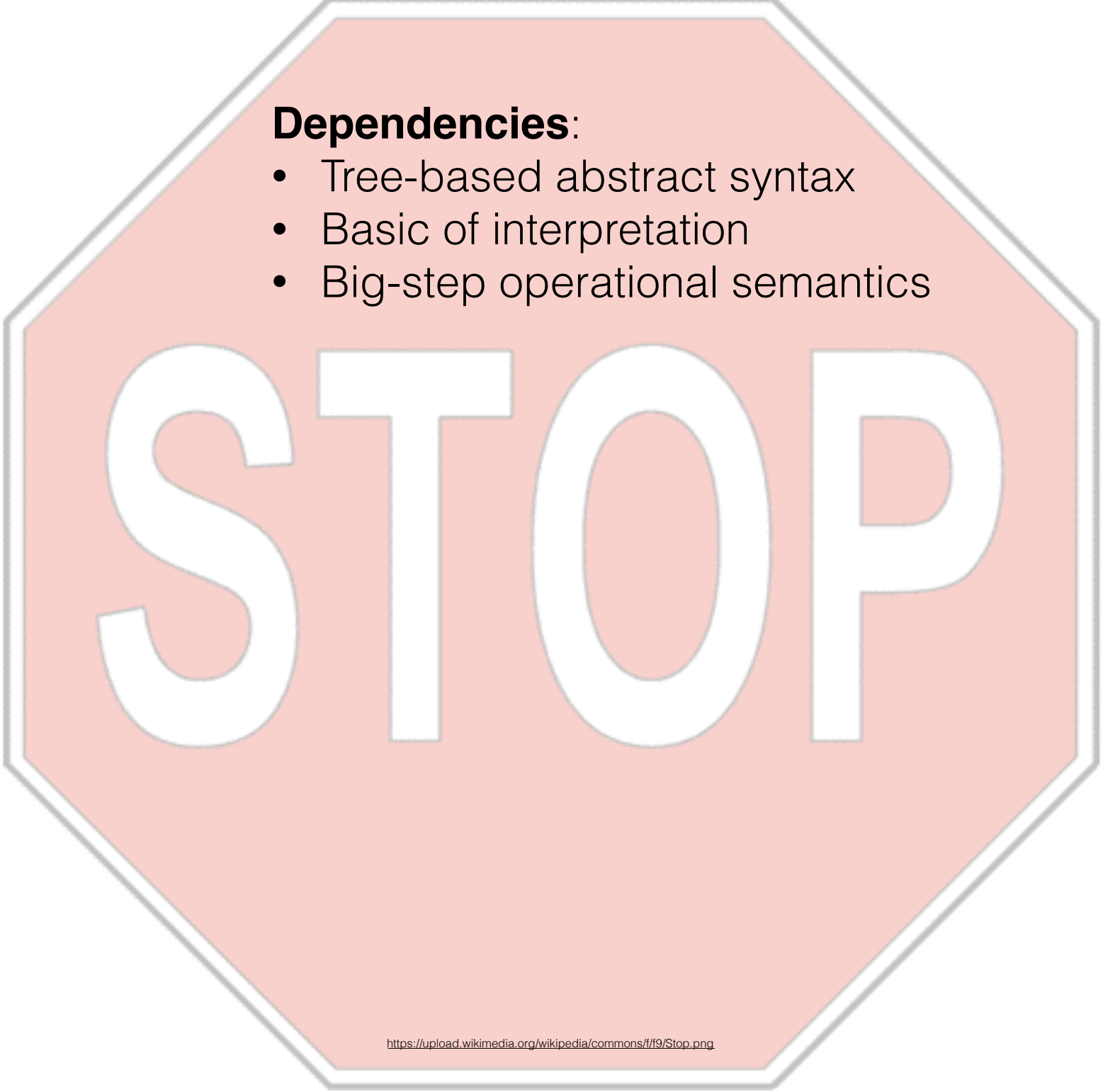
Small-step operational semantics (An introduction)

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

Dependencies:

- Tree-based abstract syntax
- Basic of interpretation
- Big-step operational semantics



https://upload.wikimedia.org/wikipedia/commons/f/f9/Stop.png

A semantics definition assigns meanings to language elements.

Abstract syntax of running example

```
symbol true : → expr ; // The Boolean "true"
symbol false : → expr ; // The Boolean "false"
symbol zero : → expr ; // The natural number zero
symbol succ : expr → expr ; // Successor of a natural number
symbol pred : expr → expr ; // Predecessor of a natural number
symbol iszero : expr → expr ; // Test for a number to be zero
symbol if : expr × expr × expr → expr ; // Conditional
```

A sample term:

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

N.B.: The expression language at hand is also referred to as **BTL** — Basic TAPL Language — where TAPL is a reference to Pierce's textbook 'Types and programming languages'.

Big-step versus small-step

- big step — $e \longrightarrow v$
- small step — $e \rightarrow e'$

(BTL)

Relation between program phrases and execution/evaluation **results**

Relation between program phrases such that one **step of computation** was completed

What is small-step step good for?

- $\text{big step} — e \longrightarrow v$
- $\text{small step} — e \rightarrow e'$

- Big-step does not handle (well) some constructs:
  - Interleaving parallelism
  - Exception throwing and handling
  - Goto
  - …
- Small-step computations are more observable and adaptive:
  - Debugging is straightforward when steps are exposed.
  - Multi-language semantics is easier to integrate with steps.

Big-step versus **small**-step

$$\text{zero} \longrightarrow \text{zero}$$

$$\frac{e \longrightarrow n}{\text{succ}(e) \longrightarrow \text{succ}(n)}$$

$$\frac{e \longrightarrow \text{zero}}{\text{pred}(e) \longrightarrow \text{zero}}$$

$$\frac{e \longrightarrow \text{succ}(n)}{\text{pred}(e) \longrightarrow n}$$

$$\frac{e \rightarrow e'}{\text{succ}(e) \rightarrow \text{succ}(e')}$$

$$\frac{e \rightarrow e'}{\text{pred}(e) \rightarrow \text{pred}(e')}$$

$$\text{pred}(\text{zero}) \rightarrow \text{zero}$$

$$\text{pred}(\text{succ}(n)) \rightarrow n$$

N.B.: There is no rule for **zero** because no step can be performed for this expression. The first rule for **pred** makes a step with the argument.

All (inference) rules for BTL's small-step semantics

$$\frac{e \rightarrow e'}{\mathsf{succ}(e) \rightarrow \mathsf{succ}(e')} \quad [\mathsf{succ}]$$

$$\frac{e \rightarrow e'}{\mathsf{pred}(e) \rightarrow \mathsf{pred}(e')} \quad [\mathsf{pred1}]$$

$$\mathsf{pred}(\mathsf{zero}) \rightarrow \mathsf{zero} \quad [\mathsf{pred2}]$$

$$\mathsf{pred}(\mathsf{succ}(n)) \rightarrow n \quad [\mathsf{pred3}]$$

as seen earlier

$$\frac{e \rightarrow e'}{\mathsf{iszero}(e) \rightarrow \mathsf{iszero}(e')} \quad [\mathsf{iszero1}]$$

$$\mathsf{iszero}(\mathsf{zero}) \rightarrow \mathsf{true} \quad [\mathsf{iszero2}]$$

$$\mathsf{iszero}(\mathsf{succ}(n)) \rightarrow \mathsf{false} \quad [\mathsf{iszero3}]$$

$$\frac{e_0 \rightarrow e_0'}{\mathsf{if}(e_0, e_1, e_2) \rightarrow \mathsf{if}(e_0', e_1, e_2)} \quad [\mathsf{if1}]$$

$$\mathsf{if}(\mathsf{true}, t_1, t_2) \rightarrow t_1 \quad [\mathsf{if2}]$$

$$\mathsf{if}(\mathsf{false}, t_1, t_2) \rightarrow t_2 \quad [\mathsf{if3}]$$

Derivation **sequences**

$$\mathsf{pred}(\mathsf{if}(\mathsf{iszero}(\mathsf{zero}),\mathsf{succ}(\mathsf{zero}),\mathsf{zero}))$$

$$\rightarrow \mathsf{pred}(\mathsf{if}(\mathsf{true},\mathsf{succ}(\mathsf{zero}),\mathsf{zero}))$$

$$\rightarrow \mathsf{pred}(\mathsf{succ}(\mathsf{zero}))$$

$$\rightarrow \mathsf{zero}$$

N.B.: This sequence was previously modeled by a single derivation tree in big-step style. Now, each step in the sequence involves a (simple) derivation tree.

Per-step derivation trees

$$\dfrac{\dfrac{\mathsf{iszero}(\mathsf{zero}) \rightarrow \mathsf{true}\ [\mathsf{iszero2}]}{\mathsf{if}(\mathsf{iszero}(\mathsf{zero}),\mathsf{succ}(\mathsf{zero}),\mathsf{zero}) \rightarrow \mathsf{if}(\mathsf{true},\mathsf{succ}(\mathsf{zero}),\mathsf{zero})}\ [\mathsf{if1}]}{\mathsf{pred}(\mathsf{if}(\mathsf{iszero}(\mathsf{zero}),\mathsf{succ}(\mathsf{zero}),\mathsf{zero})) \rightarrow \mathsf{pred}(\mathsf{if}(\mathsf{true},\mathsf{succ}(\mathsf{zero}),\mathsf{zero}))}\ [\mathsf{pred1}]$$

N.B.: This tree models that a step is made for the given expression on the argument position of the outermost **pred**-expression (rule [pred1] and, within this scope, on the condition position of the **if**-expression (rule [if1]).

Small-step closure and normal form

$$\mathsf{pred}(\mathsf{if}(\mathsf{iszero}(\mathsf{zero}), \mathsf{succ}(\mathsf{zero}), \mathsf{zero})) \longrightarrow^* \mathsf{zero}$$

N.B.: The reflexive, transitive closure of the small-step relation relates a phrase and its normal form, i.e., a phrase for which no further step is feasible. There are two kinds of normal forms:

- Proper results (i.e., values in the case of BTL)
- Stuck phrases (e.g., **pred(true)** in the case of BTL)

Small-step interpreter in Haskell

```
step :: Expr → Maybe Expr
step (Succ e) | Just e' ← step e = Just (Succ e')
step (Pred e) | Just e' ← step e = Just (Pred e')
step (Pred Zero) = Just Zero
step (Pred (Succ n)) | isNat n = Just n
step (IsZero e) | Just e' ← step e = Just (IsZero e')
step (IsZero Zero) = Just TRUE
step (IsZero (Succ n)) | isNat n = Just FALSE
step (If e0 e1 e2) | Just e0' ← step e0 = Just (If e0' e1 e2)
step (If TRUE e1 e2) = Just e1
step (If FALSE e1 e2) = Just e2
step _ = Nothing
```

N.B.: Interpreters may vary in terms of failure handling (when getting stuck), modularity (in terms of mapping rules to equations), and others.

Small-step semantics of simple **imperative** programs (**BIPL** — Basic Imperative Programming Language)

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

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

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

N.B.: Make a step with the statement and possibly transform a store along the way.

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

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

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

Small-step semantics of simple **functional** programs (**BFPL** — Basic Functional Programming Language)

$$\frac{fs \vdash e_0 \rightarrow e_0'}{fs \vdash \mathsf{if}(e_0, e_1, e_2) \rightarrow \mathsf{if}(e_0', e_1, e_2)} \quad [\text{if1}]$$

$$fs \vdash \mathsf{if}(\mathsf{boolconst}(\mathsf{true}), e_1, e_2) \rightarrow e_1 \quad [\text{if2}]$$

$$fs \vdash \mathsf{if}(\mathsf{boolconst}(\mathsf{false}), e_1, e_2) \rightarrow e_2 \quad [\text{if3}]$$

N.B.: Function arguments are normalized from left to right. Substitution replaces formal argument names by actual argument values.

$$\frac{fs \vdash e_{i+1} \rightarrow e_{i+1}'}{fs \vdash \mathsf{apply}(x, \langle v_1, \ldots, v_i, e_{i+1}, \ldots, e_n \rangle) \rightarrow \mathsf{apply}(x, \langle v_1, \ldots, v_i, e_{i+1}', \ldots, e_n \rangle)} \quad [\text{apply1}]$$

$$\frac{\langle x, sig, \langle \langle x_1, \ldots, x_n \rangle, e \rangle \rangle \in fs}{fs \vdash \mathsf{apply}(x, \langle v_1, \ldots, v_n \rangle) \rightarrow [v_1/x_1, \ldots, v_n/x_n]e} \quad [\text{apply2}]$$

Online resources

YAS' GitHub repository contains all code.

YAS (Yet Another SLR (Software Language Repository))

http://www.softlang.org/yas

See languages BTL, BIPL, and BFPL.

There are Haskell- and Prolog-based small-step style interpreters.

The Software Languages Book

http://www.softlang.org/book

The book discusses operational semantics in more detail.

Other related subjects:

denotational semantics and lambda calculus.