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
A semantics definition assigns meanings to language elements.

Abstract syntax of running example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>→ expr</td>
</tr>
<tr>
<td>false</td>
<td>→ expr</td>
</tr>
<tr>
<td>zero</td>
<td>→ expr</td>
</tr>
<tr>
<td>succ</td>
<td>expr → expr</td>
</tr>
<tr>
<td>pred</td>
<td>expr → expr</td>
</tr>
<tr>
<td>iszero</td>
<td>expr → expr</td>
</tr>
<tr>
<td>if</td>
<td>expr × expr × expr → expr</td>
</tr>
</tbody>
</table>

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’.

A sample term: if(true, zero, succ(zero)).

![A sample term: if(true, zero, succ(zero)).](image-url)
Big-step versus small-step

- **big step** — \( e \rightarrow v \)
- **small step** — \( e \rightarrow e' \)

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

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

© 2017 Ralf Lämmel, Software Languages Team [http://www.softlang.org/](http://www.softlang.org/), University of Koblenz-Landau. All rights reserved.
What is small-step step good for?

- big step — \( e \rightarrow v \)
- 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

zero \rightarrow zero

\begin{align*}
\frac{e \rightarrow n}{\text{succ}(e) \rightarrow \text{succ}(n)} \\
\frac{e \rightarrow \text{zero}}{\text{pred}(e) \rightarrow \text{zero}} \\
\frac{e \rightarrow \text{succ}(n)}{\text{pred}(e) \rightarrow n}
\end{align*}

\begin{align*}
e \rightarrow e' \\
\frac{\text{succ}(e) \rightarrow \text{succ}(e')}{e \rightarrow e'} \\
\frac{\text{pred}(e) \rightarrow \text{pred}(e')}{\text{pred}(\text{zero}) \rightarrow \text{zero}} \\
\frac{\text{pred}(\text{succ}(n)) \rightarrow n}{\text{pred}(\text{pred}(\text{succ}(n))) \rightarrow n}
\end{align*}

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

\[
\begin{align*}
  e &\rightarrow e' & \quad & e \rightarrow e' \\
  \text{succ}(e) &\rightarrow \text{succ}(e') & \quad & \text{iszero}(e) \rightarrow \text{iszero}(e') \\
  \text{pred}(e) &\rightarrow \text{pred}(e') & \quad & \text{iszero}(\text{pred}(\text{succ}(n))) \rightarrow \text{false} \\
  \text{pred}(\text{zero}) &\rightarrow \text{zero} & \quad & \text{if}(\text{true}, t_1, t_2) \rightarrow t_1 \\
  \text{pred}(\text{succ}(n)) &\rightarrow n & \quad & \text{if}(\text{false}, t_1, t_2) \rightarrow t_2
\end{align*}
\]
Derivation sequences

\[
pred(\text{if}(\text{iszero}(\text{zero}), \text{succ}(\text{zero}), \text{zero}))
\]

\[
\rightarrow pred(\text{if}(\text{true}, \text{succ}(\text{zero}), \text{zero}))
\]

\[
\rightarrow pred(\text{succ}(\text{zero}))
\]

\[
\rightarrow \text{zero}
\]

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

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

\[
\text{if(iszero(zero), succ(zero), zero)} \rightarrow \text{if(true, succ(zero), zero)} \quad \text{[if1]}
\]

\[
\text{pred(if(iszero(zero), succ(zero), zero)))} \rightarrow \text{pred(if(true, succ(zero), zero)))}
\]

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

\[ \text{pred}(\text{if}(\text{iszero}(\text{zero}), \text{succ}(\text{zero}), \text{zero})) \rightarrow^* \text{zero} \]

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., \text{pred}(\text{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)

\[
\begin{align*}
    m \vdash e \rightarrow v
\end{align*}
\]

\[
\langle m, \text{assign}(x, e) \rangle \rightarrow \langle m[x \leftarrow v], \text{skip} \rangle
\] [assign]

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

\[
\langle m, \text{seq}(s_1, s_2) \rangle \rightarrow \langle m', \text{seq}(s'_1, s_2) \rangle
\] [seq2]

\[
N.B.: \text{Make a step with the statement and possibly transform a store along the way,}
\]

\[
m \vdash e_0 \rightarrow \text{true}
\]

\[
\langle m, \text{if}(e_0, s_1, s_2) \rangle \rightarrow \langle m, s_1 \rangle
\] [if1]

\[
m \vdash e_0 \rightarrow \text{false}
\]

\[
\langle m, \text{if}(e_0, s_1, s_2) \rangle \rightarrow \langle m, s_2 \rangle
\] [if2]

\[
\langle m, \text{while}(e, s) \rangle \rightarrow \langle m, \text{if}(e, \text{seq}(s, \text{while}(e, s)), \text{skip}) \rangle
\] [while]
Small-step semantics of simple **functional** programs
(BFPL — Basic Functional Programming Language)

\[
\begin{align*}
fs \vdash e_0 & \rightarrow e_0' \\
fs \vdash \text{if}(e_0, e_1, e_2) & \rightarrow \text{if}(e_0', e_1, e_2) \\
fs \vdash \text{if}(\text{boolconst}(\text{true}), e_1, e_2) & \rightarrow e_1 \\
fs \vdash \text{if}(\text{boolconst}(\text{false}), e_1, e_2) & \rightarrow e_2 \\
fs \vdash e_i + 1 & \rightarrow e_i' + 1 \\
fs \vdash \text{apply}(x, \langle v_1, \ldots, v_i, e_i + 1, \ldots, e_n \rangle) & \rightarrow \text{apply}(x, \langle v_1, \ldots, v_i, e_i' + 1, \ldots, e_n \rangle) \\
\langle x, \text{sig}, \langle \langle x_1, \ldots, x_n \rangle, e \rangle \rangle & \in fs \\
fs \vdash \text{apply}(x, \langle v_1, \ldots, v_n \rangle) & \rightarrow [v_1/x_1, \ldots, v_n/x_n] e
\end{align*}
\]

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