Big-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
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 operational semantics definition

= rule-based definition of the relation between
program phrases and execution/evaluation results
with at least one rule per language construct.

zero → zero

That is, zero evaluates to zero.

\[ e \rightarrow n \]

\[ \text{succ}(e) \rightarrow \text{succ}(n) \]

That is, \text{succ}(e) evaluates \text{succ}(n), if \( e \) evaluates to \( n \).

\[ e \rightarrow \text{zero} \]

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

That is, \text{pred}(e) evaluates to zero, if \( e \) evaluates to zero.

\[ e \rightarrow \text{succ}(n) \]

\[ \text{pred}(e) \rightarrow n \]

That is, \text{pred}(e) evaluates to \( n \), if \( e \) evaluates to \text{succ}(n).

N.B.: ‘\( e \)’ proxies for expressions. ‘\( n \)’ proxies for natural numbers.
Natural numbers and Boolean values are the results (values) at hand.
Inference rules incl. axioms

<table>
<thead>
<tr>
<th>BTL example</th>
<th>General form</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ e \rightarrow n ] [ \text{succ}(e) \rightarrow \text{succ}(n) ]</td>
<td>[ P_1 \quad \cdots \quad P_n ] [ C ] [I]</td>
</tr>
<tr>
<td>zero \rightarrow zero</td>
<td>[ C ] [I]</td>
</tr>
</tbody>
</table>

N.B.: A rule consists of conclusion \( C \) and premises \( P_1, \ldots, P_n \). A rule without premises is an axiom. Rules and axioms (see [I]) are labeled for ease of reference.
Relationship claims = judgments

<table>
<thead>
<tr>
<th>Prefix notation</th>
<th>Mixfix notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluate($e, v$)</td>
<td>$e \rightarrow v$</td>
<td>The evaluation of a BTL expression $e$ results in the value $v$.</td>
</tr>
<tr>
<td>execute($m, s, m'$)</td>
<td>$m \vdash s \rightarrow m'$</td>
<td>The execution of statement $s$ in an imperative programming language transforms store $m$ into store $m'$.</td>
</tr>
<tr>
<td>evaluate($fs, m, e, v$)</td>
<td>$fs, m \vdash e \rightarrow v$</td>
<td>The evaluation of expression $e$ in a functional programming language for the arguments $m$ and function definitions $fs$ results in the value $v$.</td>
</tr>
</tbody>
</table>
Disciplined use of **metavariables**

<table>
<thead>
<tr>
<th>Type</th>
<th>Meta-variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr</code></td>
<td><code>e</code></td>
<td>Expressions according to the abstract syntax</td>
</tr>
<tr>
<td><code>nat</code></td>
<td><code>n</code></td>
<td>Natural numbers, i.e., <code>zero</code>, <code>succ(zero)</code>, …</td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>b</code></td>
<td>Boolean values, i.e., <code>true</code> and <code>false</code></td>
</tr>
<tr>
<td><code>value</code></td>
<td><code>v</code></td>
<td>Values, i.e., Boolean values and natural numbers</td>
</tr>
</tbody>
</table>

This could be any expression.

This must be a natural number!

\[
\begin{align*}
  e \rightarrow n \\
  \text{succ}(e) \rightarrow \text{succ}(n) \\
  [\text{SUCC}] 
\end{align*}
\]
All (inference) rules for BTL’s semantics

- **true → true** [TRUE]
- **false → false** [FALSE]
- **zero → zero** [ZERO]
- **e → n** [SUCC]
  - **succ(e) → succ(n)**
- **e → zero** [PRED1]
  - **pred(e) → zero**
- **e → succ(n)** [PRED2]
  - **pred(e) → n**

- **e → zero** [ISZERO1]
  - iszero(e) → true
- **e → succ(n)** [ISZERO2]
  - iszero(e) → false

- **e_0 → true**
  - **e_1 → v_1**
  - if(e_0, e_1, e_2) → v_1 [IF1]
- **e_0 → false**
  - **e_2 → v_2**
  - if(e_0, e_1, e_2) → v_2 [IF2]

That is, there are three axioms, [TRUE], [FALSE], and [ZERO], for all the constant forms of expressions. There is one rule, [SUCC], to construct the successor of a given natural number. There are two rules, [PRED1] and [PRED2], as seen earlier.
Derivation trees as proofs of judgments

N.B.: Subject to some modest constraints on the form of inference rules, these derivation trees can be constructed effectively, thereby 'interpreting' programs so that results are computed.
Mapping inference rules to Haskell

\[ \frac{e \rightarrow n}{\text{succ}(e) \rightarrow \text{succ}(n)} \]  

\text{evaluate} (\text{Succ } e)  
\left\{ \begin{array}{l}  
| n \leftarrow \text{evaluate } e  
, \text{isNat } n  
\equiv \text{Succ } n  
\end{array} \right. 

N.B.: A rule becomes an equation with premises as pattern guards (in this particular model). Also note that extra typechecks (see isNat) may be needed to account for metavariables.
Let us now strive for 1:1 correspondence between inference rules and function equations as opposed to mapping multiple rules for one construct to a single equation. Such a rule-by-rule mapping arguably better conveys the structure of the formal definition in the implementing code.

To this end, we may leverage Haskell 2010's pattern guards, which allow us to constrain equations not just by pattern matching and regular guards. That is, a regular guard is simply a Boolean expression over variables bound in the left-hand side patterns. By contrast, a pattern guard can perform more matching based on the results computed for the guard's expression. Consider this code pattern:

```
f (C x) | D y
```

This equation will be selected for an argument that is of shape `Cx`, but only if the application `gx` returns a result that can be matched with `Dy`. This expressiveness is sufficient to achieve a 1:1 correspondence between inference rules and function equations.

Illustration 8.6 (Modular mapping of rules to equations).

Haskell module `Language.BTL.BigStepWithGuards`

```haskell
evaluate :: Expr Æ Expr
evaluate TRUE = TRUE
evaluate FALSE = FALSE
evaluate Zero = Zero
evaluate (Succ e) |
  n  _| evaluate e,
  isNat n   = Succ n
evaluate (Pred e) |
  Zero  _- evaluate e
  = Zero
evaluate (Pred e) |
  Succ n  _- evaluate e,
  isNat n   = n
```

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

**(BIPL — Basic Imperative Programming Language)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Meta-variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt</td>
<td>s</td>
<td>Statements according to abstract syntax</td>
</tr>
<tr>
<td>expr</td>
<td>e</td>
<td>Expressions according to abstract syntax</td>
</tr>
<tr>
<td>uop</td>
<td>uo</td>
<td>Unary operators according to abstract syntax</td>
</tr>
<tr>
<td>bop</td>
<td>bo</td>
<td>Binary operators according to abstract syntax</td>
</tr>
<tr>
<td>string</td>
<td>x</td>
<td>Variable names</td>
</tr>
<tr>
<td>int</td>
<td>i</td>
<td>Integer values</td>
</tr>
<tr>
<td>bool</td>
<td>b</td>
<td>Boolean values</td>
</tr>
<tr>
<td>value</td>
<td>v</td>
<td>Integer and Boolean values</td>
</tr>
<tr>
<td>store</td>
<td>m</td>
<td>Collections of variable name-value pairs</td>
</tr>
</tbody>
</table>

- \( m \vdash s \rightarrow m' \) — Execution of statement \( s \) with \( m \) and \( m' \) as the stores prior and past execution, respectively.
- \( m \vdash e \rightarrow v \) — Evaluation of expression \( e \) with \( v \) as the evaluation result and \( m \) as the observed store.
Statement execution (BIPL)

\[ m \vdash \text{skip} \rightarrow m \]  \hspace{1cm} \text{[SKIP]}

\[ m \vdash e \rightarrow v \]

\[ m \vdash \text{assign}(x,e) \rightarrow m[x \mapsto v] \]  \hspace{1cm} \text{[ASSIGN]}

\[ m_0 \vdash s_1 \rightarrow m_1 \quad m_1 \vdash s_2 \rightarrow m_2 \]

\[ m_0 \vdash \text{seq}(s_1,s_2) \rightarrow m_2 \]  \hspace{1cm} \text{[SEQ]}

\[ m \vdash e_0 \rightarrow \text{true} \quad m \vdash s_1 \rightarrow m' \]

\[ m \vdash \text{if}(e_0,s_1,s_2) \rightarrow m' \]  \hspace{1cm} \text{[IF1]}

\[ m \vdash e_0 \rightarrow \text{false} \quad m \vdash s_2 \rightarrow m' \]

\[ m \vdash \text{if}(e_0,s_1,s_2) \rightarrow m' \]  \hspace{1cm} \text{[IF2]}

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

\[ m \vdash \text{while}(e,s) \rightarrow m' \]  \hspace{1cm} \text{[WHILE]}

The inference rules leverage some additional notation:

• \( x \mapsto v \) — This form of premise, as exercised in rule [VAR], applies a store in the sense of function application. The premise fails, if the store does not map the given variable identifier \( x \) to any value \( v \).
Expression evaluation (BIPL)

\[
\begin{align*}
    m \vdash \text{intconst}(i) & \rightarrow i & \text{[INTCONST]} \\
    m(x) & \rightarrow v & m \vdash \text{var}(x) \rightarrow v & \text{[VAR]} \\
    m \vdash e & \rightarrow v & \text{unary}(uo,v) & \rightarrow v' & m \vdash \text{unary}(uo,e) \rightarrow v' & \text{[UNARY]} \\
    m \vdash e_1 & \rightarrow v_1 & m \vdash e_2 & \rightarrow v_2 & \text{binary}(bo,v_1,v_2) & \rightarrow v' & m \vdash \text{binary}(bo,e_1,e_2) \rightarrow v' & \text{[BINARY]} \\
\end{align*}
\]
There is the judgment \( fs, m \vdash e \rightarrow v \) for expression evaluation with \( e \) as the expression to be evaluated, \( v \) as the evaluation result, \( fs \) as the list of defined functions, and \( m \) as the current argument binding (‘environment’).
Expression evaluation (BFPL) I/II

\[
fs, m \vdash \text{intconst}(i) \rightarrow i \quad \text{[INTCONST]}
\]

\[
fs, m \vdash \text{boolconst}(b) \rightarrow b \quad \text{[BOOLCONSTR]}
\]

\[
\langle x, v \rangle \in m \quad \frac{\vphantom{\langle x, v \rangle \in m}}{fs, m \vdash \text{arg}(x) \rightarrow v} \quad \text{[ARG]}
\]

\[
fs, m \vdash e_0 \rightarrow \text{true} \quad fs, m \vdash e_1 \rightarrow v \quad \frac{\vphantom{fs, m \vdash e_0 \rightarrow \text{true} \quad fs, m \vdash e_1 \rightarrow v}}{fs, m \vdash \text{if}(e_0, e_1, e_2) \rightarrow v} \quad \text{[IF1]}
\]

\[
fs, m \vdash e_0 \rightarrow \text{false} \quad fs, m \vdash e_2 \rightarrow v \quad \frac{\vphantom{fs, m \vdash e_0 \rightarrow \text{false} \quad fs, m \vdash e_2 \rightarrow v}}{fs, m \vdash \text{if}(e_0, e_1, e_2) \rightarrow v} \quad \text{[IF2]}
\]

\[
fs, m \vdash e \rightarrow v \quad \text{unary}(uo, v) \rightarrow v' \quad \frac{\vphantom{fs, m \vdash e \rightarrow v \quad \text{unary}(uo, v) \rightarrow v'}}{fs, m \vdash \text{unary}(uo, e) \rightarrow v'} \quad \text{[UNARY]}
\]

\[
fs, m \vdash e_1 \rightarrow v_1 \quad fs, m \vdash e_2 \rightarrow v_2 \quad \text{binary}(bo, v_1, v_2) \rightarrow v' \quad \frac{\vphantom{fs, m \vdash e_1 \rightarrow v_1 \quad fs, m \vdash e_2 \rightarrow v_2 \quad \text{binary}(bo, v_1, v_2) \rightarrow v'}}{fs, m \vdash \text{binary}(bo, e_1, e_2) \rightarrow v'} \quad \text{[BINARY]}
\]
Expression evaluation (BFPL) II/II

N.B.: Each function application constructs an environment to be used for the evaluation of the body of the function.
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 big-step style interpreters.

The Software Languages Book
http://www.softlang.org/book
The book discusses big-step semantics in more detail.
Other related subjects:
small-step operational and denotational semantics.