An interpreter for every season

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

- **Concrete** interpretation
- **Ad-hoc** interpretation
- **Operational** style
  - **Big**-step style
  - **Small**-step style
- **Denotational** style
- **Direct** style
- **Continuation** style

- **Abstract** interpretation
- **Standard** interpretation
- **Type** checking
- **Program** analysis
- **Inlining** interpretation
- **Specializing** interpretation
- … as opposed to … **Translation**


© 2017 Ralf Lämmel, Software Languages Team [http://www.softlang.org/], University of Koblenz-Landau. All rights reserved.
# The Software Languages Book


The book uses Haskell, Java, Python, and Prolog.

The underline chapter titles (see below) use Haskell systematically.

## Table of contents

- Preface
- Part I — Introduction
  - The notion of software language
  - Story of a language
- Part II — Syntax and metaprogramming
  - Abstract syntax — foundations
  - Abstract syntax — implementation
  - Basics of metaprogramming
  - Concrete syntax — foundations
  - Concrete syntax — implementation
- Part III — Semantics and types
  - Operational semantics
  - Type systems
  - Denotational semantics
  - The lambda calculus
- Part IV — Consolidation
  - Metaprogramming techniques
  - Logic metaprogramming
- Postface

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

YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas

MetaLib (Comparison of metaprogramming technologies)
www.softlang.org/metalib

The Software Languages Book
http://www.softlang.org/book
Ad-hoc interpretation
An ad-hoc recursive interpreter for BTL
(Basic TAPL Language)

-- Results of evaluation

type Value = Either Int Bool

evaluate :: Expr -> Value
evaluate TRUE = Right True
evaluate FALSE = Right False
evaluate Zero = Left 0
evaluate (Succ e) = Left (n+1) where Left n = evaluate e
evaluate (Pred e) = Left (n - if n==0 then 0 else 1) where Left n = evaluate e
evaluate (IsZero e) = Right (n==0) where Left n = evaluate e
evaluate (If e0 e1 e2) = evaluate (if b then e1 else e2) where Right b = evaluate e0

N.B.: Functions interpret syntactic patterns.
Recursion is used to descend into components.
An ad-hoc recursive interpreter for BTL (Basic TAPL Language)

evaluate :: Expr -> Value
evaluate TRUE = Right True
evaluate FALSE = Right False
evaluate Zero = Left 0
evaluate (Succ e) = Left (n+1) where Left n = evaluate e
evaluate (Pred e) = Left (n - if n==0 then 0 else 1) where Left n = evaluate e
evaluate (IsZero e) = Right (n==0) where Left n = evaluate e
evaluate (If e0 e1 e2) = evaluate (if b then e1 else e2)

N.B.: Functions interpret syntactic patterns.
An ad-hoc recursive interpreter for BTL
(Basic TAPL Language)

Expr -> Value
JE = Right True
SE = Right False
α = Left 0
icc e) = Left (n+1) where Left n = evaluate e
ied e) = Left (n - if n==0 then 0 else 1) where Left n = evaluate e
Zero e) = Right (n==0) where Left n = evaluate e
e0 e1 e2) = evaluate (if b then e1 else e2) where Right b = evaluate e0

N.B.: Recursion is used to descend into components.
An ad-hoc recursive interpreter for a Haskell subset (BFPL)

-- Evaluation of a program's main expression
evaluate :: Program -> Value
evaluate (fs, e) = f e empty

where

-- Evaluation of expressions
f :: Expr -> Env -> Value
f (IntConst i) _ = Left i
f (BoolConst b) _ = Right b
f (Arg x) m = m!x
f (If e0 e1 e2) m = f (if b then e1 else e2) m where Right b = f e0 m
f (Unary o e) m = uop o (f e m)
f (Binary o e1 e2) m = bop o (f e1 m) (f e2 m)
f (Apply x es) m = f body m'

where

Just (_, (xs, body))) = lookup x fs
vs = map (flip f m) es
m' = fromList (zip xs vs)

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

-- Interpretation of binary operators
bop :: BOp -> Value -> Value -> Value

...
Big-step operational semantics
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, succ(e) evaluates succ(n), if e evaluates to n.

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

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

That is, pred(e) evaluates to zero, if e evaluates to zero.

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

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

That is, pred(e) evaluates to n, if e evaluates to 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>$e \rightarrow n$</td>
<td>$\frac{P_1 \cdots P_n}{C}$ [I]</td>
</tr>
<tr>
<td>$\text{succ}(e) \rightarrow \text{succ}(n)$</td>
<td></td>
</tr>
<tr>
<td>zero $\rightarrow$ zero</td>
<td>$C$ [I]</td>
</tr>
</tbody>
</table>

**[SUCCE]**

**[ZERO]**

---

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

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

\[
e \rightarrow n
\]

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

This could be any expression.  
This must be a natural number!
All (inference) rules for BTL’s semantics

- **true → true**  
  
- **false → false**  
  
- **zero → zero**  
  
- **$e → n$**  
  **succ(e) → succ(n)**  
  
- **$e → zero$**  
  **pred(e) → zero**  
  
- **$e → succ(n)$**  
  **pred(e) → n**  

**[TRUE]**

**[FALSE]**

**[ZERO]**

**[SUCCE]**

**[ISZERO1]**  
\[ e → zero \]
\[ \text{iszero}(e) → true \]

**[ISZERO2]**  
\[ e → succ(n) \]
\[ \text{iszero}(e) → false \]

**[IF1]**  
\[ e_0 → true \quad e_1 → v_1 \]
\[ \text{if}(e_0, e_1, e_2) → v_1 \]

**[IF2]**  
\[ e_0 → false \quad e_2 → v_2 \]
\[ \text{if}(e_0, e_1, e_2) → v_2 \]

---

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

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

\([\text{SUCC}]\)

\[ \begin{align*}
\text{evaluate} & (\text{Succ } e) \\
\mid & n \leftarrow \text{evaluate } e \\
, & \text{isNat } n \\
= & \text{Succ } n
\end{align*} \]

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.
Big-step interpreter in Haskell

evaluate :: Expr \rightarrow 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

evaluate (IsZero e)
    | Zero ← evaluate e
    = TRUE
evaluate (IsZero e)
    | Succ n ← evaluate e
    , isNat n
    = FALSE
evaluate (If e0 e1 e2)
    | TRUE ← evaluate e0
    = evaluate e1
evaluate (If e0 e1 e2)
    | FALSE ← evaluate e0
    = evaluate e2

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><code>stmt</code></td>
<td><code>s</code></td>
<td>Statements according to abstract syntax</td>
</tr>
<tr>
<td><code>expr</code></td>
<td><code>e</code></td>
<td>Expressions according to abstract syntax</td>
</tr>
<tr>
<td><code>uop</code></td>
<td><code>uo</code></td>
<td>Unary operators according to abstract syntax</td>
</tr>
<tr>
<td><code>bop</code></td>
<td><code>bo</code></td>
<td>Binary operators according to abstract syntax</td>
</tr>
<tr>
<td><code>string</code></td>
<td><code>x</code></td>
<td>Variable names</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>i</code></td>
<td>Integer values</td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>b</code></td>
<td>Boolean values</td>
</tr>
<tr>
<td><code>value</code></td>
<td><code>v</code></td>
<td>Integer and Boolean values</td>
</tr>
<tr>
<td><code>store</code></td>
<td><code>m</code></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)

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

\[
m \vdash e \rightarrow v
\]

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

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

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

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

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

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

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

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

\[
m \vdash \text{while}(e, s) \rightarrow m'
\]
Expression evaluation (BIPL)

\[ m \vdash \text{intconst}(i) \rightarrow i \]  
\[ m(x) \rightarrow v \rightarrow m \vdash \text{var}(x) \rightarrow v \]  
\[ m \vdash e \rightarrow v \rightarrow \text{unary}(u_0, v) \rightarrow v' \rightarrow m \vdash \text{unary}(u_0, e) \rightarrow v' \]  
\[ m \vdash e_1 \rightarrow v_1 \rightarrow m \vdash e_2 \rightarrow v_2 \rightarrow \text{binary}(b_0, v_1, v_2) \rightarrow v' \rightarrow m \vdash \text{binary}(b_0, e_1, e_2) \rightarrow v' \]
Semantics of simple *functional* programs  
(**BFPL** — Basic Functional Programming Language)

<table>
<thead>
<tr>
<th>Type</th>
<th>Meta-variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>programs</td>
<td>(p)</td>
<td>Programs according to abstract syntax</td>
</tr>
<tr>
<td>functions</td>
<td>(fs)</td>
<td>Function collections according to abstract syntax</td>
</tr>
<tr>
<td>funsig</td>
<td>(sig)</td>
<td>Function signatures 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>Function and argument 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>Integers and Boolean values</td>
</tr>
<tr>
<td>env</td>
<td>(m)</td>
<td>Collections of argument name-value pairs</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

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

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Expression evaluation (BFPL) II/II

\[ fs, m \vdash e_1 \rightarrow v_1 \quad \cdots \quad fs, m \vdash e_n \rightarrow v_n \]

\[ \langle x, \text{sig}, \langle \langle x_1, \ldots, x_n \rangle, e \rangle \rangle \in fs \]

\[ fs, [x_1 \leftrightarrow v_1, \ldots, x_n \leftrightarrow v_n] \vdash e \rightarrow v \]

\[ fs, m \vdash \text{apply}(x, \langle e_1, \ldots, e_n \rangle) \rightarrow v \]

N.B.: Each function application constructs an environment to be used for the evaluation of the body of the function.
Let’s explore YAS
(Yet Another SLR
(Software Language Repository))
Small-step operational semantics
Big-step versus small-step for BTL

zero $\rightarrow$ zero

$e \rightarrow n$
$succ(e) \rightarrow succ(n)$

$e \rightarrow zero$
$pred(e) \rightarrow zero$

$e \rightarrow succ(n)$
$pred(e) \rightarrow n$

$e \rightarrow e'$
$succ(e) \rightarrow succ(e')$

$e \rightarrow e'$
$pred(e) \rightarrow pred(e')$

$pred(0) \rightarrow zero$

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

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

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

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

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

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

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

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

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

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

\[
\text{if}(\text{false}, t_1, t_2) \rightarrow t_2 \quad [\text{if3}]
\]
Derivation sequences

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

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

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

\[\rightarrow \text{zero}\]

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

\[
iszero(\text{zero}) \rightarrow \text{true} \quad [\text{iszzero2}]
\]

\[
\begin{align*}
\text{if}(\text{iszzero}(\text{zero}), \text{succ}(\text{zero}), \text{zero}) & \rightarrow \text{if}(\text{true}, \text{succ}(\text{zero}), \text{zero}) \\
\text{pred}(\text{if}(\text{iszzero}(\text{zero}), \text{succ}(\text{zero}), \text{zero})) & \rightarrow \text{pred}(\text{if}(\text{true}, \text{succ}(\text{zero}), \text{zero}))
\end{align*} \quad [\text{if1}]
\]

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

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

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)
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.
-- Boolean values
isBool :: Expr -> Bool
isBool TRUE = True
isBool FALSE = True
isBool _ = False

-- Natural numbers
isNat :: Expr -> Bool
isNat Zero = True
isNat (Succ e) = isNat e
isNat _ = False

-- Values
isValue :: Expr -> Bool
isValue e = isBool e || isNat e
steps :: (Expr -> Maybe Expr) -> Expr -> Maybe Expr
steps f e =
  if isValue e
    then Just e
    else case f e of
      (Just e') -> steps f e'
      Nothing -> Nothing
Small-step semantics of simple **imperative** programs  
**BIPL** — Basic Imperative Programming Language

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

[assign]  
[seq1]  
[seq2]  
[if1]  
[if2]  
[while]

N.B.: Make a step with the statement and possibly transform a store along the way,
Small-step semantics of simple **functional** programs  
(BFPL — Basic Functional Programming Language)

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

N.B.: Function arguments are normalized from left to right. Substitution replaces formal argument names by actual argument values.
Denotational interpretation
(Direct style)
Denotational semantics
= functional semantics

Semantic domains = function types of meanings
storeT = store → store // Type of store transformation
storeO = store → value // Type of store observation

Semantic functions = mappings from syntax to semantics
S : stmt → storeT // Semantics of statements
E : expr → storeO // 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**

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

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

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).
The compositional scheme

\[ storeT = store \rightarrow store \]  // Type of store transformation
\[ storeO = store \rightarrow value \]  // Type of store observation

\[ S : stmt \rightarrow storeT \]  // Semantics of statements
\[ \mathcal{E} : expr \rightarrow storeO \]  // Semantics of expressions

\[
S[skip] = \underline{skip} \\
S[assign(x,e)] = assign x (\mathcal{E}[e]) \\
S[seq(s_1,s_2)] = seq (S[s_1]) (S[s_2]) \\
S[if(e,s_1,s_2)] = if (\mathcal{E}[e]) (S[s_1]) (S[s_2]) \\
S[while(e,s)] = while (\mathcal{E}[e]) (S[s])
\]

\[
\mathcal{E}[\text{intconst}(i)] = \underline{\text{intconst}} i \\
\mathcal{E}[\text{var}(x)] = \underline{\text{var}} x \\
\mathcal{E}[\text{unary}(o,e)] = \underline{\text{unary}} o (\mathcal{E}[e]) \\
\mathcal{E}[\text{binary}(o,e_1,e_2)] = \underline{\text{binary}} o (\mathcal{E}[e_1]) (\mathcal{E}[e_2])
\]

N.B.: The underlined functions are the \textbf{semantic combinators}. They combine meanings — no syntax is involved.
Types of semantic combinators

\[
\begin{align*}
\text{skip} & : \text{storeT} \\
\text{assign} & : \text{string} \rightarrow \text{storeO} \rightarrow \text{storeT} \\
\text{seq} & : \text{storeT} \rightarrow \text{storeT} \rightarrow \text{storeT} \\
\text{if} & : \text{storeO} \rightarrow \text{storeT} \rightarrow \text{storeT} \rightarrow \text{storeT} \\
\text{while} & : \text{storeO} \rightarrow \text{storeT} \rightarrow \text{storeT} \\
\text{intconst} & : \text{int} \rightarrow \text{storeO} \\
\text{var} & : \text{string} \rightarrow \text{storeO} \\
\text{unary} & : \text{uo} \rightarrow \text{storeO} \rightarrow \text{storeO} \\
\text{binary} & : \text{bo} \rightarrow \text{storeO} \rightarrow \text{storeO} \rightarrow \text{storeO}
\end{align*}
\]

N.B.: The combinators combine meanings — no syntax is involved.

Well, ints, strings, and operators are hybrids.
Semi-formal definitions of semantic combinators

// The identity function for type store
skip m = m

// Pointwise store update
assign x f m = m[x \mapsto (f m)], if f m is defined

// Function composition for type storeT
seq f g m = g (f m)

// Select either branch for Boolean value
if f g h m = \begin{cases} 
g m, & \text{if } f m = \text{true} \\
h m, & \text{if } f m = \text{false} \\
\text{undefined}, & \text{otherwise}
\end{cases}

N.B.: What about \textbf{while}?
Fixed-point semantics I/II

Suppose $\textbf{while } f g = t$. Then, it would hold that:

$$t \equiv \textbf{if } f \ (\textbf{seq } g \ t) \ \textbf{skip}$$

Let us capture the expression as $h$ parametrized in $t$:

$$h \ t \ = \ \textbf{if } f \ (\textbf{seq } g \ t) \ \textbf{skip}$$

Now consider the following progression of applications of $h$:

$$h \ undefined$$
$$h \ (h \ undefined)$$
$$h \ (h \ (h \ undefined))$$

$$\vdots$$

The meaning of the while-loop can be understood as the repeated application of $h$ to an undefined meaning while the number of repetitions is unbounded.
Fixed-point semantics II/II

The meaning of the while-loop is thus:

\[ \text{while } f \ g = \text{fix } h \]

where

\[ h \ t = \text{if } f \ (\text{seq } g \ t) \ \text{skip} \]

The fixed-point combinator is ‘defined’ as follows:

\[ \text{fix } k = k \ (\text{fix } k) \]

N.B.: For such a fixed-point semantics to make sense, denotational semantics definitions need to meet some constraints studied in domain theory, which we skip here (and in the book as well).
Denotational interpreter in Haskell

execute euclideanDiv (fromList [("x", Left 14), ("y", Left 4)])
fromList [("q", Left 3), ("r", Left 2), ("x", Left 14), ("y", Left 4)]

Ingredients in need of encoding:

• Abstract syntax
• Semantic **domains**
• Semantic **functions**
• Semantic **combinators**
• fix

© 2017 Ralf Lämmel, Software Languages Team [http://www.softlang.org/], University of Koblenz-Landau. All rights reserved.
Denotational interpreter in Haskell:

`-- Results of expression evaluation`

```haskell

type Value = Either Int Bool

`-- Stores as maps from variable ids to values`

type Store = Map String Value

`-- Store transformers (semantics of statements)`

type StoreT = Store -> Store

`-- Store observers (semantics of expressions)`

type StoreO = Store -> Value

execute :: Stmt -> StoreT
execute Skip = skip'
execute (Assign x e) = assign' x (evaluate e)
execute (Seq s1 s2) = seq' (execute s1) (execute s2)
execute (If e s1 s2) = if' (evaluate e) (execute s1) (execute s2)
execute (While e s) = while' (evaluate e) (execute s)

evaluate :: Expr -> StoreO
evaluate (IntConst i) = intconst' i

...
Denotational interpreter in Haskell: semantic combinators

\[
\begin{align*}
\text{skip}' &:: \text{StoreT} \\
\text{skip}' &= \text{id} \\
\text{assign}' &:: \text{String} \to \text{StoreO} \to \text{StoreT} \\
\text{assign}' \times f m &= \text{insert} \times (f m) m \\
\text{seq}' &:: \text{StoreT} \to \text{StoreT} \to \text{StoreT} \\
\text{seq}' &= \text{flip} (.) \\
\text{if}' &:: \text{StoreO} \to \text{StoreT} \to \text{StoreT} \to \text{StoreT} \\
\text{if}' f g h m &= \text{let} \ \text{Right} v = f m \ \text{in} \ \text{if} v \ \text{then} g m \ \text{else} h m \\
\text{while}' &:: \text{StoreO} \to \text{StoreT} \to \text{StoreT} \\
\text{while}' f g &= \text{fix} h \ \text{where} \ h \ t = \text{if}' f \ (\text{seq}' g t) \ \text{skip}' \\
\text{intconst}' &:: \text{Int} \to \text{StoreO} \\
\text{intconst}' i \_ &= \text{Left} i \\
\ldots
\end{align*}
\]

The fixed-point combinator at hand

\[
\begin{align*}
\text{fix} &:: (\text{a} \to \text{a}) \to \text{a} \\
\text{fix} f &= f \ (\text{fix} f)
\end{align*}
\]
Denotational interpretation (Continuation style)
How to jump?
How to throw?
(We can’t.)

execute :: Stmt → StoreT
execute Skip = skip'
execute (Assign x e) = assign' x (evaluate e)
execute (Seq s1 s2) = seq' (execute s1) (execute s2)
execute (If e s1 s2) = if’ (evaluate e) (execute s1) (execute s2)
execute (While e s) = while' (evaluate e) (execute s)

evaluate :: Expr → StoreO
evaluate (IntConst i) = intconst' i
...

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Semantic domains for continuation style

-- Results of expression evaluation
type Value = Either Int Bool
-- Stores as maps from variable ids to values
type Store = Map String Value
-- Store transformers (semantics of statements)
type StoreT = Store -> Store
-- Store observers (semantics of expressions)
type StoreO = Store -> Value
-- Goto tables
type Gotos = [(String, StoreT)]
-- Transformation with gotos
type StoreTT' = (StoreT, Gotos) -> (StoreT, Gotos)
Continuation style interpreter

execute :: Stmt -> StoreT
execute s = execute' s id
  where
    execute' :: Stmt -> StoreTT
    execute' Skip = skip'
    execute' (Assign x e) = assign' x (evaluate e)
    execute' (Seq s1 s2) = seq' (execute' s1) (execute' s2)
    execute' (If e s1 s2) = if' (evaluate e) (execute' s1) (execute' s2)
    execute' (While e s) = while' (evaluate e) (execute' s)
Semantic combinators on continuations

skip' :: StoreTT
skip' = id
assign' :: String -> Store0 -> StoreTT
assign' x f c sto = c (insert x (f sto) sto)
seq' :: StoreTT -> StoreTT -> StoreTT
seq' = (.)
if' :: Store0 -> StoreTT -> StoreTT -> StoreTT
if' f g h c = DS.if' f (g c) (h c)
while' :: Store0 -> StoreTT -> StoreTT
while' f g c = fix h c where h t = if' f (seq' g t) skip'
The pattern of abstract interpretation
Signature of BIPL algebras

-- Aliases to shorten function signatures
type Trafo sto = sto -> sto -- Store transformation
type Obs sto val = sto -> val -- Store observation
-- The signature of algebras for interpretation
data Alg sto val = Alg {
  skip' :: Trafo sto,
  assign' :: String -> Obs sto val -> Trafo sto,
  seq' :: Trafo sto -> Trafo sto -> Trafo sto,
  if' :: Obs sto val -> Trafo sto -> Trafo sto -> Trafo sto,
  while' :: Obs sto val -> Trafo sto -> Trafo sto,
  intconst' :: Int -> Obs sto val,
  var' :: String -> Obs sto val,
  unary' :: UOp -> Obs sto val -> Obs sto val,
  binary' :: BOp -> Obs sto val -> Obs sto val -> Obs sto val
}
The scheme of abstract interpretation

\[
\text{interpret} :: \text{Alg sto val} \rightarrow \text{Stmt} \rightarrow \text{sto} \rightarrow \text{sto}
\]

\[
\text{interpret } a = \text{execute}
\]

where

-- Compositional interpretation of statements
\[
\text{execute } \text{Skip} = \text{skip}' a
\]
\[
\text{execute } (\text{Assign } x e) = \text{assign}' a x (\text{evaluate } e)
\]
\[
\text{execute } (\text{Seq } s1 s2) = \text{seq}' a (\text{execute } s1) (\text{execute } s2)
\]
\[
\text{execute } (\text{If } e s1 s2) = \text{if}' a (\text{evaluate } e) (\text{execute } s1) (\text{execute } s2)
\]
\[
\text{execute } (\text{While } e s) = \text{while}' a (\text{evaluate } e) (\text{execute } s)
\]

-- Compositional interpretation of expressions
\[
\text{evaluate } (\text{IntConst } i) = \text{intconst}' a i
\]
\[
\text{evaluate } (\text{Var } n) = \text{var}' a n
\]
\[
\text{evaluate } (\text{Unary } o e) = \text{unary}' a o (\text{evaluate } e)
\]
\[
\text{evaluate } (\text{Binary } o e1 e2) = \text{binary}' a o (\text{evaluate } e1) (\text{evaluate } e2)
\]
An algebra for standard interpretation

```
type Value = Either Int Bool
type Store = Map String Value
algebra :: Alg Store Value
algebra = a where a = Alg {
  skip' = id,
  assign' = \n f m -> insert n (f m) m,
  seq' = flip (.),
  if' = \f g h m -> let (Right b) = f m in if b then g m else h m,
  while' = \f g -> fix (\x -> if' a f (seq' a g x) (skip' a)),
  intconst' = \i -> const (Left i),
  var' = \n m -> m!n,
  unary' = \o f m ->
    case (o, f m) of
      (Negate, Left i) -> Left (negate i)
      (Not, Right b) -> Right (not b),
  binary' = \o f g m -> ...
}
```
Type checking by abstract interpretation
Signature of *monadic* BIPL algebras

-- Aliases to shorten function signatures

```haskell
type Trafo m sto = sto -> m sto -- Store transformation
type Obs m sto val = sto -> m val -- Store observation
-- The signature of algebras for interpretation
data Alg m sto val = Alg {
    skip' :: Trafo m sto,
    assign' :: String -> Obs m sto val -> Trafo m sto,
    seq' :: Trafo m sto -> Trafo m sto -> Trafo m sto,
    if' :: Obs m sto val -> Trafo m sto -> Trafo m sto -> Trafo m sto,
    while' :: Obs m sto val -> Trafo m sto -> Trafo m sto,
    intconst' :: Int -> Obs m sto val,
    var' :: String -> Obs m sto val,
    unary' :: UOp -> Obs m sto val -> Obs m sto val,
    binary' :: BOp -> Obs m sto val -> Obs m sto val -> Obs m sto val
}```
An algebra for type checking

data Type = IntType | BoolType

type VarTypes = Map String Type

algebra :: Alg Maybe VarTypes Type

algebra = Alg {  
  skip' = Just,
  assign' = \x f m -> f m >>= \t ->
          case lookup x m of
            (Just t') -> guard (t==t') >>= Just m
            Nothing    -> Just (insert x t m),
  seq' = flip (<<=),
  if' = \f g h m -> do
    t <- f m
    guard (t==BoolType)
    m1 <- g m
    m2 <- h m
    guard (m1==m2)
    Just m1,
  while' = \f g m -> do
    t <- f m
    guard (t==BoolType)
    m' <- g m
    guard (m==m')
    Just m,
  var' = \x m -> lookup x m,
  unary' = \o f m -> f m >>= \t ->
          case (o, t) of
            (Negate, IntType) -> Just IntType
            (Not, BoolType)   -> Just BoolType
            _                  -> Nothing,
  binary' = \o f g m -> ...

}
Program analysis (sign detection) by abstract interpretation
Optimization potential subject to sign detection

```plaintext
{
    ...
    y = x * x + 42;
    if (y < 0)
        y = -y;
    ...
}
```
From concrete to abstract domains
Expected I/O behavior of sign detection (and limits thereof)

// Assume x to be positive
y = 1;
i = 1;
while (i <= x) {
y = y * i;
i = i + 1;
}

// Assume x to be positive
y = 1;
while (x >= 2) {
y = y * x;
x = x - 1;
}

➤ interpret BasicAnalysis.algebra facv1 (fromList [("x", Left Pos)])
fromList [("i", Left Pos), ("x", Left Pos), ("y", Left Pos)]

➤ interpret BasicAnalysis.analysis facv2 (fromList [("x", Left Pos)])
fromList [("x", Left TopSign), ("y", Left TopSign)]
data Sign = Zero | Pos | Neg | BottomSign | TopSign

instance Num Sign
  where
    fromInteger n
    | n > 0 = Pos
    | n < 0 = Neg
    | otherwise = Zero

TopSign + _ = TopSign
_ + TopSign = TopSign
BottomSign + _ = BottomSign
_ + BottomSign = BottomSign
Zero + Zero = Zero
Zero + Pos = Pos
...

instance CPO Sign where
  pord x y | x == y = True
  pord BottomSign _ = True
  pord _ TopSign = True
  pord _ _ = False
  lub x y | x == y = x
  lub BottomSign x = x
  lub x BottomSign = x
  lub _ _ = TopSign

instance Bottom Sign where
  bottom = BottomSign
  ...
An algebra for sign detection

type Property = Either Sign CpoBool

type VarProperties = Map String Property

algebra :: Alg VarProperties Property

algebra = a where a = Alg {
    skip' = id,
    assign' = \n f m -> insert n (f m) m,
    seq' = flip (.),
    if' = \f g h m ->
        let Right b = f m in
            case b of
                (ProperBool True) -> g m
                (ProperBool False) -> h m
                BottomBool -> bottom
                TopBool -> g m `lub` h m,
    while' = \f g -> fix' (\x -> if' a f (x . g) id) (const bottom),
    intconst' = \i -> const (Left (fromInteger (toInteger i))),
    var' = \n m -> m!n,
    unary' = \o f m ->
        case (o, f m) of
            (Negate, Left s) -> Left (negate s)
            (Not, Right b) -> Right (cpoNot b),
    binary' = \o f g m -> ...
}
A designated fixed point combinator

\[ \text{fix'} :: \text{Eq a} \Rightarrow ((a \to a) \to a \to a) \to (a \to a) \to a \to a \]
\[ \text{fix'} h i x = \text{limit} (\text{iterate } h i) \]
\[
\quad \text{where} \quad \text{limit} (b1:b2:bs) = \text{if } b1 \ x = = b2 \ x \ \text{then } b1 \ x \ \text{else} \ \text{limit} (b2:bs)
\]
Complete partial orders

instance (Ord k, CPO v) => CPO (Map k v) where
  pord x y = and (map (f y) (toList x))
    where f y (k,v) = pord v (y!k)
  lub x y = foldr f y (toList x)
    where f (k,v) m = Data.Map.insert k (lub v (y!k)) m
instance (Ord k, CPO v) => Bottom (Map k v) where
  bottom = empty
Inlining
Let’s optimize applications of power!

```
power :: Int -> Int -> Int
power n x = 
  if (==) n 0 
  then 1 
  else (*) x (power ((-) n 1) x)
```

```
power :: Function
power = ( 
  "power",
  ([IntType, IntType], IntType),
  (["n", "x"],
   If (Binary Eq (Arg "n") (IntConst 0))
       (IntConst 1) 
       (Binary Mul 
        (Arg "x")
        (Apply "power" [Binary Sub (Arg "n") (IntConst 1), Arg "x"])))))
```
Let’s optimize applications of power!

```
peval ([power], (Apply "power" [IntConst 3, IntConst 2]))
IntConst 8
```

```
peval ([power], (Apply "power" [IntConst 3, Arg "x"]))
Binary Mul (Arg "x") (Binary Mul (Arg "x") (Binary Mul (Arg "x") (IntConst 1)))
```
An interpreter with inlining

type Env = Map String Expr

peval :: Program → Expr
peval (fs, e) = f e empty

where
  f :: Expr → Env → Expr
  f e@(IntConst _) _ = e
  f e@(BoolConst _) _ = e
  f e@(Arg x) env =
    case Data.Map.lookup x env of
      (Just e') → e'
      Nothing → e
  f (If e0 e1 e2) env =
    let
      r0 = f e0 env
      r1 = f e1 env
      r2 = f e2 env
    in
      case toValue r0 of
        (Just (Right bv)) → if bv then r1 else r2
        Nothing → If r0 r1 r2

  f (Unary o e) env =
    let r = f e env
    in case toValue r of
      (Just v) → fromValue (uop o v)
      _ → Unary o r
  f (Binary o e1 e2) env = ...
  f (Apply fn es) env = f body env'

  where
    Just (_, (ns, body)) = Prelude.lookup fn fs
    rs = map (flip f env) es
    env' = fromList (zip ns rs)

  -- Attempt extraction of value from expression
  toValue :: Expr → Maybe Value
  toValue (IntConst iv) = Just (Left iv)
  toValue (BoolConst bv) = Just (Right bv)
  toValue _ = Nothing

  -- Represent value as expression
  fromValue :: Value → Expr
  fromValue (Left iv) = IntConst iv
  fromValue (Right bv) = BoolConst bv
Limits of inlining

peval ([power], (Apply "power" [Arg "n", IntConst 2]))
if (==) n 0
then 1
else (*) 2 (if (==) ((-) n 1) 0)
then 1
else (*) 2 (if (==) ((-) ((-) n 1) 1) 0 ...)
Specialization
From inlining to specialization

```
peval ([power], (Apply "power" [IntConst 3, Arg "x"]))
(["power'a", ([IntType], IntType), (["x'",
  Binary Mul (Arg "x") (Apply "power'b" [Arg "x"]))]),
("power'b", ([IntType], IntType), (["x'",
  Binary Mul (Arg "x") (Apply "power'c" [Arg "x"]))]),
("power'c", ([IntType], IntType), (["x'",
  Binary Mul (Arg "x") (Apply "power'd" [Arg "x"]))]),
("power'd", ([IntType], IntType), (["x'",
    IntConst 1])))
],
Apply "power'a" [Arg "x"]
```

In concrete syntax:

```
exp'a x = x * exp'b x
exp'b x = x * exp'c x
exp'c x = x * exp'd x
exp'd x = 1
```
Specialization is more general than inlining

```latex
\texttt{peval ([power], (Apply "power" [Arg "n", IntConst 2])))}

( [ "power'a", ([IntType], IntType), (["n"],
    If (Binary Eq (Arg "n")
        (IntConst 0))
        (IntConst 1) (Binary Mul
            (IntConst 2)
            (Apply "power'a" [Binary Sub (Arg "n") (IntConst 1)])))))

Apply "power'a" [Arg "n"]
```
Aggregation of specialized functions

```haskell
type Env = Map String Value

peval :: Program → Program
peval (fs, e) = swap (runState (f e empty) [])

where
  f :: Expr → Env → State [Function] Expr

...
Evaluation, when possible.
Pattern reconstruction, otherwise.

\[ f \text{(Binary } o \text{ e1 e2)} \text{ env} = \begin{cases} \text{do} \\ r1 \leftarrow f \text{ e1 env} \\ r2 \leftarrow f \text{ e2 env} \\ \text{case (toValue r1, toValue r2) of} \\ \quad (\text{Just v1, Just v2}) \rightarrow \text{return (fromValue (bop o v1 v2))} \\ \quad _ \rightarrow \text{return (Binary } o \text{ r1 r2)} \end{cases} \]
f (Apply fn es) env = do
  -- Look up function
  let Just ((ts, t), (ns, body)) = Prelude.lookup fn fs
  -- Partially evaluate arguments
  rs ← mapM (flip f env) es
  -- Determine static and dynamic arguments
  let trs = zip ts rs
  let ntrs = zip ns trs
  let sas = [ (n, fromJust (toValue r)) | (n, (r)) ← ntrs, isJust (toValue r) ]
  let das = [ (n, (t, r)) | (n, (t, r)) ← ntrs, isNothing (toValue r) ]
  -- Specialize body
  let body' = f body (fromList sas)
  -- Inlining as a special case
  if null das then body'
  -- Specialization
  else do
    -- Fabricate function name
    let fn' = fn ++ show sas
    -- Memoize new residual function, if necessary
    fs' ← get
    when (isNothing (Prelude.lookup fn' fs')) (do
      -- Create placeholder for memoization
      put (fs' ++ [(fn', undefined)])
      -- Partially evaluate function body
      body" ← body'
      -- Define residual
      let r = ((map (fst . snd) das, t), (map fst das, body"))
      -- Replace placeholder by actual definition
      modify (update (const r) fn'))
    -- Apply the specialized function
    return (Apply fn' (map (snd . snd) das))
Let’s explore MetaLib
(A chrestomathy (a collection of software systems useful for learning) of DSL implementations)
Thank you!