# 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**

[https://pixabay.com/en/animal-cat-kitten-funny-yawning-339400/]

# The Software Languages Book

http://www.softlang.org/book

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

The underline chapter titles (see below) use Haskell systematically

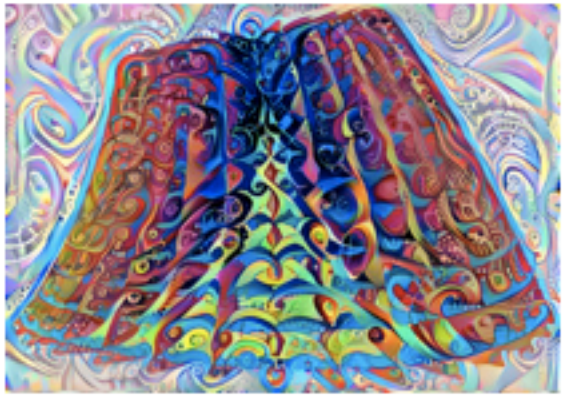
## Table of contents

# 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 Le
evaluate (Pred e) = Left (n - if n==0 t
evaluate (IsZero e) = Right (n==0) wher
evaluate (If e0 e1 e2) = evaluate (if b
```

N.B.: Functions interpret syntactic patterns.

# An ad-hoc recursive interpreter for BTL (Basic TAPL Language)

```
 Expr -> Value
JE = Right True
.SE = Right False
'o = Left 0
ıcc e) = Left (n+1) where Left n = evaluate e
'ed 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
...
```

**B**asic
**F**unctional
**P**rogramming
**L**anguage

# 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.

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

That is, **zero** evaluates to ***zero***.

---

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

That is, **succ**(*e*) evaluates **succ**(*n*), if *e* evaluates to *n*.

---

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

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

---

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

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

| **BTL example** | **General form** |
|---|---|
| $\dfrac{e \longrightarrow n}{\mathsf{succ}(e) \longrightarrow \mathsf{succ}(n)}$ [SUCC] | $\dfrac{P_1 \quad \cdots \quad P_n}{C}$ [I] |
| $\mathsf{zero} \longrightarrow \mathsf{zero}$ [ZERO] | $C$ [I] |

N.B.: A <u>rule</u> consists of <u>conclusion</u> $C$ and <u>premises</u> $P_1, \ldots, P_n$.
A rule without premises is an <u>axiom</u>.

Rules and axioms (see [I]) are labeled for ease of reference.

# Relationship claims = **judgments**

| Prefix notation | Mixfix notation | Meaning |
|---|---|---|
| *evaluate*(*e*, *v*) | $e \longrightarrow v$ | The evaluation of a BTL expression *e* results in the value *v*. |
| *execute*(*m*, *s*, *m'*) | $m \vdash s \longrightarrow m'$ | The execution of statement *s* in an <u>imperative programming language</u> transforms store *m* into store *m'*. |
| *evaluate*(*fs*, *m*, *e*, *v*) | $fs, m \vdash e \longrightarrow v$ | The evaluation of expression *e* in a <u>functional programming language</u> for the arguments *m* and function definitions *fs* results in the value *v*. |

# Disciplined use of **metavariables**

| Type | Meta-variable | Purpose |
|---|---|---|
| *expr* | $e$ | Expressions according to the abstract syntax |
| *nat* | $n$ | Natural numbers, i.e., *zero*, *succ*(*zero*), ... |
| *bool* | $b$ | Boolean values, i.e., *true* and *false* |
| *value* | $v$ | Values, i.e., Boolean values and natural numbers |

$$\frac{e \longrightarrow n}{\mathsf{succ}(e) \longrightarrow \mathsf{succ}(n)} \quad [\mathsf{SUCC}]$$

This could be any expression.

This must be a natural number!

# All (inference) rules for BTL's semantics

$$\text{true} \longrightarrow \text{true} \qquad [\text{TRUE}]$$

$$\text{false} \longrightarrow \text{false} \qquad [\text{FALSE}]$$

$$\text{zero} \longrightarrow \text{zero} \qquad [\text{ZERO}]$$

$$\frac{e \longrightarrow n}{\text{succ}(e) \longrightarrow \text{succ}(n)} \qquad [\text{SUCC}]$$

$$\frac{e \longrightarrow \text{zero}}{\text{pred}(e) \longrightarrow \text{zero}} \qquad [\text{PRED1}]$$

$$\frac{e \longrightarrow \text{succ}(n)}{\text{pred}(e) \longrightarrow n} \qquad [\text{PRED2}]$$

as seen earlier

$$\frac{e \longrightarrow \text{zero}}{\text{iszero}(e) \longrightarrow \text{true}} \qquad [\text{ISZERO1}]$$

$$\frac{e \longrightarrow \text{succ}(n)}{\text{iszero}(e) \longrightarrow \text{false}} \qquad [\text{ISZERO2}]$$

$$\frac{e_0 \longrightarrow \text{true} \qquad e_1 \longrightarrow v_1}{\text{if}(e_0, e_1, e_2) \longrightarrow v_1} \qquad [\text{IF1}]$$

$$\frac{e_0 \longrightarrow \text{false} \qquad e_2 \longrightarrow v_2}{\text{if}(e_0, e_1, e_2) \longrightarrow v_2} \qquad [\text{IF2}]$$

# **Derivation trees** as proofs of judgments

leaf nodes

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{zero} \longrightarrow \text{zero} \quad [\text{ZERO}]}{\text{iszero}(\text{zero}) \longrightarrow \text{true}}\,[\text{ISZERO1}] \qquad \dfrac{\text{zero} \longrightarrow \text{zero} \quad [\text{ZERO}]}{\text{succ}(\text{zero}) \longrightarrow \text{succ}(\text{zero})}\,[\text{SUCC}]}{\text{if}(\text{iszero}(\text{zero}), \text{succ}(\text{zero}), \text{zero}) \longrightarrow \text{succ}(\text{zero})}\,[\text{IF1}]}{\text{pred}(\text{if}(\text{iszero}(\text{zero}), \text{succ}(\text{zero}), \text{zero})) \longrightarrow \text{zero}}\,[\text{PRED2}]}$$

root

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

$$\frac{e \longrightarrow n}{\mathsf{succ}(e) \longrightarrow \mathsf{succ}(n)} \quad [\mathsf{SUCC}]$$

```
evaluate (Succ e)
  | n ← evaluate e
  , isNat n
  = Succ n
```

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 → 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)

| **Type** | **Meta-variable** | **Purpose** |
|---|---|---|
| *stmt* | $s$ | Statements according to abstract syntax |
| *expr* | $e$ | Expressions according to abstract syntax |
| *uop* | $uo$ | Unary operators according to abstract syntax |
| *bop* | $bo$ | Binary operators according to abstract syntax |
| *string* | $x$ | Variable names |
| *int* | $i$ | Integer values |
| *bool* | $b$ | Boolean values |
| *value* | $v$ | Integer and Boolean values |
| *store* | $m$ | Collections of variable name-value pairs |

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

# Statement execution (BIPL)

$$m \vdash \mathsf{skip} \longrightarrow m \qquad \text{[SKIP]}$$

$$\frac{m \vdash e \longrightarrow v}{m \vdash \mathsf{assign}(x, e) \longrightarrow m[x \mapsto v]} \qquad \text{[ASSIGN]}$$

$$\frac{m_0 \vdash s_1 \longrightarrow m_1 \qquad m_1 \vdash s_2 \longrightarrow m_2}{m_0 \vdash \mathsf{seq}(s_1, s_2) \longrightarrow m_2} \qquad \text{[SEQ]}$$

$$\frac{m \vdash e_0 \longrightarrow \mathsf{true} \qquad m \vdash s_1 \longrightarrow m'}{m \vdash \mathsf{if}(e_0, s_1, s_2) \longrightarrow m'} \qquad \text{[IF1]}$$

$$\frac{m \vdash e_0 \longrightarrow \mathsf{false} \qquad m \vdash s_2 \longrightarrow m'}{m \vdash \mathsf{if}(e_0, s_1, s_2) \longrightarrow m'} \qquad \text{[IF2]}$$

$$\frac{m \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{skip}) \longrightarrow m'}{m \vdash \mathsf{while}(e, s) \longrightarrow m'} \qquad \text{[WHILE]}$$

# Expression evaluation (BIPL)

$$m \vdash \mathsf{intconst}(i) \longrightarrow i \qquad \text{[INTCONST]}$$

$$\frac{m(x) \mapsto v}{m \vdash \mathsf{var}(x) \longrightarrow v} \qquad \text{[VAR]}$$

Auxiliary relations

$$\frac{m \vdash e \longrightarrow v \qquad \underline{unary}(uo, v) \longrightarrow v'}{m \vdash \mathsf{unary}(uo, e) \longrightarrow v'} \qquad \text{[UNARY]}$$

$$\frac{m \vdash e_1 \longrightarrow v_1 \qquad m \vdash e_2 \longrightarrow v_2 \qquad \underline{binary}(bo, v_1, v_2) \longrightarrow v'}{m \vdash \mathsf{binary}(bo, e_1, e_2) \longrightarrow v'} \qquad \text{[BINARY]}$$

# Semantics of simple **functional** programs (**BFPL** — Basic Functional Programming Language)

| **Type** | **Meta-variable** | **Purpose** |
|---|---|---|
| *programs* | $p$ | Programs according to abstract syntax |
| *functions* | $fs$ | Function collections according to abstract syntax |
| *funsig* | $sig$ | Function signatures according to abstract syntax |
| *expr* | $e$ | Expressions according to abstract syntax |
| *uop* | $uo$ | Unary operators according to abstract syntax |
| *bop* | $bo$ | Binary operators according to abstract syntax |
| *string* | $x$ | Function and argument names |
| *int* | $i$ | Integer values |
| *bool* | $b$ | Boolean values |
| *value* | $v$ | Integers and Boolean values |
| *env* | $m$ | Collections of argument name-value pairs |

There is the judgment $fs, m \vdash e \longrightarrow v$ 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

$$fs, m \vdash \mathsf{intconst}(i) \longrightarrow i \qquad \text{[INTCONST]}$$

$$fs, m \vdash \mathsf{boolconst}(b) \longrightarrow b \qquad \text{[BOOLCONST]}$$

$$\frac{\langle x, v\rangle \in m}{fs, m \vdash \mathsf{arg}(x) \longrightarrow v} \qquad \text{[ARG]}$$

$$\frac{fs, m \vdash e_0 \longrightarrow \mathsf{true} \qquad fs, m \vdash e_1 \longrightarrow v}{fs, m \vdash \mathsf{if}(e_0, e_1, e_2) \longrightarrow v} \qquad \text{[IF1]}$$

$$\frac{fs, m \vdash e_0 \longrightarrow \mathsf{false} \qquad fs, m \vdash e_2 \longrightarrow v}{fs, m \vdash \mathsf{if}(e_0, e_1, e_2) \longrightarrow v} \qquad \text{[IF2]}$$

$$\frac{fs, m \vdash e \longrightarrow v \qquad \underline{unary}(uo, v) \longrightarrow v'}{fs, m \vdash \mathsf{unary}(uo, e) \longrightarrow v'} \qquad \text{[UNARY]}$$

$$\frac{fs, m \vdash e_1 \longrightarrow v_1 \qquad fs, m \vdash e_2 \longrightarrow v_2 \qquad \underline{binary}(bo, v_1, v_2) \longrightarrow v'}{fs, m \vdash \mathsf{binary}(bo, e_1, e_2) \longrightarrow v'} \qquad \text{[BINARY]}$$

# Expression evaluation (BFPL) II/II

$$\frac{fs, m \vdash e_1 \longrightarrow v_1 \qquad \cdots \qquad fs, m \vdash e_n \longrightarrow v_n \qquad \langle x, sig, \langle\langle x_1, \ldots, x_n\rangle, e\rangle\rangle \in fs \qquad fs, [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e \longrightarrow v}{fs, m \vdash \mathsf{apply}(x, \langle e_1, \ldots, e_n\rangle) \longrightarrow v} \quad [\text{APPLY}]$$

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

$$\mathsf{zero} \longrightarrow \mathsf{zero}$$

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

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

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

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

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

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

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

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

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

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

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

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

as seen earlier

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

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

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

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

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

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

# Derivation **sequences**

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

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

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

$$\rightarrow \mathsf{zero}$$

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

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

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)

# 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.

# BTL's normal forms in Haskell

```
-- 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
```

# BTL closure in Haskell

```
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)

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

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

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

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

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

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

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

# 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.

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

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

# Denotational interpretation (Direct style)

# Denotational semantics = <u>functional</u> semantics

**Semantic domains = <u>function</u> types of meanings**

$$storeT = store \nrightarrow store \;\; // \text{ Type of store transformation}$$
$$storeO = store \nrightarrow value \;\; // \text{ Type of store observation}$$

**Semantic <u>functions</u> = mappings from syntax to semantcs**

$$\mathcal{S} \;:\; stmt \rightarrow storeT \;\; // \text{ Semantics of statements}$$
$$\mathcal{E} \;:\; expr \rightarrow storeO \;\; // \text{ Semantics of expressions}$$

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

$$\frac{m_0 \vdash s_1 \longrightarrow m_1 \qquad m_1 \vdash s_2 \longrightarrow m_2}{m_0 \vdash \mathsf{seq}(s_1, s_2) \longrightarrow m_2} \quad [\text{SEQ}]$$

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

N.B.: [SEQ] is compositional, but [WHILE] is <u>not</u>, 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

$$
\begin{aligned}
&\mathit{storeT} = \mathit{store} \nrightarrow \mathit{store} \;\; // \text{ Type of store transformation} \\
&\mathit{storeO} = \mathit{store} \nrightarrow \mathit{value} \;\; // \text{ Type of store observation}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S} \;:\; \mathit{stmt} \rightarrow \mathit{storeT} \;\; // \text{ Semantics of statements} \\
&\mathcal{E} \;:\; \mathit{expr} \rightarrow \mathit{storeO} \;\; // \text{ Semantics of expressions}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}\,[\![\mathsf{skip}]\!] &= \underline{\mathit{skip}} \\
\mathcal{S}\,[\![\mathsf{assign}(x,e)]\!] &= \underline{\mathit{assign}}\ x\ (\mathcal{E}\,[\![e]\!]) \\
\mathcal{S}\,[\![\mathsf{seq}(s_1,s_2)]\!] &= \underline{\mathit{seq}}\ (\mathcal{S}\,[\![s_1]\!])\ (\mathcal{S}\,[\![s_2]\!]) \\
\mathcal{S}\,[\![\mathsf{if}(e,s_1,s_2)]\!] &= \underline{\mathit{if}}\ (\mathcal{E}\,[\![e]\!])\ (\mathcal{S}\,[\![s_1]\!])\ (\mathcal{S}\,[\![s_2]\!]) \\
\mathcal{S}\,[\![\mathsf{while}(e,s)]\!] &= \underline{\mathit{while}}\ (\mathcal{E}\,[\![e]\!])\ (\mathcal{S}\,[\![s]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}\,[\![\mathsf{intconst}(i)]\!] &= \underline{\mathit{intconst}}\ i \\
\mathcal{E}\,[\![\mathsf{var}(x)]\!] &= \underline{\mathit{var}}\ x \\
\mathcal{E}\,[\![\mathsf{unary}(o,e)]\!] &= \underline{\mathit{unary}}\ o\ (\mathcal{E}\,[\![e]\!]) \\
\mathcal{E}\,[\![\mathsf{binary}(o,e_1,e_2)]\!] &= \underline{\mathit{binary}}\ o\ (\mathcal{E}\,[\![e_1]\!])\ (\mathcal{E}\,[\![e_2]\!])
\end{aligned}
$$

N.B.: The underlined functions are the **semantic combinators**.
They combine meanings — no syntax is involved.

# Types of semantic combinators

$$
\begin{array}{ll}
\underline{skip} & : storeT \\
\underline{assign} & : string \rightarrow storeO \rightarrow storeT \\
\underline{seq} & : storeT \rightarrow storeT \rightarrow storeT \\
\underline{if} & : storeO \rightarrow storeT \rightarrow storeT \rightarrow storeT \\
\underline{while} & : storeO \rightarrow storeT \rightarrow storeT
\end{array}
$$

$$
\begin{array}{ll}
\underline{intconst} & : int \rightarrow storeO \\
\underline{var} & : string \rightarrow storeO \\
\underline{unary} & : uo \rightarrow storeO \rightarrow storeO \\
\underline{binary} & : bo \rightarrow storeO \rightarrow storeO \rightarrow storeO
\end{array}
$$

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*

$\underline{skip}\ m\ =\ m$

// Pointwise store update

$\underline{assign}\ x\ f\ m\ =\ m[x \mapsto (f\ m)]$, if $f\ m$ is defined

// Function composition for type *storeT*

$\underline{seq}\ f\ g\ m\ =\ g\ (f\ m)$

// Select either branch for Boolean value

$$\underline{if}\ f\ g\ h\ m\ =\ \begin{cases} g\ m, \text{if } f\ m = \mathsf{true} \\ h\ m, \text{if } f\ m = \mathsf{false} \\ undefined, \text{otherwise} \end{cases}$$

N.B.: What about *while*?

# Fixed-point semantics I/II

Suppose *while* *f g* = *t*. Then, it would hold that:

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

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

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

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

$$\begin{gathered} h\ undefined \\ h\ (h\ undefined) \\ h\ (h\ (h\ undefined)) \\ \vdots \end{gathered}$$

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:

$$\underline{while}\ f\ g = fix\ h$$
$$\text{where}$$
$$h\ t = \underline{if}\ f\ (\underline{seq}\ g\ t)\ \underline{skip}$$

The fixed-point combinator is 'defined' as follows:

$$fix\ k\ =\ k\ (fix\ k)$$

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

# Denotational interpreter in Haskell: **semantic domains and functions**

```
-- 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
```

```
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**

```
skip' :: StoreT
skip' = id
assign' :: String → StoreO → StoreT
assign' x f m = insert x (f m) m
seq' :: StoreT → StoreT → StoreT
seq' = flip (.)
if' :: StoreO → StoreT → StoreT → StoreT
if' f g h m = let Right v = f m in if v then g m else h m
while' :: StoreO → StoreT → StoreT
while' f g = fix h where h t = if' f (seq' g t) skip'
intconst' :: Int → StoreO
intconst' i _ = Left i
```

...

## The fixed-point combinator at hand

```
fix :: (a → a) → a
fix f = f (fix f)
```

# 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
```

...

# 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

```haskell
-- 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

```
interpret :: Alg sto val -> Stmt -> sto -> sto
interpret a = execute
  where
    -- Compositional interpretation of statements
    execute Skip = skip' a
    execute (Assign x e) = assign' a x (evaluate e)
    execute (Seq s1 s2) = seq' a (execute s1) (execute s2)
    execute (If e s1 s2) = if' a (evaluate e) (execute s1) (execute s2)
    execute (While e s) = while' a (evaluate e) (execute s)
    -- Compositional interpretation of expressions
    evaluate (IntConst i) = intconst' a i
    evaluate (Var n) = var' a n
    evaluate (Unary o e) = unary' a o (evaluate e)
    evaluate (Binary o e1 e2) = binary' a o (evaluate e1) (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
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

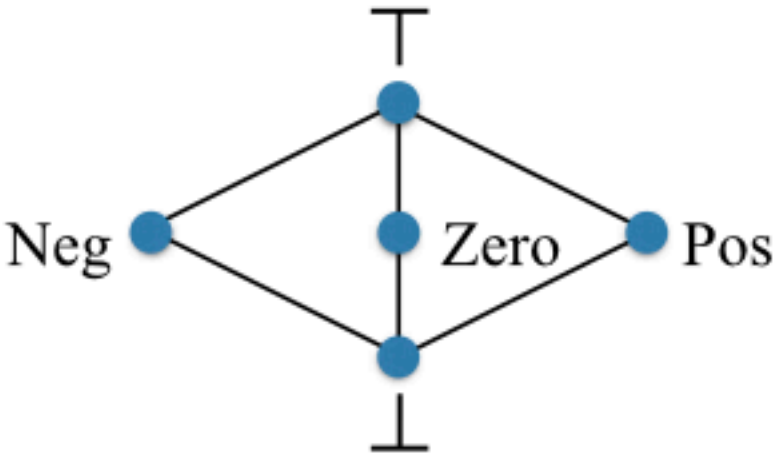
```
data Type = IntType | BoolType
type VarTypes = Map String Type
algebra :: Alg Maybe VarTypes Type
algebra = Alg {
  skip' = Just,
  assign' = \x f m -> f m >>= \t ->
    case lookup x m of
      (Just t') -> guard (t==t') >> Just m
      Nothing -> Just (insert x t m),
  seq' = flip (<=<),
  if' = \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,
  intconst' = const (const (Just IntType)),
  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 -> ...
}
```

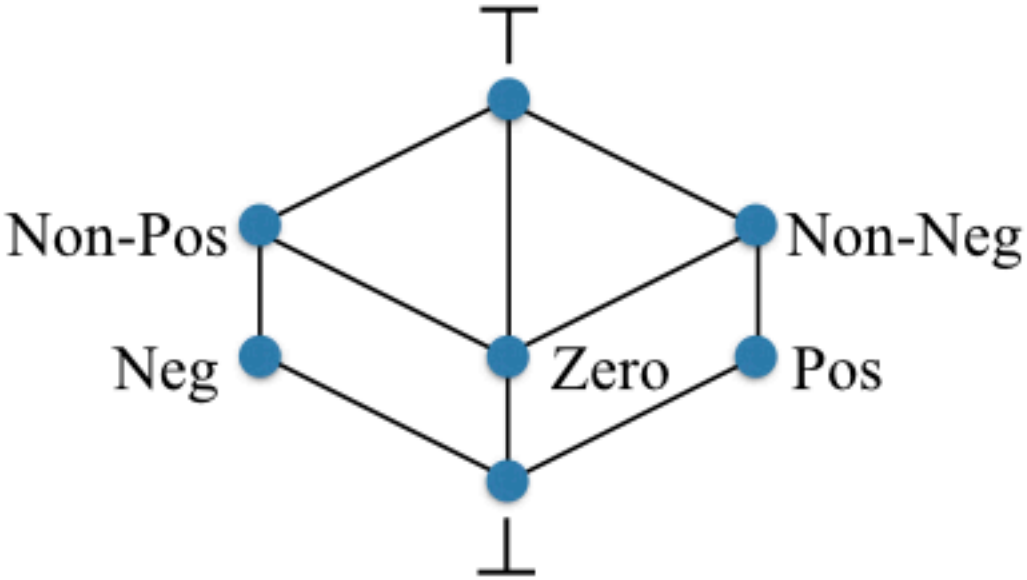
# Program analysis (sign detection) by abstract interpretation

# Optimization potential subject to sign detection

```
{
    ...
    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)]
```

```haskell
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
```

. . .

# The abstract domain for signs

# 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

```
fix' :: Eq a => ((a → a) → a → a) → (a → a) → a → a
fix' h i x = limit (iterate h i)
  where limit (b1:b2:bs) = if b1 x == b2 x then b1 x else 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)
```

Concrete syntax

```
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"]))))
```

Abstract syntax

# 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

```haskell
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
```

```haskell
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

```
► 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

```
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 (Binary o e1 e2) env = do
  r1 ← f e1 env
  r2 ← f e2 env
  case (toValue r1, toValue r2) of
    (Just v1, Just v2) → return (fromValue (bop o v1 v2))
    _ → return (Binary o r1 r2)
```

```
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))
```

Partial evaluation of function applications

Let's explore ***MetaLib***
(A chrestomathy (a collection of software systems useful for learning) of DSL implementations)

# Thank you!