The *Haskell* Road to Software Language Engineering and Metaprogramming

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

A *Haskell*-oriented overview of topics in the *Software Languages Book*

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

Topics on the Haskell road

- **Object-program representation**
- **Parsing**
- **Interpretation**
- **Software analysis**
- **Software transformation**
- **Software composition**
- **Compilation**
- **Formatting**
- **Term rewriting**
- **Quasi-quotation**
- **Attribute grammars**
- **Multi-stage programming**
- **Formal semantics**
- **Abstract interpretation**
- **Partial evaluation**

YAS' GitHub repository contains all code.
YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas

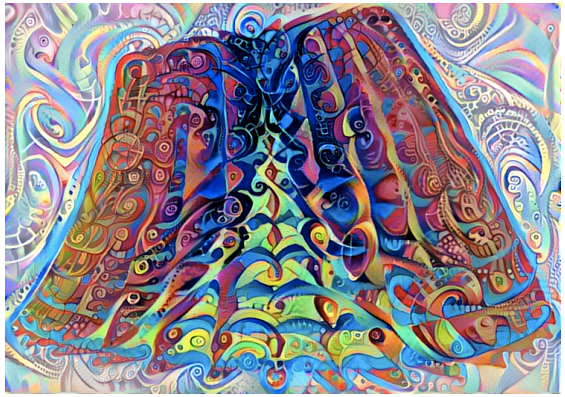
YAS' GitHub repository contains all code.

YAS (Yet Another SLR (Software Language Repository))

http://www.softlang.org/yas

The Software Languages Book develops the topics.

http://www.softlang.org/book



---

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

This presentation is more a *Haskell*-oriented **overview** of topics in the *Software Languages Book* rather than a serious attempt at explaining the underlying concepts.

Object-program representation

- **SLE/MP relevance**
  - Representation of software artifacts in metaprograms
- **Aspects**
  - Typeful representation
  - Universal representation
  - (GADTs)

Typeful representation
for a simple expression language

```haskell
data Expr
  = TRUE -- True taken by Haskell Prelude
  | FALSE -- False taken by Haskell Prelude
  | Zero
  | Succ Expr
  | Pred Expr
  | IsZero Expr
  | If Expr Expr Expr
```

```haskell
sampleExpr :: Expr
sampleExpr = Pred (If (IsZero Zero) (Succ Zero) Zero)
```

Universal representation
for a simple expression language

```
data TermRep = TermRep ConstrId [TermRep]
type ConstrId = String
```

```
sampleExpr :: TermRep
sampleExpr =
  TermRep "Pred" [
    TermRep "If" [
      TermRep "IsZero" [TermRep "Zero" []],
      TermRep "Succ" [TermRep "Zero" []],
      TermRep "Zero" [] ]]
```

Parsing

- **SLE/MP relevance**
  - Parse text into trees
  - ‘Text to model’ in MDE
- **Aspects**
  - Parsing algorithms
  - Recursive descent
  - Parser combinators

Algorithm for top-down acceptance

```
accept :: [Rule] → String → Bool
accept g = steps g [N s]
  where
    -- Retrieve start symbol
    ((_, s, _):_) = g

steps :: [Rule] → [GSymbol] → String → Bool
-- Acceptance succeeds (empty stack, all input consumed)
steps _ [] [] = True
-- Consume terminal at top of stack from input
steps g (T t:z) (t':i) | t==t' = steps g z i
-- Expand a nonterminal; try different alternatives
steps g (N n:z) i = or (map (λ rhs → steps g (rhs++z) i) rhss)
  where
    rhss = [ rhs | (_, n', rhs) ← g, n == n' ]
-- Otherwise parsing fails
steps _ _ _ = False
```

```
type Grammar = [Rule]
type Rule = (Label, Nonterminal, [GSymbol])
data GSymbol = T Terminal | N Nonterminal
type Label = String
type Terminal = Char
type Nonterminal = String
```

N.B.: This simple algorithm is sufficient to discuss issues of backtracking and termination.

Recursive descent
for parsing binary numbers

```haskell
-- Accept and enforce complete input consumption
accept :: String → Bool
accept i = case number i of
  Just [] → True
  _ → False

-- Functions for nonterminals
number, bits, bit, rest :: String → Maybe String

-- [number] number : bits rest ;
number i = bits i >>=rest

-- [single] bits : bit ;
-- [many] bits : bit bits ;
bits i = many `mplus` single
  where
    single = bit i
    many = bit i >>=bits
```

```haskell
-- [zero] bit : '0' ;
-- [one] bit : '1' ;
bit i = zero `mplus` one
  where
    zero = match '0' i
    one = match '1' i

-- [integer] rest : ;
-- [rational] rest : '.' bits ;
rest i = rational `mplus` integer
  where
    integer = Just i
    rational = match '.' i >>=bits

-- Match a terminal (a character)
match :: Char → String → Maybe String
match t (t':i) | t == t' = Just i
match _ _ = Nothing
```

Using **parser combinators** for an acceptor of finite state machines

```
fsm = many state
state =
      optional (reserved "initial")
   >> reserved "state"
   >> stateid
   >> braces (many transition)
transition =
      event
   >> optional (op "/" >> action)
   >> optional (op "->" >> stateid)
   >> semi
stateid = name
event = name
action = name
```

```
initial state locked {
  ticket/collect -> unlocked;
  pass/alarm -> exception;
}
state unlocked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

Interpretation

- **SLE/MP relevance**
  - Language implementation
  - Definitional interpreters
- **Aspects**
  - Recursive evaluation
  - Execution with stores
  - Evaluation with environments
  - Stepwise execution

Recursive evaluation
for a simple expression language

```
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
```

Execution with stores

for a simple imperative programming language

```haskell
-- Results of expression evaluation
type Value = Either Int Bool
-- Stores as maps from variable names to values
type Store = Map String Value
-- Execution of statements
execute :: Stmt → Store → Store
execute Skip m = m
execute (Assign x e) m = insert x (evaluate e m) m
execute (Seq s1 s2) m = execute s2 (execute s1 m)
execute (If e s1 s2) m = execute (if b then s1 else s2) m where Right b = evaluate e m
execute (While e s) m = execute (If e (Seq s (While e s)) Skip) m

-- Evaluation of expressions
evaluate :: Expr → Store → Value
evaluate (IntConst i) _ = Left i
evaluate (Var x) m = m!x
evaluate (Unary o e) m = uop o (evaluate e m)
evaluate (Binary o e1 e2) m = bop o (evaluate e1 m) (evaluate e2 m)
```

Evaluation with environments

for a simple functional programming language

```haskell
-- Results of expression evaluation
type Value = Either Int Bool
-- Environments as maps from argument names to values
type Env = Map String Value
-- 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)
```

Stepwise execution
for a DSML for finite state machines

```
-- FSM simulation starting from initial state
simulate :: Fsm → Input → Output
simulate (Fsm ss) xs = snd (foldl makeTransition (getInitial, []) xs)
  where
    -- Look up initial state
    getInitial :: StateId
    getInitial = ini
      where [State _ ini _] = [ s | s@(State initial _ _) ← ss, initial ]

    -- Process event; extent output
    makeTransition :: (StateId, Output) → Event → (StateId, Output)
    makeTransition (source, as) x = (target, as ++ maybeToList a)
      where (Transition _ a target) = getTransition source x

    -- Look up transition
    getTransition :: StateId → Event → Transition
    getTransition sid x = t
      where
        [t] = [ t | t@(Transition x' _ _) ← ts, x == x' ]
        [(State _ _ ts)] = [ s | s@(State _ sid' _) ← ss, sid == sid' ]
```

```
data Fsm = Fsm { getStates :: [State] }
data State = State {
    getInitial :: Initial,
    getId :: StateId,
    getTransitions :: [Transition]
  }
data Transition = Transition {
    getEvent :: Event,
    getAction :: (Maybe Action),
    getTarget :: StateId
  }
type Initial = Bool
type StateId = String
```

Software analysis

- **SLE/MP relevance**
  - Implementation of type systems of PLs
  - Static semantics of DSLs and DSMLs
  - Software reverse engineering and program comprehension
- **Aspects**
  - Type checking based on Boolean constraints
  - (Type checking with error messages)
  - Well-formedness checking
  - Fact extraction for software metrics

Type checking
for a simple expression language

```
-- Types of expressions
data Type = NatType | BoolType

-- Well-typedness of expressions
wellTyped :: Expr → Bool
wellTyped e = case typeOf e of { Just _ → True; Nothing → False }

-- Types of expressions
typeOf :: Expr → Maybe Type
typeOf TRUE = Just BoolType
typeOf FALSE = Just BoolType
typeOf Zero = Just NatType
typeOf (Succ e) = case typeOf e of { Just NatType → Just NatType; _ → Nothing }
typeOf (Pred e) = case typeOf e of { Just NatType → Just NatType; _ → Nothing }
typeOf (IsZero e) = case typeOf e of { Just NatType → Just BoolType; _ → Nothing }
typeOf (If e0 e1 e2) =
  case typeOf e0 of
    (Just BoolType) →
      case (typeOf e1, typeOf e2) of
        (Just t1, Just t2) → if t1==t2 then Just t1 else Nothing
        _ → Nothing
```

Well-formedness checking
for a DSML for finite state machines

```
data Fsm = Fsm { getStates :: [State] }
data State = State {
    getInitial :: Initial,
    getId :: StateId,
    getTransitions :: [Transition]
  }
data Transition = Transition {
    getEvent :: Event,
    getAction :: (Maybe Action),
    getTarget :: StateId
  }
type Initial = Bool
type StateId = String
```

```
check :: Fsm → Bool
check fsm = and (map ($ fsm) [
    distinctStateIds,
    singleInitialState,
    resolvableTargetStates,
    deterministicTransitions,
    reachableStates ] )

distinctStateIds :: Fsm → Bool
distinctStateIds (Fsm ss) = sids == nub sids
  where sids = [ sid | (State _ sid _) ← ss ]

singleInitialState :: Fsm → Bool
singleInitialState (Fsm ss) = length inis == 1
  where inis = [ sid | s@(State initial sid _) ← ss, initial ]

resolvableTargetStates :: Fsm → Bool
resolvableTargetStates (Fsm ss) = and (map (λ(State _ _ ts) → and (map f ts)) ss)
  where f (Transition _ _ target) =
            not (null [ s | s@(State _ source _) ← ss, source == target ])
```

...

Fact extraction for software metrics

for operator frequency
for a simple imperative programming language

```
▶ ops euclideanDiv
fromList [("Add",1),("Geq",1),("Sub",1)]
```

```
ops :: Stmt → Map String Int
ops s = foldr (λ o m → insertWith (+) o 1 m) empty os
  where
    os = everything (++) ([] `mkQ` f) s
    f (Unary o _) = [showConstr (toConstr o)]
    f (Binary o _ _) = [showConstr (toConstr o)]
    f _ = []
```

Software transformation

- **SLE/MP relevance**
  - Language implementation
  - Software re-engineering
- **Aspects**
  - Refactoring (e.g., renaming)
  - (Program optimization)

Renaming refactoring

for state ids for finite state machines

Input

```
initial state locked {
    ticket / collect -> unlocked;
    pass / alarm -> exception;
}
state unlocked {
    ticket / eject;
    pass -> locked;
}
…
```

Output

```
initial state closed {
    ticket / collect -> open;
    pass / alarm -> exception;
}
state open {
    ticket / eject;
    pass -> closed;
}
…
```

```
pre, post :: StateId → StateId → Fsm → Bool
pre i i' x = elem i (states x) && not (elem i' (states x))
post i i' y = not (elem i (states y)) && elem i' (states y)
states :: Fsm → [StateId]
states fsm =
  concatMap (λ s →
    getId s : map getTarget (getTransitions s))
      (getStates fsm)
rename :: StateId → StateId → Fsm → Maybe Fsm
rename i i' x = do
    guard $ pre i i' x
    guard $ post i i' y
    return y
  where
    y = Fsm (map perState (getStates x))
    perState s =
      State
        (getInitial s)
        (if getId s == i then i' else getId s)
        (map perTransition (getTransitions s))
    perTransition t =  ...
```

Renaming refactoring for state ids for finite state machines

Software composition

- **SLE/MP relevance**
  - Modularity
- **Aspects**
  - Model composition (e.g., merging)

Model composition
for finite state machines

```
initial state locked {
    ticket / collect -> unlocked;
}
state unlocked {
    ticket / eject;
    pass -> locked;
}
```

+

```
state locked {
   pass / alarm -> exception;
}
state exception {
   ticket / eject;
   pass;
   mute;
   release -> locked;
}
```

```
initial state locked {
    ticket / collect -> unlocked;
    pass / alarm -> exception;
}
state unlocked {
    ticket / eject;
    pass -> locked;
}
state exception {
   ticket / eject;
   pass;
   mute;
   release -> locked;
}
```

Model composition for finite state machines

```
ok :: Fsm → Bool
ok fsm = and $ map ($fsm) [
  zeroOrOneInitialState,
  distinctStateIds,
  resolvableTargetStates,
  deterministicTransitions ]
merge :: Fsm → Fsm → Maybe Fsm
merge x y = do
    guard $ ok x && ok y
    let z = fromMap (unionWith f (toMap x) (toMap y))
    guard $ ok z
    return z
  where
    -- Per-state composition
    f sx sy = State
        (getInitial sx || getInitial sy)
        (getId sx)
        (getTransitions sx ++ getTransitions sy)
    toMap = fromList . map (λs → (getId s, s)) . getStates
    fromMap = Fsm . map snd . toList
```

...

Compilation

- **SLE/MP relevance**
  - Language implementation
- **Aspects**
  - Assembly language
  - Machine language
  - Code generation

Assembly language

```
euclideanDiv = [
    Const 14, Store "x", -- x = 14;
    Const 4, Store "y", -- y = 4;
    Const 0, Store "q", -- q = 0;
    Load "x", Store "r", -- r = x;
    Label "0", -- Beginning of while loop
    Load "r", Load "y", Geq, -- (r >= y)
    Not, CJump "1", -- Skip while loop
    Load "r", Load "y", Sub, Store "r", -- r = r - y;
    Load "q", Const 1, Add, Store "q", -- q = q + 1;
    Jump "0", -- Next iteration of loop
    Label "1" -- Label to goto when skipping loop
 ]
```

Machine language

```
euclideanDiv = [Const 14,Store 0,Const 4,Store 1,Const 0,Store 2,Load 0,Store 3,Load 3,
    Load 1,Geq,Not,CJump 22,Load 3,Load 1,Sub,Store 3,Load 2,Const 1,Add,Store 2,
    Jump 8]
```

Translation to assembler code

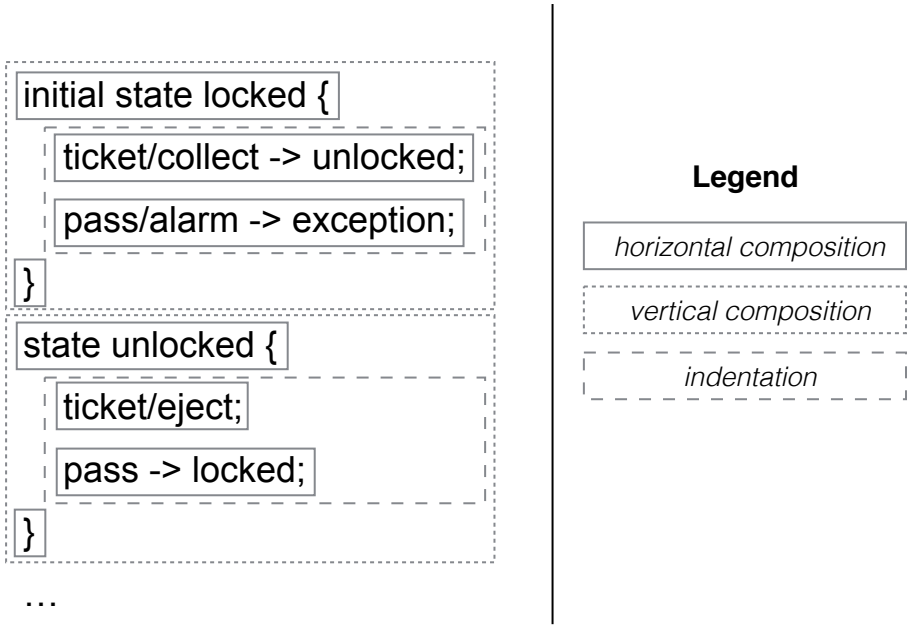
```
compile :: Stmt → [Instr]
compile s = fst (stmt s 0)

stmt :: Stmt → Int → ([Instr], Int)
stmt Skip l = ([], l)
stmt (Assign x e) l = (expr e ++ [Store x], l)
stmt (Seq s1 s2) l0 =
  let
    (zs1, l1) = stmt s1 l0
    (zs2, l2) = stmt s2 l1
  in (zs1 ++ zs2, l2)
stmt (If e s1 s2) l0 =
 let l1 = l0+1
      (zs1, l2) = stmt s1 (l1+1)
      (zs2, l3) = stmt s2 l2
 in (expr e
      ++ (CJump (show l0) : zs2)
      ++ (Jump (show l1) : (Label (show l0) : zs1))
      ++ [Label (show l1)], l3)
stmt (While e s) l0 = ...
```

Formatting

- **SLE/MP relevance**
  - Source code generation
  - Model-to-text in MDE
- **Aspects**
  - Pretty printing combinators
  - Template processing

Pretty printing combinators
used for formatting FSMs




```
fsm :: Fsm → Doc
fsm (Fsm ss) = vcat (map state ss)

state :: State → Doc
state (State initial source ts) =
  (if initial then text "initial" else empty)
  <+> text "state"
  <+> text source
  <+> text "{"
  $$ nest 2 (vcat (map (transition source) ts))
  $$ text "}"
```

```
transition :: String → Transition → Doc
transition source (Transition ev ac target) =
  text ev
  <> maybe empty (λ ac'→  text "/" <> text ac') ac
  <+> (if source == target
          then empty
          else text "->" <+> text target)
  <> text ";"
```

Template processing used for formatting FSMs

```
templates :: STGroup String
templates = groupStringTemplates [
    ("main", newSTMP "$fsm.states:state(); separator='\n'$"),
    ("state", newSTMP $ unlines [
          "$if(it.initial)$initial $endif$state $it.stateid$ {",
          "$it.transitions:transition(); separator='\n'$",
          "}"
        ]
    ),
    ("transition", newSTMP (
              " $it.event$\
              \$if(it.action)$/$it.action$$endif$\
              \$if(it.target)$ -> $it.target$$endif$\
              \;"
        )
    )
  ]

format :: Fsm → String
format fsm =
    let Just t = getStringTemplate "main" templates
      in render $ setAttribute "fsm" fsm t
```

Term rewriting

- **SLE/MP relevance**
  - Software transformation
- **Aspects**
  - Rewrite rules
  - Rewrite systems
  - Strategic programming

Laws of an expression language

$$
\begin{array}{lll}
X + 0 & = X & \text{-- Unit of addition} \\
X * 1 & = X & \text{-- Unit of multiplication} \\
X * 0 & = 0 & \text{-- Zero of multiplication} \\
X + Y & = Y + X & \text{-- Commutativity of addition} \\
X * Y & = Y * X & \text{-- Commutativity of multiplication} \\
(X + Y) + Z & = X + (Y + Z) & \text{-- Associativity of addition} \\
(X * Y) * Z & = X * (Y * Z) & \text{-- Associativity of multiplication} \\
(X * Y) + (X * Z) & = X * (Y + Z) & \text{-- Distributivity}
\end{array}
$$

Implemented **rewrite rules** for simplification of expressions

```
simplify :: Expr → Maybe Expr
simplify (Binary Add x (IntConst 0)) = Just x
simplify (Binary Mul x (IntConst 1)) = Just x
simplify (Binary Mul x (IntConst 0)) = Just $ IntConst 0
simplify _ = Nothing
```

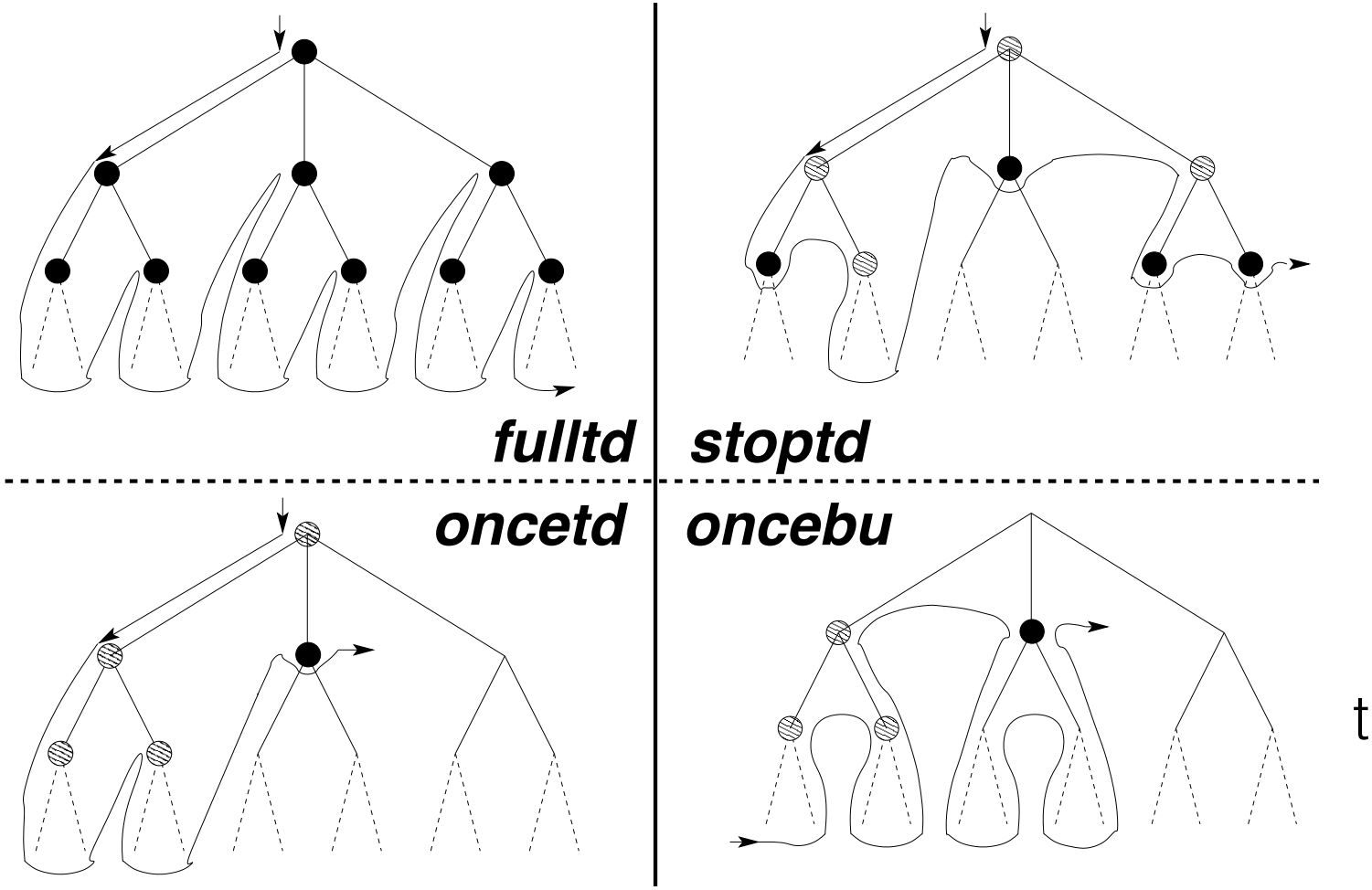
```
▶ simplify (Binary Add (Var "a") (IntConst 0))
Just (Var "a")
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ simplify (IntConst 42)
Nothing
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ simplify (Binary Add (Var "a") (Binary Add (Var "b") (IntConst 0)))
Nothing
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ simplify (Binary Add (IntConst 0) (Var "a"))
Nothing
```

Adding commutativity

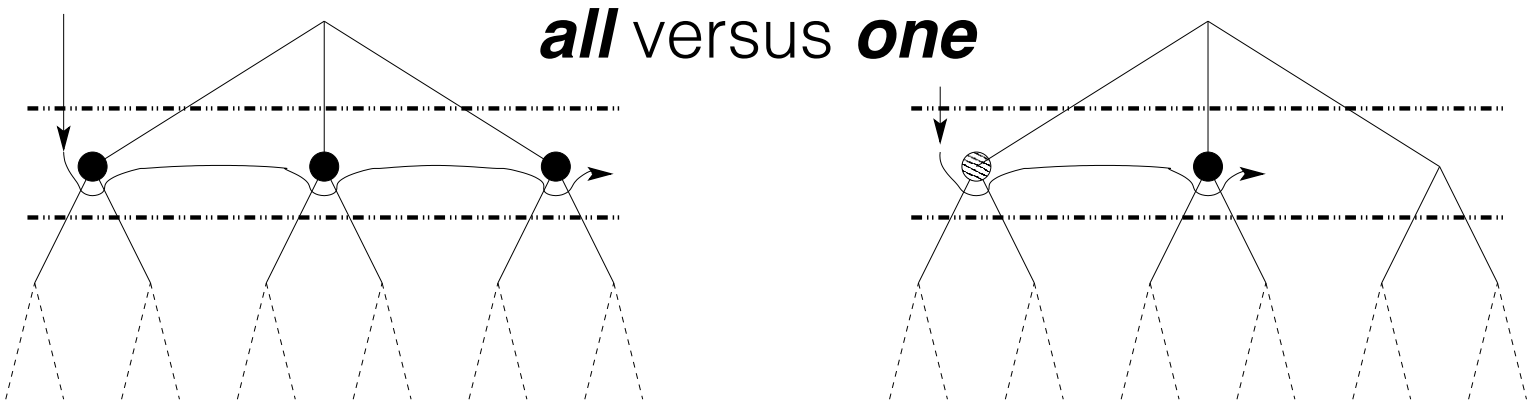
```
commute :: Expr → Maybe Expr
commute (Binary Add x y) = Just $ Binary Add y x
commute (Binary Mul x y) = Just $ Binary Mul y x
commute _ = Nothing
```

```
simplify' x = simplify x `mplus` commute x >>=simplify
```

```
▶ simplify' (Binary Add (IntConst 0) (Var "a"))
Just (Var "a")
```



Strategic programming: traversal schemes and one-layer traversal



A strategy library in Haskell

```
-- Strategic traversal schemes
fulltd s = s `sequ` all (fulltd s)
fullbu s = all (fullbu s) `sequ` s
stoptd s = s `choice` all (stoptd s)
oncetd s = s `choice` one (oncetd s)
oncebu s = one (oncebu s) `choice` s
innermost s = repeat (oncebu s)

-- Basic strategy combinators
s1 `sequ` s2 = λx → s1 x >>=s2 -- monadic function composition
s1 `choice` s2 = λx → s1 x `mplus` s2 x -- monadic choice
all = ... -- depends on type class Data
one = ... -- depends on type class Data

-- Helper strategy combinators
try s = s `choice` return -- recover from failure
vary s v = s `choice` (v `sequ` s) -- preprocess term, if necessary
repeat s = try (s `sequ` repeat s) -- repeat strategy until failure

-- Strategy builders
orFail f = const mzero `extM` f -- fail for all other types
orSucceed f = return `extM` f' -- id for all other types
 where f' x = f x `mplus` return x -- id in case of failure
```

Strategy application in Haskell

```
-- The expression "a + b * 0" with simplification potential
▶ let e1 = Binary Add (Var "a") (Binary Mul (Var "b") (IntConst 0))
-- The expression "((a * b) * c) * d" associated to the left
▶ let e2 = Binary Mul (Binary Mul (Binary Mul (Var "a") (Var "b")) (Var "c")) (Var "d")
-- The expression "0 + a" requiring commutativity for simplification
▶ let e3 = Binary Add (IntConst 0) (Var "a")
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Incomplete simplification with fulltd
▶ fulltd (orSucceed simplify) e1
Binary Add (Var "a") (IntConst 0)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Complete simplification with fullbu
▶ fullbu (orSucceed simplify) e1
Var "a"
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Incomplete association to the right with fullbu
▶ fullbu (orSucceed associate) e2
Binary Mul (Var "a") (Binary Mul (Binary Mul (Var "b") (Var "c")) (Var "d"))
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Complete association to the right with innermost
▶ innermost (orFail associate) e2
Binary Mul (Var "a") (Binary Mul (Var "b") (Binary Mul (Var "c") (Var "d")))
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Apply simpification module commutativity
▶ vary (orFail simplify) (orFail commute) e3
Var "a"
```

Quasi-quotation

- **SLE/MP relevance**
  - Language embedding
- **Aspects**
  - Parser embedding
  - Metavariables or splices

Parser embedding

for finite state machines in Haskell programs

```
turnstileFsm :: Fsm
turnstileFsm = [fsml|
        initial state locked {
                ticket / collect → unlocked;
                pass / alarm → exception;
        }
        state unlocked {
                ticket / eject;
                pass → locked;
        }
        state exception {
                ticket / eject;
                pass;
                mute;
                release → locked;
        }
|]
```

A simple quasi-quoter for finite state machines in Haskell programs

```
fsml :: QuasiQuoter
fsml = QuasiQuoter
    { quoteExp = quoteFsmlExp
    , quotePat = undefined
    , quoteType = undefined
    , quoteDec = undefined
    }

quoteFsmlExp :: String → Q Exp
quoteFsmlExp str = do
    x ← parseQ fsm str
    case check x of
      [] → dataToExpQ (const Nothing) x
      errs → error $ unlines errs
```

Splices for metavariables

in simplification rules for expressions

```
-- Laws on expressions
x + 0 = x
x * 1 = x
x * 0 = 0
```

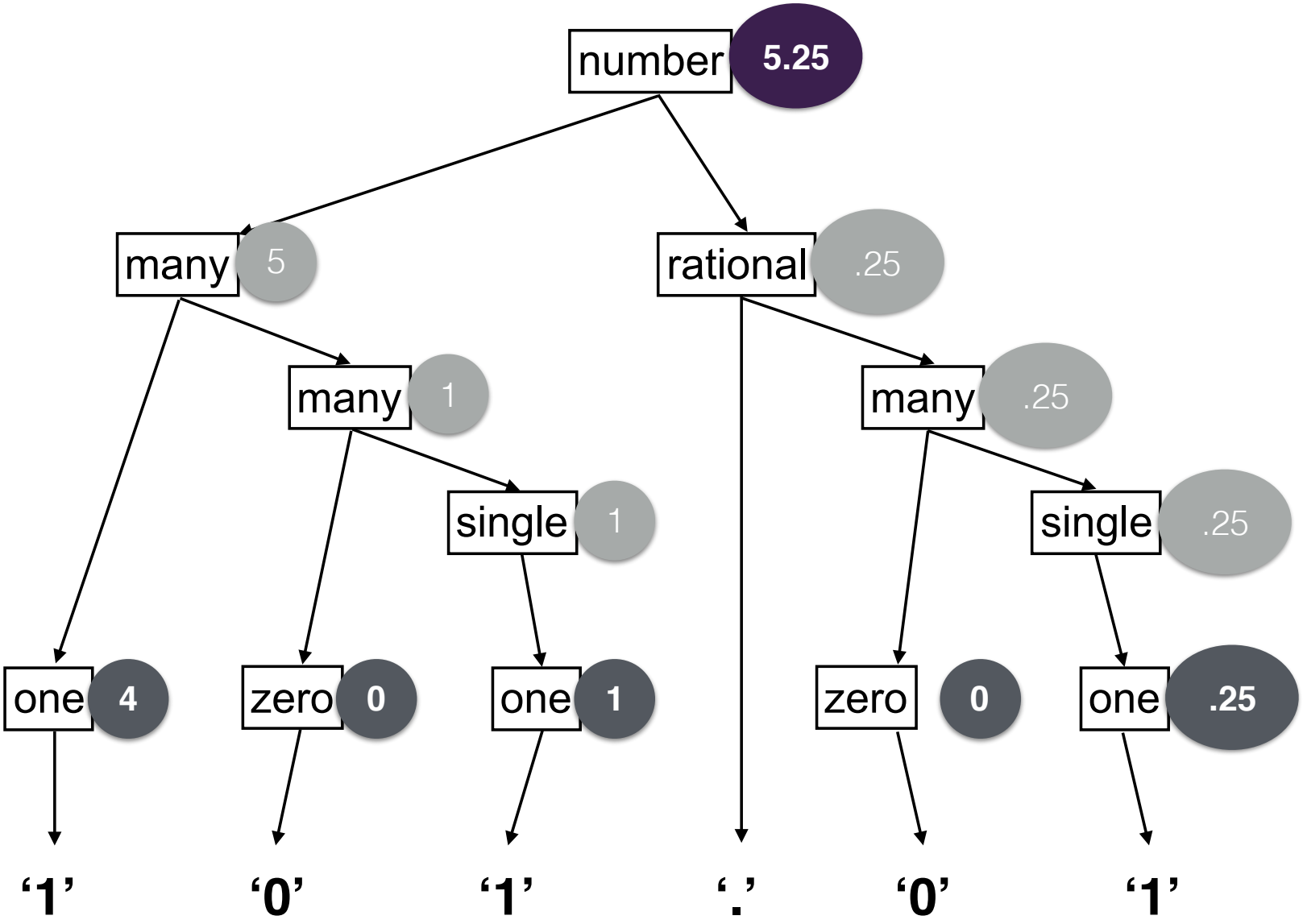
```
-- Implementation based on abstract object syntax
simplify :: Expr -> Maybe Expr
simplify (Binary Add x (IntConst 0)) = Just x
simplify (Binary Mul x (IntConst 1)) = Just x
simplify (Binary Mul x (IntConst 0)) = Just $ IntConst 0
simplify _ = Nothing
```

```
-- Implementation based on concrete object syntax
simplify :: Expr -> Maybe Expr
simplify [el| $x + 0 |] = Just [el| $x |]
simplify [el| $x * 1 |] = Just [el| $x |]
simplify [el| $x * 0 |] = Just [el| 0 |]
simplify _ = Nothing
```

Attribute grammars

- **SLE/MP relevance**
  - Program analysis and translation
- **Aspects**
  - Attributed CSTs
  - Attribute dependencies
  - Encoding in lazy functional programming

An attributed CST for a binary number



An **attribute grammar** for binary-to-decimal number conversion

[number] number : bits rest ;

$$bits.Pos = bits.Len - 1$$
$$number.Val = bits.Val + rest.Val$$

[single] bits : bit ;

$$bit.Pos = bits.Pos$$
$$bits.Val = bit.Val$$
$$bits.Len = 1$$

[many] bits0 : bit bits1 ;

$$bit.Pos = bits0.Pos$$
$$bits1.Pos = bits0.Pos - 1$$
$$bits0.Val = bit.Val + bits1.$$
$$bits0.Len = bits1.Len + 1$$

[zero] bit : '0' ;

$$bit.Val = 0$$

[one] bit : '1' ;

$$bit.Val = 2^{bit.Pos}$$

[integer] rest : ;

$$rest.Val = 0$$

[rational] rest : '.' bits ;

$$rest.Val = bits.Val$$
$$bits.Pos = -1$$

```
number :: Number → Float
number (Number bs r) = val0
  where
    (len1, val1) = bits bs pos1
    pos1 = len1 − 1
    val2 = rest r
    val0 = val1 + val2

bits :: Bits → Int → (Int, Float)
bits (Single b) pos = (1, bit b pos)
bits (Many b bs) pos0 = (len0, val0)
  where
    val1 = bit b pos0
    (len1, val2) = bits bs pos1
    pos1 = pos0 − 1
    len0 = len1 + 1
    val0 = val1 + val2

bit :: Bit → Int → Float
bit Zero _pos = 0
bit One pos = 2^^pos

rest :: Rest → Float
...
```

Binary-to decimal-number conversion in Haskell

Multi-stage programming

- **SLE/MP relevance**
  - Generative programming
- **Aspects**
  - Use-case: inlining
  - Quasi-quotes and splicing
  - More typeful staging

Use-case inlining

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

Wanted: a ‘power 3’ function which involves no recursion.
Instead, it should be equivalent to `\x -> x * x * x * 1`.

Quasi-quotes and splicing
with Template Haskell

```
power :: Int → Q Exp → Q Exp
power n x =
  if n==0
    then [| 1 |]
    else [| $x * $(power (n−1) x) |]

mk_power :: Int → Q Exp
mk_power n = [| λx → $(power n [| x |]) |]
```

More typeful staging

```
power :: Int → Q (TExp Int) → Q (TExp Int)
power n x =
  if n==0
    then [|| 1 ||]
    else [|| $$x * $$(power (n−1) x) ||]

mk_power :: Int → Q (TExp (Int → Int))
mk_power n = [|| λx → $$(power n [|| x ||]) ||]
```

Formal semantics

- **SLE/MP relevance**
  - Programming language theory meets SLE and MP
- **Aspects**
  - Big-step operational semantics
  - Small-step operational semantics
  - Compositional (denotational) semantics

Big-step operational semantics
for a simple expression language

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

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

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

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

```
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
```

Small-step operational semantics
for a simple expression language

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

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

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

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

```
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
```

Compositional (denotational) semantics

for a simple imperative programming language

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

```
-- 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
```

```
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)
```

Combinators of denotational semantics

// 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

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

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

```
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'
```

Abstract interpretation

- **SLE/MP relevance**
  - Program analysis
- **Aspects**
  - Abstract domains
  - Semantic algebras

Abstract domain
for signs instead of numbers

+	Neg	Zero	Pos	?
Neg	Neg	Neg	?	?
Zero	Neg	Zero	Pos	?
Pos	?	Pos	Pos	?
?	?	?	?	?

```
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
  ...
```

Signature of semantic 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
}
```

Partial evaluation

- **SLE/MP relevance**
  - Program optimization
- **Aspects**
  - Program specialization

```
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, getValue r) | (n, (_, r)) ← ntrs, isValue r ]
  let das = [ (n, (t, r)) | (n, (t, r)) ← ntrs, not (isValue 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))
```

Specialization
of a function
application

http://www.softlang.org/book

Wrap-up

http://www.softlang.org/yas

- What are your ‘related’ experiences with teaching?
- **Please get in touch**, if you want to use the material or you can contribute to it or you may provide feedback: http://www.softlang.org/book