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
## 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
  - Metaprogramming in Prolog
- Postface
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

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

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

Let us also look at more complex illustrations of typeful representation. That is, we deal with abstract syntaxes that involve more than just one sort; also, we exercise ESL expressiveness for tuples, lists, and primitive types. Here are the Haskell-based abstract syntaxes of BIPL (Basic Imperative Programming Language), BFPL (Basic Functional Programming Language), and FSML (Finite State Machine Language).
Universal representation for a simple expression language

```haskell
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" []]]
```

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

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

steps :: [Rule] → [GSymbol] → String → Bool
  where
    -- 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.

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Recursive descent
for parsing binary numbers

-- 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

-- [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

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Using **parser combinators**
for an acceptor of finite state machines

```haskell
fsm = many state
state =
    optional (reserved "initial")
  >>= reserved "state"
  >>= stateid
  >>= braces (many transition)
transition =
    event
  >>= optional (op "/" >>= action)
  >>= optional (op "\rightarrow" >>= 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

```haskell
import Data.Either

data Value = Left Int | Right 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

--- Results of expression evaluation

```haskell
-- 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
execute (While e s) m = execute (If e (Seq s (While e s)) Skip) m
```

--- Evaluation of expressions

```haskell
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)
```

---

© 2017 Ralf Lämmel, Software Languages Team [http://www.softlang.org/], University of Koblenz-Landau. All rights reserved.
Evaluation with environments
for a simple functional programming language

--- 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 :: Stateld
getInitial = ini
    where [State _ ini _] = [ s | s@(State initial _ _) ← ss, initial ]

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

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

```haskell
data Type = NatType | BoolType
```

--- Well-typedness of expressions

```haskell
wellTyped :: Expr → Bool
wellTyped e = case typeOf e of { Just _ → True; Nothing → False }
```

--- Types of expressions

```haskell
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 }
```

```haskell
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

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

\[
\begin{align*}
&\textbf{ops euclideanDiv} \\
&\textbf{fromList} \ [("Add",1),("Geq",1),("Sub",1)] \\
&\textbf{ops} : \textbf{Stmt} \rightarrow \textbf{Map} \ \textbf{String} \ \textbf{Int} \\
&\textbf{ops} \ s = \textbf{foldr} \ (\lambda \textbf{o} \ \textbf{m} \rightarrow \textbf{insertWith} \ (+) \ \textbf{o} \ \textbf{1} \ \textbf{m}) \ \textbf{empty} \ \textbf{os} \\
&\text{where} \\
&\quad \textbf{os} = \textbf{everything} \ (++) \ ([] \ \text{`mkQ` } f) \ s \\
&\quad f \ (\textbf{Unary} \ \textbf{o} \ _\ _) = [\textbf{showConstr} \ (\textbf{toConstr} \ \textbf{o})] \\
&\quad f \ (\textbf{Binary} \ \textbf{o} \ _\ _\ _) = [\textbf{showConstr} \ (\textbf{toConstr} \ \textbf{o})] \\
&\quad f \ _\ _ = []
\end{align*}
\]
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;
}

...

Laws
Identify transformation laws which may be helpful in better understanding the transformation. For instance, one may identify the inverse of a transformation. For instance, renaming is obviously inversive while optimizations generally are not inversive.

Implementation
Implement the actual transformation.

5.4.2 Refactoring
A refactoring is a transformation that changes the program's 'design' without changing its behavior [Fow99, MT04]. Refactorings can be automated (see Opdyke's seminal work [Opd92]) and they make sense across all kinds of software languages [Läm02]. That is, renaming can also be applied to models, grammars, schemas, systems, etc.

We are going to focus here on renaming as a refactoring. We demonstrate renaming for state-id renaming for FSML and variable renaming for BIPL. The FSML instance is illustrated in Figure 1; the illustration provides a positive test case for renaming such that state id locked is renamed to closed and unlocked is renamed to open. The Haskell implementation of the pre- and postconditions follows.

Illustration 5.28 (Pre-/postconditions for state-id renaming).
rename :: StateId

FSM while replacing state ids systematically.

rename i

can be reverted as follows; we glance over the detail here that the

Devise negative test cases covering the di

Exercise 5.17

stronger formulation.

Argue that the stated postcondition is not as strong as possible. Attempt a

Exercise 5.16

(a declaration thereof) or in the target location of a transition.

hold, then the postcondition is implied.

anymore) and

renaming state id

states :: Fsm

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 → [Stateld]

states fsm =

concatMap (λ s →
  getId s : map getTarget (getTransitions s))
  (getStates fsm)

rename :: Stateld → Stateld → 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 = ...
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 locked {
    pass / alarm -> exception;
}
state exception {
    ticket / eject;
    pass;
    mute;
    release -> locked;
}
```
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

Model composition for finite state machines
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]

Illustration 5.16 (Euclidian division in BML)

Illustration 5.17 (Abstract syntax of BML)
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

Initial state locked:
- ticket/collect -> unlocked;
- pass/alarm -> exception;

State unlocked:
- ticket/eject;
- pass -> locked;

...
Template processing used for formatting FSMs

templates :: STGroup String
templates = groupStringTemplates [  
  ("main", newSTMP "$fsm.states:state(); separator='\n'"),
  ("state", newSTMP $ unlines [  
    "$it.initial$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

\[ X + 0 = X \quad \text{-- Unit of addition} \]
\[ X \times 1 = X \quad \text{-- Unit of multiplication} \]
\[ X \times 0 = 0 \quad \text{-- Zero of multiplication} \]
\[ X + Y = Y + X \quad \text{-- Commutativity of addition} \]
\[ X \times Y = Y \times X \quad \text{-- Commutativity of multiplication} \]
\[ (X + Y) + Z = X + (Y + Z) \quad \text{-- Associativity of addition} \]
\[ (X \times Y) \times Z = X \times (Y \times Z) \quad \text{-- Associativity of multiplication} \]
\[ (X \times Y) + (X \times Z) = X \times (Y + Z) \quad \text{-- Distributivity} \]
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")
In this paper, we introduce a novel approach to strategic programming, focusing on traversal schemes and one-layer traversal.

The diagrams illustrate the concepts of full traversal (fulltd) and stop traversal (stoptd), as well as one traversal (onced) and one traversal with backtracking (oncebu). These schemes differ in how they handle redexes and control flow, with fulltd allowing for repeated actions and stoptd halting after a single action.

The strategic programming approach encourages designing actions that can be combined in various ways, promoting a more flexible and expressive programming model.

For further details, please refer to [VBT98, BKVV08].
That is, the argument \(s\) (which is essentially a polymorphic function) is applied to all immediate subterms by combining the applications in the applicative functor style. (We could also use monadic bind instead.) There is one case for every constructor.

In reality, \(all\) and \(one\) are generically defined or definable for all (most) Haskell types based on the type-class instances for Haskell's 'Scrap Your Boilerplate' approach to generic functional programming [LJ03, LJ04, LJ05]. That is, the shown code would be essentially derived by the Haskell compiler. The following library defines the earlier traversal schemes and it also provides more basic combinators.

Illustration 12.8

(A small strategic programming library)

Haskell resource lib/Haskell/Data/Generics/Strategies.hs

```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)
onctd s = s `choice` one (onctd s)
oncebu s = one (oncebu s) `choice` s
innermost s = repeat (oncebu s)

-- Basic strategy combinators
s1 `sequ` s2 = \x \rightarrow s1 x >>= s2 -- monadic function composition
s1 `choice` s2 = \x \rightarrow 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
```

Thus, the traversal schemes are essentially defined in terms of sequential composition (\(sequ\)), left-biased choice, \(all\), \(one\), and recursive function definition. The strategy builders are needed to turn type-specific rewrite rules into generic functions so that we can traverse into terms of different types. (For instance, we may want to optimize expressions within BIPL statements or BFPL functions.) Several traversal schemes and the strategy builders are illustrated below.

Interactive Haskell session:

A strategy library in Haskell
The expression "$a + b \times 0$" with simplification potential

- let $e_1 = \text{Binary Add} (\text{Var} "a") (\text{Binary Mul} (\text{Var} "b") (\text{IntConst} 0))$

The expression "$(a \times b) \times c \times d$" associated to the left

- let $e_2 = \text{Binary Mul} (\text{Binary Mul} (\text{Binary Mul} (\text{Var} "a") (\text{Var} "b")) (\text{Var} "c")) (\text{Var} "d")$

The expression "$0 + a$" requiring commutativity for simplification

- let $e_3 = \text{Binary Add} (\text{IntConst} 0) (\text{Var} "a")$

---

Incomplete simplification with fulltd

- \text{fulltd (orSucceed simplify) } e_1
  Binary Add (Var "a") (IntConst 0)

Complete simplification with fullbu

- \text{fullbu (orSucceed simplify) } e_1
  Var "a"

Incomplete association to the right with fullbu

- \text{fullbu (orSucceed associate) } e_2
  Binary Mul (Var "a") (Binary Mul (Binary Mul (Var "b") (Var "c")) (Var "d"))

Complete association to the right with innermost

- \text{innermost (orFail associate) } e_2
  Binary Mul (Var "a") (Binary Mul (Var "b") (Binary Mul (Var "c") (Var "d")))

Apply simplification module commutativity

- \text{vary (orFail simplify) (orFail commute) } e_3
  Var "a"
Quasi-quotation

- SLE/MP relevance
  - Language embedding
- Aspects
  - Parser embedding
  - Metavariables or splices
An obvious drawback of this poorman's approach is that the proper use of the object language's syntax is not checked at compile time. Syntax errors as well as issues with conformance of the query to the underlying database schema would only be found at runtime. Perhaps a less obvious consequence of such poor checking is that programs become even vulnerable to injection attacks [BDV07, BDV10]. In this section, we focus on proper syntax-aware embedding of the object language into the metalanguage. In an extended Java language such that SQL is embedded, the above example may look as follows; this code is again adopted from [BDV07, BDV10]:

```sql
define q = <| SELECT id FROM users WHERE
    name = ${userName} AND password = ${password} |>
if (executeQuery(q.toString()).size() == 0) ...
```

The key idea is that, within the metalanguage (here: Java), we can embed object program fragments (here: SQL) by means of an appropriate escaping or quoting mechanism (see the '<|'···'|>' brackets) and we can escape back to the metalanguage to fill in details computed in the metaprogram (see the access to Java variables such as `serName$u`). Thus, syntax of object and metalanguage syntax are amalgamated in a certain manner.

### 7.5.1 Basic object syntax embedding

We will discuss here an approach to concrete object syntax which combines so-called quasi-quotation and language or syntax embedding [Mai07, Tra08, WH13]. We begin with a trivial example. Consider the following Haskell code which declares a binding for a finite state machine. We use so-called quasi-quote brackets `
[fsml|···|]`

(Or Oxford brackets) to quote an FSM with the Haskell code.

```
Illustration 7.26 (Embedding of FSML into Haskell).
```

```
Haskell module Language.FSML.QQ.Sample

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;
    }

|]
```

Quasi-quotation is realized (in Haskell) such that the quoted text is actually parsed at compile time. What happens underneath is that the parser synthesizes an abstract syntax tree using the regular algebraic datatype-based, abstract syntactical representation of the parsed language and the resulting expression is then mapped to Haskell and inserted into the code of the module. Thus, the shown binding has exactly the same meaning as if we had written Haskell code using abstract syntax instead. This can be compared with storing the FSM in a file and parsing it at runtime—except that quasi-quotation allows us to embed the FSM directly into the Haskell code and parsing happens transparently at compile time.

The quasi-quote brackets specify the language to be used; this is `fsml` in the example. The name is, in fact, the name of a binding of type `QuasiQuoter`, subject to Haskell's extension for quasi-quotation [Mai07] based also on Template Haskell [SP02].

A quasi-quoter essentially describes how to map strings to Haskell expressions or elements of other categories of Haskell's syntax.

```
Illustration 7.27 (A quasi-quoter for FSML).
```

```
Haskell module Language.FSML.QuasiQuoter

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

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

Thus, the quasi-quoter is a record with four components, each one applying to a different syntactic category. We are only concerned with expressions here (see `quoteExp = ···`) and we leave the components for patterns, types, and declarations undefined. Let us describe the actual binding for `quoteFsmlExp` in terms of three phases:

**Parsing**
The expression `parseQ fsm str` parses the string `str` between the quasi-quote brackets; it uses a standard parser `fsm` (as of Section 7.3.4). The function `parseQ` is a convenience wrapper around the normal run function...
A simple quasi-quoter for finite state machines in Haskell programs

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

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

Quasi-quotation is realized (in Haskell) such that the quoted text is actually parsed at compile time. What happens underneath is that the parser synthesizes an abstract syntax tree using the regular algebraic datatype-based, abstract syntactical representation of the parsed language and the resulting expression is then mapped to Haskell and inserted into the code of the module. Thus, the shown binding has exactly the same meaning as if we had written Haskell code using abstract syntax instead. This can be compared with storing the FSM in a file and parsing it at runtime—except that quasi-quotation allows us to embed the FSM directly into the Haskell code and parsing happens transparently at compile time.

The quasi-quote brackets specify the language to be used; this is `fsml` in the example. The name is, in fact, the name of a binding of type `QuasiQuoter`, subject to Haskell's extension for quasi-quotation [Mai07] based also on Template Haskell [SP02]. A quasi-quoter essentially describes how to map strings to Haskell expressions or elements of other categories of Haskell's syntax.
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

[Diagram of an attributed syntax tree for the binary number '101.01'.]

Fig. 3: An attributed syntax tree for the binary number '101.01'.

lates attributes of nonterminal symbols within the scope of a specific context-free rule. The order of computation (attribution) is not explicitly described, but it can be inferred from the attribute dependencies expressed by the computational rules. In Chapter 7, we discussed already a limited form of attribute grammars: grammars enhanced by semantic actions for AST construction to serve as input for a parser generator.

AGs are supported explicitly by some metaprogramming systems with designated AG languages, e.g., Eli [KPJ98], JastAdd [Hed11], Silver [VBGK10], or (Aspect)Lisa [RMHP06] and these systems support several AG extensions [KW94, LR99, Läm99]. The AG-style of metaprogramming and computation can also be leveraged, if a sufficiently powerful (declarative) meta-language is used. In particular, AGs can be 'encoded' in functional programming [SAS99, SKV13], as we will show below. Furthermore, a limited form of AGs is also supported by mainstream parser generators such as ANTLR, which we discussed earlier. In this section, we introduce the notion of AGs as a means of approaching problems of analysis, annotation, and translation.
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
22.12 Metaprogramming techniques

\[
\text{bits0.Val} = 2 \times \text{bits1.Val} + \text{bit.Val} \\
\text{bits0.Len} = \text{bits1.Len} + 1
\]

12.2.3 Functional programming-based encoding

Attribute grammars provide a declarative, computational paradigm that is actually very similar to (some form of) functional programming. That is, we may encode AGs as disciplined functional programs [SAS99, SKV13]. The encoding scheme is summarized as follows:

- Without loss of generality, we operate on the abstract as opposed to the concrete syntax. That is, we interpret computational rules on top of algebraic constructors as opposed to context-free rules.
- We associate each syntactic category with a function which models the computational rules for the cases of the category. The inherited attributes of the category become function arguments; the synthesized attributes become function results. Overall, we switch from the use of attribute names to the use of positions in argument and result tuples.
- Types of attribute values and operations on these types—as they are referred to in computational rules—are also to be modeled in the functional program. The operations correspond to additional functions.

This encoding is illustrated for the AG for binary-to-decimal number conversion.

**Illustration 12.14** (Haskell-based abstract syntax of BNL)

```haskell
data Number = Number Bits Rest

data Bits = Single Bit | Many Bit Bits

data Bit = Zero | One

data Rest = Integer | Rational Bits
```

**Illustration 12.15** (Binary to decimal number conversion in Haskell)

```haskell
number :: Number → Float
number (Number bs r) = val0

  where
  (len1, val1) = bits bs pos1
  pos1 = len1 – 1
  val2 = rest r
  val0 = val1 + val2
```

```haskell
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
```

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

rest :: Rest → Float

. . .
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 \( \lambda x \rightarrow x * x * x * 1 \).
Quasi-quotes and splicing with Template Haskell

Illustration 12.22

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

Illustration 12.23

```
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

zero \rightarrow zero

e \rightarrow n
\quad \Rightarrow\quad \text{succ}(e) \rightarrow \text{succ}(n)

\quad \begin{aligned}
  & e \rightarrow zero \\
  & \text{pred}(e) \rightarrow zero
\end{aligned}

\quad \begin{aligned}
  & e \rightarrow \text{succ}(n) \\
  & \text{pred}(e) \rightarrow n
\end{aligned}

evaluate \ Zero = \ Zero
evaluate (\text{Succ} \ e)
\quad | \quad \text{n} \leftarrow \text{evaluate} \ e
\quad | \quad , \ \text{isNat} \ n
\quad | \quad = \text{Succ} \ n
evaluate (\text{Pred} \ e)
\quad | \quad \text{Zero} \leftarrow \text{evaluate} \ e
\quad | \quad = \text{Zero}
evaluate (\text{Pred} \ e)
\quad | \quad \text{Succ} \ n \leftarrow \text{evaluate} \ e
\quad | \quad , \ \text{isNat} \ n
\quad | \quad = \text{n}
Small-step operational semantics
for a simple expression language

\[
\begin{align*}
    e & \rightarrow e' \\
    \text{succ}(e) & \rightarrow \text{succ}(e') \\
    e & \rightarrow e' \\
    \text{pred}(e) & \rightarrow \text{pred}(e') \\
    \text{pred}(\text{zero}) & \rightarrow \text{zero} \\
    \text{pred}(\text{succ}(n)) & \rightarrow n
\end{align*}
\]
Compositional (denotational) semantics
for a simple imperative programming language

\[
\begin{align*}
\mathcal{S}[\text{skip}] & = \text{skip} \\
\mathcal{S}[\text{assign}(x,e)] & = \text{assign } x (\mathcal{E}[e]) \\
\mathcal{S}[\text{seq}(s_1,s_2)] & = \text{seq } (\mathcal{S}[s_1]) (\mathcal{S}[s_2]) \\
\mathcal{S}[\text{if}(e,s_1,s_2)] & = \text{if } (\mathcal{E}[e]) (\mathcal{S}[s_1]) (\mathcal{S}[s_2]) \\
\mathcal{S}[\text{while}(e,s)] & = \text{while } (\mathcal{E}[e]) (\mathcal{S}[s])
\end{align*}
\]

\begin{verbatim}
-- 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 \rightarrow Store

execute :: Stmt \rightarrow 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)
\end{verbatim}

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

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

// Pointwise store update
assign x f m = m[x → (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 = \{ g m, if f m = true \\
                  h m, if f m = false \\
                  "undefined", otherwise \}

while f g = fix h
  where
    h t = if f (seq g t) 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

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
...

<table>
<thead>
<tr>
<th>+</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Neg</td>
<td>Neg</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Zero</td>
<td>Neg</td>
<td>Zero</td>
<td>Pos</td>
<td>?</td>
</tr>
<tr>
<td>Pos</td>
<td>?</td>
<td>Pos</td>
<td>Pos</td>
<td>?</td>
</tr>
</tbody>
</table>
Signature of semantic 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
}
```
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))
Wrap-up

• 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