Term rewriting — a primer

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
http://www.softlang.org/
Some laws for expressions forms

\[
\begin{align*}
X + 0 &= X & \text{-- Unit of addition} \\
X \times 1 &= X & \text{-- Unit of multiplication} \\
X \times 0 &= 0 & \text{-- Zero of multiplication} \\
X + Y &= Y + X & \text{-- Commutativity of addition} \\
X \times Y &= Y \times X & \text{-- Commutativity of multiplication} \\
(X + Y) + Z &= X + (Y + Z) & \text{-- Associativity of addition} \\
(X \times Y) \times Z &= X \times (Y \times Z) & \text{-- Associativity of multiplication} \\
(X \times Y) + (X \times Z) &= X \times (Y + Z) & \text{-- Distributivity}
\end{align*}
\]

Now assume that we wanted to implements these laws for the purpose of \textit{optimization}. 
Application of equations

Equation

\[ X \times 1 = X \]

Term rewriting

\[ a + b \times 1 + c = a + b + c \]

N.B.:
- An underlined term is called ‘redex’.
- The equation is applied here from left to right.
From equations to rules

\[
\begin{align*}
X + 0 & \rightsquigarrow X \quad \text{-- Unit of addition} \\
X \times 1 & \rightsquigarrow X \quad \text{-- Unit of multiplication} \\
X \times 0 & \rightsquigarrow 0 \quad \text{-- Zero of multiplication}
\end{align*}
\]

N.B.: In these directions, the rules can be understood to serve \textit{simplification}. 
From concrete to abstract syntax

\[ X + 0 \sim X \quad \text{-- Unit of addition} \]
\[ X \times 1 \sim X \quad \text{-- Unit of multiplication} \]
\[ X \times 0 \sim 0 \quad \text{-- Zero of multiplication} \]

\[
\begin{align*}
\text{binary}(\text{add},X,\text{intconst}(0)) & \sim X \quad \text{-- Unit of addition} \\
\text{binary}(\text{mul},X,\text{intconst}(1)) & \sim X \quad \text{-- Unit of multiplication} \\
\text{binary}(\text{mul},X,\text{intconst}(0)) & \sim \text{intconst}(0) \quad \text{-- Zero of multiplication}
\end{align*}
\]

N.B.: In this manner, metaprogramming is more straightforward.
Abstract syntax—signature vs. Haskell

/// Expressions
symbol intconst : integer → expr ;
symbol boolconst : boolean → expr ;
symbol var : string → expr ;
symbol unary : uop × expr → expr ;
symbol binary : bop × expr × expr → expr ;

/// Unary operators
symbol negate : → uop ;
symbol not : → uop ;

/// Binary operators
symbol add : → bop ;
symbol sub : → bop ;
symbol mul : → bop ;
symbol lt : → bop ;
symbol le : → bop ;
symbol eq : → bop ;
symbol geq : → bop ;

--- Expressions
data Expr =
  IntConst Int | BoolConst Bool | Var String | Unary UOp Expr | Binary BOp Expr Expr

--- Unary operators
data UOp = Negate | Not

--- Binary operators
data BOp = Add | Sub | Mul | Lt | Le | Eq | Geq | Gt | And | Or
Implementation of rewrite rules

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

N.B.:
• Functional programming-based implementation is straightforward.
• Function = group of rewrite rules with Maybe for success/failure.
Implementation of rewrite rules

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

N.B.:
• Functional programming-based implementation is straightforward.
• Function = group of rewrite rules with Maybe for success/failure.
Top-level application of rules

- 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

N.B.:
- The first application succeeds fine.
- The other applications fail for different reasons. (Which?)
Rewrite rules with ‘extras’

—– Cancel double negation on Ints

doubleNegate (Unary Negate (Unary Negate e)) = Just e
doubleNegate (Unary Negate (IntConst i)) | i <= 0 = Just (IntConst (−i))
doubleNegate _ = Nothing

—– Swap variable names

swap x y (Var z) | z == x = Just (Var y)
swap x y (Var z) | z == y = Just (Var x)
swap _ _ _ = Nothing

—– Compose simplification with optional commute

simplify’ x = simplify x `mplus` commute x >>= simplify

N.B.:
• Function equations may involve pattern guards.
• Functions may carry auxiliary argument.
• Functions can be composed.
Application of a rule with ‘extras’

- \[\text{simplify}' (\text{Binary Add} \ (\text{IntConst} \ 0) \ (\text{Var} \ "a"))\]
- \[\text{Just} \ (\text{Var} \ "a")\]

- \[\text{simplify} (\text{Binary Add} \ (\text{IntConst} \ 0) \ (\text{Var} \ "a"))\]
- \[\text{Nothing}\]

N.B.: simplify’ incorporates commutativity.

\[\text{Compose simplification with optional commute}\]
\[\text{simplify}' \ x = \text{simplify} \ x \ `\text{mplus}` \ \text{commute} \ x \ >>\text{=}\text{simplify}\]
In need of normalization

```
normalize simplify (Binary Add (Var "a") (Binary Add (Var "b") (IntConst 0)))
Binary Add (Var "a") (Var "b")
```

```
simplify (Binary Add (Var "a") (Binary Add (Var "b") (IntConst 0)))
Nothing
```

N.B.:
- We need a function like ‘normalize’ to get to redexes.
- That is, by default rules would only be applied at the top.
Boilerplate code for normalization

\[
\text{normalize} :: (\text{Expr} \rightarrow \text{Maybe Expr}) \rightarrow \text{Expr} \rightarrow \text{Expr}
\]

\[
\text{normalize } f \ e = \begin{cases} 
\text{let } e' = \text{pass } e \text{ in if } e == e' \text{ then } e \text{ else } \text{normalize } f \ e' 
\end{cases}
\]

where

\(-- \text{ Apply one pass of normalization} \)--

\[
\text{pass } e = \begin{cases} 
\text{sub (maybe } e \text{ id (f } e)\text{)}
\end{cases}
\]

\(-- \text{Push normalization into subexpressions} \)--

\[
\text{sub (Unary } o \ e) = \text{Unary } o \ (\text{pass } e)
\]
\[
\text{sub (Binary } o \ e1 \ e2) = \text{Binary } o \ (\text{pass } e1) \ (\text{pass } e2)
\]
\[
\text{sub } e = e
\]

N.B.:
- This code is specific to expressions.
- This code cannot be used for expressions with statements.
In need of traversal schemes

N.B.: Illustration of different traversal schemes parametrized by a 'strategy' (e.g., rewrite rules). It is conveyed what nodes are encountered along the traversal and whether the given strategy fails (see the gray nodes) or succeeds (see the black nodes).
In need of one-layer traversal

all versus one

N.B.: Recursive traversal = one-layer traversal + recursion
### A strategy library

--- Strategic traversal schemes

- `fulltd s = s `sequ` all (fulltd s)`
- `fullbu s = all (fullbu s) `sequ` s`
- `stoptd s = s `choice` all (stoptd s)`
- `oncedt s = s `choice` one (oncedt s)`
- `oncebu s = one (oncebu s) `choice` s`
- `innermost s = repeat (oncebu s)`

--- Basic strategy combinators

- `s1 `sequ` s2 = \( \lambda x \rightarrow s1 x \triangleright\triangleright= s2 \)` -- monadic function composition
- `s1 `choice` s2 = \( \lambda x \rightarrow s1 x \triangleright\triangleright= s2 x \)` -- monadic choice
- `all s = ...` -- magically apply \( s \) to all immediate subterms
- `one s = ...` -- magically find first immediate subterm for which \( s \) succeeds

--- 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

--- Heavy lifting!
One-layer traversal

all versus one

all s (IntConst i) = IntConst <$> s i
all s (BoolConst b) = BoolConst <$> s b
all s (Var v) = Var <$> s v
all s (Unary o e1) = Unary <$> s o <*> s e1
all s (Binary o e1 e2) = Binary <$> s o <*> s e1 <*> s e2
Application of strategies

--- 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

Exercise 12.4 (Applicability of innermost).

Consider again the swap function of Illustration 12.2. Why would a traversal based on innermost not produce the correct result with all occurrences of the two variables consistently swapped?
The expression 

\((a \cdot b) \cdot c) = a \cdot \text{id} \cdot (b \cdot c)\)

\--- The expression "0 + a" requiring commutativity for simplification

\>>> let e2 = Binary Mul (Binary Mul (Var "a") (Var "b")) (Var "c") (Var "d")

\--- 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 (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 simplification module commutativity

\>>> vary (orFail simplify) (orFail commute) e3

Var "a"
In need of concrete object syntax

-- 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

in Haskell
Syntax extension

```
data Expr
    = ...  -- The same syntax as before
    | MetaVar String  -- An additional constructor for the abstract syntax

factor :: Parser Expr
factor
    = ...  -- The same syntax as before
    <|> (MetaVar <$> (op "$" >>= identifier))  -- An additional choice in parsing
```
A quasi-quoter in Haskell

```haskell
el :: QuasiQuoter
el = QuasiQuoter
  { quoteExp = quoteElExp
    , quotePat = quoteElPat
    , quoteType = undefined
    , quoteDec = undefined
  }

quoteElExp :: String → Q Exp
quoteElExp str = do
  x ← parseQ expr str
  dataToExpQ (const Nothing `extQ` f) x
  where
    f :: Expr → Maybe (Q Exp)
    f (MetaVar v) = Just $ varE (mkName v)
    f _ = Nothing

quoteElPat :: String → Q Pat
quoteElPat str = do
  x ← parseQ expr str
  dataToPatQ (const Nothing `extQ` f) x
  where
    f :: Expr → Maybe (Q Pat)
    f (MetaVar v) = Just $ varP (mkName v)
    f _ = Nothing
```

```
simplify :: Expr → Maybe Expr
simplify [ell $x + 0 |] = Just [ell $x 1]
simplify [ell $x * 1 |] = Just [ell $x 1]
simplify [ell $x * 0 |] = Just [ell 0 1]
simplify _ = Nothing
```

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

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

The Software Languages Book
http://www.softlang.org/book