Term rewriting

Course "Software Language Engineering"

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team
Yaccification

number = bit+ ("." bit+)?
bit = "0" | "1"

Useful in grammar implementation

rewriting

number ::= bits rest
bits ::= bit
bits ::= bit bits
bit ::= "0"
bit ::= "1"
rest ::= ε
rest ::= "." bits
De-yaccification

number := bits rest
bits ::= bit
bits ::= bit bits
bit ::= "0"
bit ::= "1"
rest ::= ε
rest ::= "." bits

Useful in grammar re-documentation

number = bit+ ("." bit+)?
bit = "0" \| "1"
Refactoring ("extract method")

void printOwning(double amount) {
    printBanner();
    //print details
    System.out.println("name:" + _name);
    System.out.println("amount" + amount);
}

void printDetails(double amount) {
    System.out.println("name:" + _name);
    System.out.println("amount" + amount);
}

void printOwning(double amount) {
    printBanner();
    printDetails(amount);
}

Useful in an IDE
Schema normalization

Useful in Object/XML mapping

rewriting

```xml
<xs:complexType>
  <xs:sequence>
    <xs:element name="x" type="xs:int" minOccurs="1"/>
    <xs:element name="y" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
```
Primitive to tail recursion

```
public static int factorial(int n) {
    return (n == 0) ? 1 : n * factorial(n-1);
}
```

rewriting

```
public static int factorial(int n) {
    return factorial(1,n);
}
public static int factorial(int x, int n) {
    return (n==0) ? x : factorial(n*x,n-1);
}
```

Useful in optimizing compilation
Tail recursion to iteration

public static int factorial(int x, int n) {
    return (n==0) ? x : factorial(n*x,n-1);
}

rewriting

public static int factorial(int x, int n) {
    while (n!=0) {
        x *= n;
        n--;
    }
    return x;
}

Useful in optimizing compilation
Loop-invariant code motion

```
for (int i = 0; i < n; i++) {
    x = y + z;
    a[i] = 6 * i + x * x;
}
```

Rewriting:
```
x = y + z;
tl = x * x;
for (int i = 0; i < n; i++) {
    a[i] = 6 * i + tl;
}
```

Useful in optimizing compilation

Overall issues

- Describe patterns (terms) of interest.
- Describe side conditions.
- Describe search strategy.
- Reason about correctness including termination.
- Implement all ingredients modularly and declaratively.
From equations to rewriting strategies via rewrite rules
Motivation

Consider this expression:

\[ 0 \times a + b \]

with \( a, b \) as placeholders for expressions.

How to describe a simplification producing this result?

\[ b \]

Much of language engineering is about such simplifications. Can you name additional examples?
Relevant equations (laws)

\[
\begin{align*}
\text{x+0} &= \text{x} \\
\text{x*1} &= \text{x} \\
\text{x*0} &= 0 \\
\text{x+y} &= \text{y+x} \\
\text{x*y} &= \text{y*x} \\
(\text{x+y})+\text{z} &= \text{x+(y+z)} \\
(\text{x*y})z &= \text{x*(y*z)} \\
\text{x*y+x*z} &= \text{x*(y+z)}
\end{align*}
\]

How to perform this simplification?

\[0*a + b \rightarrow b\]
Equational reasoning

\[ 0 \times a + b = a \times 0 + b \quad \text{(commutativity of \( \times \))} \]
\[ = 0 + b \quad \text{(zero of \( \times \))} \]
\[ = b + 0 \quad \text{(commutativity of \( + \))} \]
\[ = b \quad \text{(unit of \( + \))} \]
Nontermination

\[ a = a + 0 = a + 0 + 0 = \ldots \]

Apply unit of "+" forever.
Rewrite rules (as opposed to equations)

- \( x + 0 \rightarrow x \)
- \( x \cdot 1 \rightarrow x \)
- \( x \cdot 0 \rightarrow 0 \)
- \( x + y \rightarrow y + x \)
- \( x \cdot y \rightarrow y \cdot x \)
- \( (x + y) + z \rightarrow x + (y + z) \)
- \( (x \cdot y) \cdot z \rightarrow x \cdot (y \cdot z) \)
- \( x \cdot y + x \cdot z \rightarrow x \cdot (y + z) \)

Is there any solid argument for the direction of the rules?
Nontermination

\[
\begin{align*}
  a+b &= b+a \\
  &= a+b \\
  &= \ldots
\end{align*}
\]

Apply commutativity of "+" forever.
Different kinds of rewrite rules

- Simplification / reduction:
  - $x + 0 \rightarrow x$
  - $x^1 \rightarrow x$
  - $x^0 \rightarrow 0$
  - $x + y \rightarrow y + z$
  - $x \cdot y \rightarrow y \cdot x$

- Self-enabling:
  - $(x + y) + z \rightarrow x + (y + z)$
  - $(x^\ast y)^\ast z \rightarrow x^\ast (y^\ast z)$

- Strongly normalizing:
  - $x \cdot y + x^\ast z \rightarrow x^\ast (y + z)$
Interplay

Commutativity may help generalizing simplification.

\[
\begin{align*}
\text{x+0} & \rightarrow \text{x} \\
\text{x*1} & \rightarrow \text{x} \\
\text{x*0} & \rightarrow 0
\end{align*}
\]

\[
\begin{align*}
\text{x+y} & \rightarrow y+z \\
\text{x*y} & \rightarrow y*x
\end{align*}
\]
Element of choice

Both directions of associativity normalization make sense.

\[(x+y)+z \rightarrow x+(y+z)\]
\[(x*y)*z \rightarrow x^*(y*z)\]

\[(x+y)+z \leftarrow x+(y+z)\]
\[(x*y)*z \leftarrow x^*(y*z)\]
Normalization

How to search for redexes?

- Bottom-up?
- Top-down?

How many times to go over term?

- One time?
- Many times?
Top down, one time

Objective: simplification

Input: \(a \times 1 + b \times 0\)

No simplification rule is applicable at the top.

Simplification rules are applicable to operands of "+":

- \(a \times 1 \rightarrow a\)
- \(b \times 0 \rightarrow 0\)

Output: \(a + 0\)

Another pass would complete simplification.
Bottom up, one time

Objective: simplification

Input: \(a \times 1 + b \times 0\)

Simplification rules are applicable to operands of "+".

\[\begin{align*}
 a \times 1 & \rightarrow a \\
 b \times 0 & \rightarrow 0
\end{align*}\]

Simplification rule is applicable at the top:

\[\begin{align*}
 a + 0 & \rightarrow a
\end{align*}\]

Output: \(a\)

Is 1 pass always sufficient?
Bottom up, one time

Objective: normalization for left associativity

\[ a \cdot (b \cdot (c \cdot d)) = a \cdot ((b \cdot c) \cdot d) = (a \cdot (b \cdot c)) \cdot d \]

More than 1 pass needed!
Strategic programming
The strategic programming method

- Define problem-specific rules.
  - Possibly define several rule sets.
- Identify appropriate traversal scheme.
- Apply scheme to rules.
  - Possibly compose nested traversal.
Traversals schemes

Examples:

- `full_td` -- full top-down
  - Apply argument to all nodes.

- `once_bu` -- once bottom-up
  - Find one node to which the argument can be applied.

Define schemes in terms of one-layer traversal.
Deep traversal

Below we illustrate two recursive completions of the one-layer combinators. The upper one completes all into a full top-down traversal (for short, full_td). The lower one completes one into a single-hit bottom-up traversal (for short, once_bu).

["The Essence of Strategic Programming" by Ralf Lämmel, Eelco Visser, Joost Visser]
One-layer traversal

Below we illustrate two one-layer traversal combinators: **all** to process all immediate components, and **one** to process the leftmost one for which the argument strategy succeeds. (Shaded vs. black nodes represent failure vs. success of processing.)

["The Essence of Strategic Programming" by Ralf Lämmel, Eelco Visser, Joost Visser]
"The Essence of Strategic Programming" by Ralf Lämmel, Eelco Visser, Joost Visser
Variation points in traversal

- Top-down versus bottom-up
- Find one redex versus find all or many redexes
- Inspect all nodes versus terminate traversal “sometimes”
- Left to right or vice versa (“not very interesting”)
- One pass versus repeat until failure or “no change”
- Pass down information during traversal (or not)
- ...

Rewriting in Prolog
Rewrite rules as predicates

simplify(X+0,X).
simplify(X*1,X).
simplify(_*0,0).
simplify(X*Y+X*Z,X*(Y+Z)).
Rewrite rules as predicates

commute(X+Y,Y+X).
commute(X*Y,Y*X).

associate(((X*Y)*Z,X*(Y*Z))).
associate(((X+Y)+Z,X+(Y+Z))).
Meta-predicates for strategies

simplify: $a^*0+b \rightarrow \text{FAIL}$

try(simplify): $a^*0+b \rightarrow a^*0+b$

repeat(simplify): $a^*0+b \rightarrow a^*0+b$

oncebu(simplify): $a^*0+b \rightarrow 0+b$

innermost(simplify): $a^*0+b \rightarrow 0+b$

innermost(vary(commute,simplify)): $a^*0+b \rightarrow b$
% Always succeed and preserve term
\[ \text{id}(X,X). \]
% Try S1; if it fails, try S2
choice(S1,S2,X,Y) :-
   apply(S1,[X,Y]) -> true
; apply(S2,[X,Y]).
% Try S but succeed anyway
try(S,X,Y) :-
  choice(S,id,X,Y).
% Sequentially compose S1 and S2
seq(S1,S2,X,Z) :-
    apply(S1,[X,Y]),
    apply(S2,[Y,Z]).
% Apply S exhaustively
repeat(S,X,Y) :-
  try(seq(S,repeat(S)),X,Y).
% Apply S once in bottom-up order
oncebu(S,X,Y) :-
choice(one(oncebu(S)),S,X,Y).
% Apply S to one immediate subterm successfully
one(S,X,Y) :-
  X =.. [F|LX],
  append(Prefix,[EX|Postfix],LX),
  apply(S,[EX,EY]),
  append(Prefix,[EY|Postfix],LY),
  Y =.. [F|LY].
% Apply S to one immediate subterm successfully
one(S,X,Y) :-
X = .. [F|LX],
append(Prefix,[EX|Postfix],LX),
apply(S,[EX,EY]),
!,
append(Prefix,[EY|Postfix],LY),
Y = .. [F|LY].

Pick leftmost term forever.
% Repeat oncebu exhaustively
innermost(S,X,Y) :-
repeat(oncebu(S),X,Y).
Concluding remarks

- Term rewriting is programmable equational reasoning.
- Term rewriting supports language implementation.
- Control is needed for term rewriting in practice.
- Aka strategic programming
- Strategic programming can be expressed in declarative languages.
Rewriting-able technologies

- Prolog (as shown)
- Haskell (see Data.Generics or Strafunski)
- Rascal (see for yourself)
- Tom (on top of Kava)
- Kiama (on top of Scala)
- JastAdd (on top of attribute grammars)

...