Automated Reasoning

Lecture 12: Rewriting I

Jacques Fleuriot
jdf@inf.ed.ac.uk
Recap

- Previously:
  - Unification
- This time: Rewriting
  - Sets of rewrite rules
  - Termination
  - Rewriting in Isabelle
Term Rewriting

Rewriting is a technique for replacing terms in an expression with equivalent terms.

For example, the rules:

\[ x \cdot 0 \Rightarrow 0 \quad x + 0 \Rightarrow x \]

can be used to simplify an expression:

\[ x + (x \cdot 0) \rightarrow x + 0 \rightarrow x \]

We use the notation \( L \Rightarrow R \) to define a rewrite rule that replaces the term \( L \) with the term \( R \) in an expression and \( s \rightarrow t \) to denote a rewrite rule application, where expression \( s \) gets rewritten to an expression \( t \).

In general, rewrite rules contain (meta-)variables (e.g., \( X + 0 \Rightarrow X \)), and are instantiated using matching (one-way unification).
The Power of Rewrites

Given this set of rules:

\[ 0 + N \Rightarrow N \quad (1) \]
\[ (0 \leq N) \Rightarrow \text{True} \quad (2) \]
\[ s(M) + N \Rightarrow s(M + N) \quad (3) \]
\[ s(M) \leq s(N) \Rightarrow M \leq N \quad (4) \]

We can prove this statement:

\[
0 + s(0) \leq s(0) + x \\
\Rightarrow \quad s(0) \leq s(0) + x \quad \text{by (1)} \\
\Rightarrow \quad s(0) \leq s(0 + x) \quad \text{by (3)} \\
\Rightarrow \quad 0 \leq 0 + x \quad \text{by (4)} \\
\Rightarrow \quad \text{True} \quad \text{by (2)}
\]
Symbolic Computation

Given this set of rules:

0 + \( N \)  \( \Rightarrow \)  \( N \)  \hspace{1cm} (1)

\( s(M) + N \)  \( \Rightarrow \)  \( s(M + N) \)  \hspace{1cm} (2)

0 * \( N \)  \( \Rightarrow \)  0  \hspace{1cm} (3)

\( s(M) * N \)  \( \Rightarrow \)  \( (M * N) + N \)  \hspace{1cm} (4)

(\( s(x) \) means “successor of \( x \)”, \( i.e. \) \( 1 + x \))

We can rewrite \( 2 * x \) to \( x + x \):

\[
\begin{align*}
s(s(0)) * x & \rightarrow (s(0) * x) + x \hspace{1cm} \text{by (4)} \\
& \rightarrow ((0 * x) + x) + x \hspace{1cm} \text{by (4)} \\
& \rightarrow (0 + x) + x \hspace{1cm} \text{by (3)} \\
& \rightarrow x + x \hspace{1cm} \text{by (1)}
\end{align*}
\]
Rewrite Rule of Inference

\[
\begin{array}{ccc}
P\{t\} & L \Rightarrow R & L[\theta] \equiv t \\ \hline \\ P\{R[\theta]\}
\end{array}
\]

where \(P\{t\}\) means that \(P\) contains \(t\) somewhere inside it.

Note: rewriting uses matching, not unification (the substitution \(\theta\) is not applied to \(t\)).

Example

Given an expression \((s(a) + s(0)) + s(b)\)

and a rewrite rule \(s(X) + Y \Rightarrow s(X + Y)\)

we can find \(t = s(a) + s(0)\)

and \(\theta = [a/X, s(0)/Y]\)

\[\text{to yield } s(a + s(0)) + s(b)\]
A rewrite rule $\alpha \Rightarrow \beta$ must satisfy the following restrictions:

- $\alpha$ is not a variable.
  For example, $x \Rightarrow x + 0$ is not allowed. If the LHS can match anything, then it’s very hard to control.

- $\text{vars}(\beta) \subseteq \text{vars}(\alpha)$.
  This rules out $0 \Rightarrow 0 \times x$ for example. This ensures that if we start with a ground term, we will always have a ground term.
More on Notation

- Rewrite rules: $L \Rightarrow R$, as we’ve seen already.

- Rewrite rule applications: $s \rightarrow t$
  
  e.g., $s(s(0)) \ast x \rightarrow (s(0) \ast x) + x$

- Multiple (zero or more) rewrite rule applications: $s \rightarrow^* t$
  
  e.g., $s(s(0)) \ast x \rightarrow^* x + x$
  
  e.g., $0 \rightarrow^* 0$

- Back-and-forth:
  
  - $s \leftrightarrow t$ for $s \rightarrow t$ or $t \rightarrow s$
  
  - $s \leftrightarrow^* t$ for a chain of zero or more $u_i$ such that
    
    $s \leftrightarrow u_1 \leftrightarrow ... \leftrightarrow u_n \leftrightarrow t$
A rewrite rule $L \Rightarrow R$ on its own is just a “replace” instruction. To be useful, it must have some logical meaning attached to it.

Most commonly, a rewrite $L \Rightarrow R$ means that $L = R$;

- Rewrites can instead be based on implications and other formulas (e.g., $a = b \mod n$), but care is needed to make sure that rewriting corresponds to logically valid steps.

  e.g., if $A \rightarrow B$ means $A$ implies $B$, then it is safe to rewrite $A$ to $B$ in $A \land C$, but not in $\neg A \land C$. Why?
How to choose rewrite rules?

There are often many equalities to choose from:

\[
X + Y = Y + X \quad X + (Y + Z) = (X + Y) + Z \quad X + 0 = X \\
0 + X = X \quad 0 + (X + Y) = Y + X \quad \ldots
\]

Could all be valid rewrite rules.

But: *Not everything that can be rewrite rule should be a rewrite rule!*

- Ideally, a set of rewrite rules should be **terminating**
- Ideally, they should rewrite to a **canonical normal form**
An Example: Algebraic Simplification

Rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Example</th>
<th>Rule</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \cdot 0 \Rightarrow 0$</td>
<td>$a^{2 \cdot 0} \cdot 5 + b \cdot 0$</td>
<td>$1 \cdot x \Rightarrow x$</td>
<td>$\longrightarrow a^0 \cdot 5 + b \cdot 0$ by (1)</td>
</tr>
<tr>
<td>$x^0 \Rightarrow 1$</td>
<td>$\longrightarrow 1 \cdot 5 + b \cdot 0$ by (3)</td>
<td>$x + 0 \Rightarrow x$</td>
<td>$\longrightarrow 5 + b \cdot 0$ by (2)</td>
</tr>
</tbody>
</table>

Any subexpression that can be rewritten (i.e. matches the LHS of a rewrite rule) is called a redex (reducible expression).

The redxes used (but not all redxes) have been underlined above.

Choices: Which redex to choose? Which rule to choose?
The Rewrite Search Tree

In general, get a tree of possible rewrites:

\[ a^{2*0} \* 5 + b \* 0 \]

Common strategies:
- Innermost (inside-out) leftmost redex
- Outermost (outside-in) leftmost redex

Important questions:
- Is the tree finite? (does the rewriting always terminate?)
- Does it matter which path we take? (is every leaf the same?)
Termination

We say that a set of rewrite rules is *terminating* if:

*starting with any expression, successively applying rewrite rules eventually brings us to a state where no rule applies.*

Also called *(strongly) normalizing* or *noetherian*.

All the rewrite sets so far in this lecture are terminating.

Examples of rules that *may* cause non-termination:

- Reflexive rules: *e.g.* $0 \Rightarrow 0$
- Self-commuting rewrites: *e.g.* $X \ast Y \Rightarrow Y \ast X$, but not with a lexicographical measure.
- Commuting pairs of rewrites: *e.g.:
  
  $X + (Y + Z) \Rightarrow (X + Y) + Z$ and $(X + Y) + Z \Rightarrow X + (Y + Z)$

An expression to which no rewrite rules apply is called a *normal form* (with respect to that set of rewrite rules).
Proving Termination

Termination can be shown in some cases by:

1. defining a natural number measure on expressions
2. such that each rewrite rule decreases the measure

Measure cannot go below zero, so any sequence will terminate.

Example:

\[
\begin{align*}
x \cdot 0 & \Rightarrow 0 & (1) \\
1 \cdot x & \Rightarrow x & (2) \\
x^0 & \Rightarrow 1 & (3) \\
x + 0 & \Rightarrow x & (4)
\end{align*}
\]

For these rules, define the measure of an expression as the number of binary operations \((+, -, *)\) it contains.

Every rule removes a binary operation, so each rule application will reduce the overall measure of an expression.

In general: look for a well-founded termination order (e.g., lexicographical path ordering (LPO))
Consider the following rewrite rule:

\[ f(f(x)) \Rightarrow f(g(f(x))) \]

Is it terminating? If so, why?

How about:

\[ -(x + y) \Rightarrow (- - x + y) + y \]

where \( x \) and \( y \) are variables? Can you show that it is non-terminating?
Interlude: Rewriting in Isabelle

Isabelle has two rules for primitive rewriting (useful with `erule`):

\[
\begin{align*}
\text{subst} & : \quad [?s = ?t; ?P ?s] \Rightarrow ?P ?t \\
\text{s subj} & : \quad [?t = ?s; ?P ?s] \Rightarrow ?P ?t
\end{align*}
\]

The $?P$ is matched against the term using higher-order unification.

There is also a tactic that rewrites using a theorem:

\[
\begin{align*}
\text{apply (subst theorem)} & : \text{rewrites goal using theorem} \\
\text{apply (subst (asm) theorem)} & : \text{rewrites assumptions using theorem} \\
\text{apply (subst (i1 i2...) theorem)} & : \text{rewrites goal at positions } i_1, i_2, \ldots \\
\text{apply (subst (asm) (i1 i2...) theorem)} & : \text{rewrites assumptions at positions } i_1, i_2, \ldots
\end{align*}
\]

Working out what the right positions are is essentially just trial and error, and can be quite brittle.
The Isabelle Simplifier

The methods (tactics) simp and auto:

- simp does automatic rewriting on the first subgoal, using a database of rules also known as a simpset.
- auto simplifies all subgoals, not just the first one.
- auto also applies all obvious logical (Natural Deduction) steps:
  - splitting conjunctive goals and disjunctive assumptions
  - quantifier removals – which ones?

Adding [simp] after a lemma (or theorem) name when declaring it adds that lemma to the simplifier’s database/simpset.

- If it is not an equality, then it is treated as $P = \text{True}$.
- Many rules are already added to the default simpset – so the simplifier often appears quite magical.
The Isabelle Simplifier

Variations on simp and auto enable control over the rules used:
- simp add: ... del: : ...
- simp only: : ...
- simp (no_asm) – ignore assumptions
- simp (no_asm_simp) – use assumps, but do not rewrite them
- simp (no_asm_use) – rewrite assumps, don’t use them
- auto simp add: ... del: ...

A few specialised simpsets (for arithmetic reasoning):
- add_ac and mult_ac: associative-commutative properties of addition and multiplication
- algebra_simps: useful for multiplying out polynomials
- field_simps: useful for multiplying out denominators when proving inequalities e.g. auto simp add: field_simps

Note Every definition defn in Isabelle generates an associated rewrite rule defn_def.
The Isabelle Simplifier

The Isabelle simplifier also has more bells and whistles:

1. Conditional rewriting: Apply \([P_1; \ldots ; P_n] \implies s = t\) if
   - the lhs \(s\) matches some expression and
   - Isabelle can \textit{recursively} prove \(P_1, \ldots, P_n\) by rewriting.

   \[
   \begin{align*}
   \text{Example:} & \quad \left\lbrack a \neq 0; b \neq 0 \right\rbrack \implies \frac{b}{(a \ast b)} = \frac{1}{a} \\
   \end{align*}
   \]

2. (Termination of) Ordered rewriting: a lexicographical (dictionary) ordering is used to \textit{prevent} (some) loops like:

   \[
   a + b \rightarrow b + a \rightarrow a + b \rightarrow \ldots
   \]

   Using \(x + y = y + x\) as a rewrite rule is actually okay in Isabelle.

3. Case splitting:

   \[
   \begin{align*}
   ?P \ (\text{case } ?x \text{ of True } \Rightarrow \ ?f_1 \mid \text{False } \Rightarrow \ ?f_2) \\
   = \ ((?x = \text{True } \Rightarrow ?P \ ?f_1) \land (?x = \text{False } \Rightarrow ?P \ ?f_2))
   \end{align*}
   \]

   Applies when there is an explicit case split in the goal
Summary

- Rewriting (Bundy Ch. 9)
  - Rewriting expressions using rules
  - Termination (by strictly decreasing measure)
- Rewriting in Isabelle (Isabelle Tutorial, Section 3.1)
- Next time: More on Rewriting