Automated Reasoning

Lecture 14: 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:

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

can be used to simplify an expression:

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

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:

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

We can prove this statement:

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

Given this set of rules:

\begin{align}
0 + N & \Rightarrow N \quad (1) \\
s(M) + N & \Rightarrow s(M + N) \quad (2) \\
0 \times N & \Rightarrow 0 \quad (3) \\
s(M) \times N & \Rightarrow (M \times N) + N \quad (4)
\end{align}

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

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

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

\[
P\{t\} \quad L \Rightarrow R \quad L[\theta] \equiv t \quad \frac{\quad P\{R[\theta]\} \quad}{
\]

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

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: \( s \longrightarrow t \)
  
  e.g., \( s(s(0)) \ast x \longrightarrow (s(0) \ast x) + x \)

- Multiple (zero or more) rewrite rule applications: \( s \longrightarrow^* t \)

  e.g., \( s(s(0)) \ast x \longrightarrow^* x + x \)

  e.g., \( 0 \longrightarrow^* 0 \)

- Back-and-forth:
  - \( s \leftrightarrow t \) for \( s \longrightarrow t \) or \( t \longrightarrow s \)
  - \( s \leftrightarrow^* t \) for a chain of zero or more \( u_i \) such that
    
    \( s \leftrightarrow u_1 \leftrightarrow \ldots \leftrightarrow u_n \leftrightarrow t \)
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:

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

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>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x * 0 \Rightarrow 0$</td>
<td>(1)</td>
</tr>
<tr>
<td>$1 * x \Rightarrow x$</td>
<td>(2)</td>
</tr>
<tr>
<td>$x^0 \Rightarrow 1$</td>
<td>(3)</td>
</tr>
<tr>
<td>$x + 0 \Rightarrow x$</td>
<td>(4)</td>
</tr>
</tbody>
</table>

**Example:**

$$a^{2*0} * 5 + b * 0$$

- $\rightarrow a^0 * 5 + b * 0$ by (1)
- $\rightarrow 1 * 5 + b * 0$ by (3)
- $\rightarrow 5 + b * 0$ by (2)
- $\rightarrow 5 + 0$ by (1)
- $\rightarrow 5$ by (4)

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

The redexes used (but *not* all redexes) 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:

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

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:

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

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/":

\[
\text{subst} \quad : \quad \left[ ?s = ?t; ?P ?s \right] \Rightarrow ?P ?t
\]

\[
\text{ssubst} \quad : \quad \left[ ?t = ?s; ?P ?s \right] \Rightarrow ?P ?t
\]

The \( ?P \) is matched against the term using higher-order unification.

There is also a tactic that rewrites using a theorem:

\[
\text{apply} \ (\text{subst} \ \text{theorem}) \quad : \quad \text{rewrites goal using theorem}
\]

\[
\text{apply} \ (\text{subst} \ (\text{asm}) \ \text{theorem}) \quad : \quad \text{rewrites assumptions using theorem}
\]

\[
\text{apply} \ (\text{subst} \ (i_1 \ i_2 \ldots) \ \text{theorem}) \quad : \quad \text{rewrites goal at positions } i_1, i_2, \ldots
\]

\[
\text{apply} \ (\text{subst} \ (\text{asm}) \ (i_1 \ i_2 \ldots) \ \text{theorem}) \quad : \quad \text{rewrites assumptions at positions } i_1, i_2, \ldots
\]

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 assms, but do not rewrite them
- simp (no_asm_use) – rewrite assms, 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 \([P_1, \ldots, P_n] \Rightarrow s = t\) if
   - the lhs \(s\) matches some expression and
   - Isabelle can recursively prove \(P_1, \ldots, P_n\) by rewriting.

   Example: \([a \neq 0; b \neq 0] \Rightarrow b/(a \times b) = 1/a\)

2. (Termination of) Ordered rewriting: a lexicographical (dictionary) ordering is used to 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:

   \(?P (\text{case } ?x \text{ of True } \Rightarrow ?f_1 | \text{False } \Rightarrow ?f_2)\)
   \(= (?(x = \text{True } \Rightarrow ?P ?f_1) \land (?(x = \text{False } \Rightarrow ?P ?f_2)))\)

   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