Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

http://www.infsec.ethz.ch/education/permanent/csmr/

Isabelle: Term Rewriting

Burkhart Wolff

Outline of this Part

- Higher-order rewriting
- Extensions: Ordered, pattern, congruence, splitting rewriting
- Organizing simplification rules

In this context, a term is a λ-term, since we use the λ-calculus to encode object logics.

Higher-Order Rewriting

Motivation:

- Simplification is a very important part of deduction, e.g.:

$$0 + (x + 0) = x$$
$$[a, b, d] \mathbin{@} [a, b] = [a, b, d, a, b]$$

- Based on rewrite rules as in functional programming:

$$x + 0 = x, \qquad 0 + x = x$$
$$[] \mathbin{@} X = X, \qquad (x :: X) \mathbin{@} Y = x :: (X \mathbin{@} Y)$$

Term Rewriting: Foundation

- Recall: An equational theory consists of rules

$$\frac{}{x = x}\ \textit{refl} \qquad \frac{x = y}{y = x}\ \textit{sym} \qquad \frac{x = y \quad y = z}{x = z}\ \textit{trans}$$

$$\frac{x = y \quad P(x)}{P(y)}\ \textit{subst}$$

- plus additional (possibly conditional) rules of the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Rightarrow \phi = \psi$.

The additional rules can be interpreted as rewrite rules, i.e. they are applied from left to right.

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:
  $simplify_R(e = e') =>$
  $repeat$
  (a) pick terms h and t such that $(e = e') \equiv h(t)$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:
  $simplify_R(e = e') =>$
  $repeat$
  (a) pick terms h and t such that $(e = e') \equiv h(t)$
  (b) pick a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ from R,
  match (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:
  $simplify_R(e = e') =>$
  $repeat$
  (a) pick terms h and t such that $(e = e') \equiv h(t)$
  (b) pick a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ from R, match (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$
  (c) replace $e = e'$ by $h(\psi\theta)$ provided all $simplify((\phi_i = \psi_i)\theta)$ are solved for all $i \in \{1..n\}$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:

  $simplify_R(e = e') =>$

  $repeat$

  (a) pick terms h and t such that $(e = e') \equiv h(t)$

  (b) pick a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ from R, match (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$

  (c) replace $e = e'$ by $h(\psi\theta)$ provided all $simplify((\phi_i = \psi_i)\theta)$ are solved for all $i \in \{1..n\}$

  $until$ no replacement possible, return current $e = e'$

Algorithm $simplify_R$

- We assume a rule set R
- An equation is solved if it has the form $e = e$
- An equation is simplified by:

  $simplify_R(e = e') =>$

  $repeat$

  (a) pick terms h and t such that $(e = e') \equiv h(t)$

  (b) pick a rewrite rule $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$ from R, match (unify) ϕ against t, i.e., find θ such that $\phi\theta = t$

  (c) replace $e = e'$ by $h(\psi\theta)$ provided all $simplify((\phi_i = \psi_i)\theta)$ are solved for all $i \in \{1..n\}$

  $until$ no replacement possible, return current $e = e'$

Problems with $simplify$

- This algorithm may fail because:
  - it diverges (the rules are not terminating), e.g. $x + y = y + x$ or $x = y \Longrightarrow x = y$;

Problems with $simplify$

- This algorithm may fail because:
  - it diverges (the rules are not terminating), e.g. $x + y = y + x$ or $x = y \Longrightarrow x = y$;
  - rewriting does not yield a unique normal form (the rules are not confluent), e.g. rules $a = b$, $a = c$.

Problems with $simplify$

- This algorithm may fail because:
  - it diverges (the rules are not terminating), e.g. $x + y = y + x$ or $x = y \Longrightarrow x = y$;
  - rewriting does not yield a unique normal form (the rules are not confluent), e.g. rules $a = b$, $a = c$.
- Providing criteria for terminating and confluent rule sets R is an active research area (see [BN98, Klo93], RTA, . . .).

Extensions of Rewriting

- Symmetric rules are problematic, e.g. ACI:

$$\begin{aligned} (x + y) + z &= x + (y + z) && \text{(A)} \\ x + y &= y + x && \text{(C)} \\ x + x &= x && \text{(I)} \end{aligned}$$

Extensions of Rewriting

- Symmetric rules are problematic, e.g. ACI:

$$\begin{aligned} (x+y)+z &= x+(y+z) && \text{(A)} \\ x+y &= y+x && \text{(C)} \\ x+x &= x && \text{(I)} \end{aligned}$$

- Idea: apply only if replaced term gets smaller w.r.t. some term ordering. In example, if $y+x\theta$ is smaller than $x+y\theta$.
- Ordered rewriting solves rewriting modulo ACI, using derived rules (exercise).

Extension: HO-Pattern Rewriting

Rules such as $F(G\,c) = \ldots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to higher-order patterns.

Extension: HO-Pattern Rewriting

Rules such as $F(G\,c) = \ldots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to higher-order patterns.

A term t is a HO-pattern if

- it is in β-normal form; and
- any free F in t occurs in a subterm $F\,x_1 \ldots x_n$ where the x_i are η-equivalent to distinct bound variables.

Extension: HO-Pattern Rewriting

Rules such as $F(G\,c) = \ldots$ lead to highly ambiguous matching and hence inefficiency.

Solution: restrict l.h.s. of a rule to higher-order patterns.

A term t is a HO-pattern if

- it is in β-normal form; and
- any free F in t occurs in a subterm $F\,x_1 \ldots x_n$ where the x_i are η-equivalent to distinct bound variables.

Matching (unification) is decidable, unitary ('unique') and efficient algorithms exist.

HO-Pattern Rewriting (Cont.)

A rule $\ldots \Rightarrow \phi = \psi$ is a HO-pattern rule if:

- the left-hand side ϕ is a HO-pattern;
- all free variables in ψ occur also in ϕ; and
- ϕ is constant-head, i.e. of the form $\lambda x_1..x_m.c\, p_1 \ldots p_n$ (where c is a constant, $m \geq 0$, $n \geq 0$).

HO-Pattern Rewriting (Cont.)

A rule $\ldots \Rightarrow \phi = \psi$ is a HO-pattern rule if:

- the left-hand side ϕ is a HO-pattern;
- all free variables in ψ occur also in ϕ; and
- ϕ is constant-head, i.e. of the form $\lambda x_1..x_m.c\, p_1 \ldots p_n$ (where c is a constant, $m \geq 0$, $n \geq 0$).

Example: $(\forall x.Px \wedge Qx) = (\forall x.Px) \wedge (\forall x.Qx)$

Result: HO-pattern allows for very effective quantifier reasoning.

Extension: Congruence Rewriting

Problem :

$$\texttt{if}\ A\ \texttt{then}\ P\ \texttt{else}\ Q\ =\ \texttt{if}\ A\ \texttt{then}\ P'\ \texttt{else}\ Q$$
$$\text{where } P = P' \text{ under condition } A$$

is not a rule.

Solution in Isabelle: explicitely admit this extra class of rules (congruence rules)

$$[\![A \Longrightarrow P = P']\!] \Longrightarrow$$
$$\texttt{if}\ A\ \texttt{then}\ P\ \texttt{else}\ Q\ =\ \texttt{if}\ A\ \texttt{then}\ P'\ \texttt{else}\ Q$$

Extension: Splitting Rewriting

Problem:

$$P(\texttt{if}\ A\ \texttt{then}\ x\ \texttt{else}\ y) = ((A \Longrightarrow P\,x) \wedge (\neg A \Longrightarrow P\,y))$$

is not a HO-pattern rule (since it is not constant-head).
Similar problems arise in connection with data types and their resulting case match statements (to be discussed later).
Solution in Isabelle: explicitely admit this extra class of (splitting rules).

Organizing Simplification Rules

- Standard (HO-pattern conditional ordered rewrite) rules;
- congruence rules;
- splitting rules.

In the Isabelle kernel, on the SML level, the data structure `simpset` is provided. Some operations:

- `addsimps` : `simpset` $*$ `thm list` $\longrightarrow$ `simpset`
- `delsimps` : `simpset` $*$ `thm list` $\longrightarrow$ `simpset`
- `addcongs` : `simpset` $*$ `thm list` $\longrightarrow$ `simpset`
- `addsplits` : `simpset` $*$ `thm list` $\longrightarrow$ `simpset`

Commutativity can be added without losing termination.

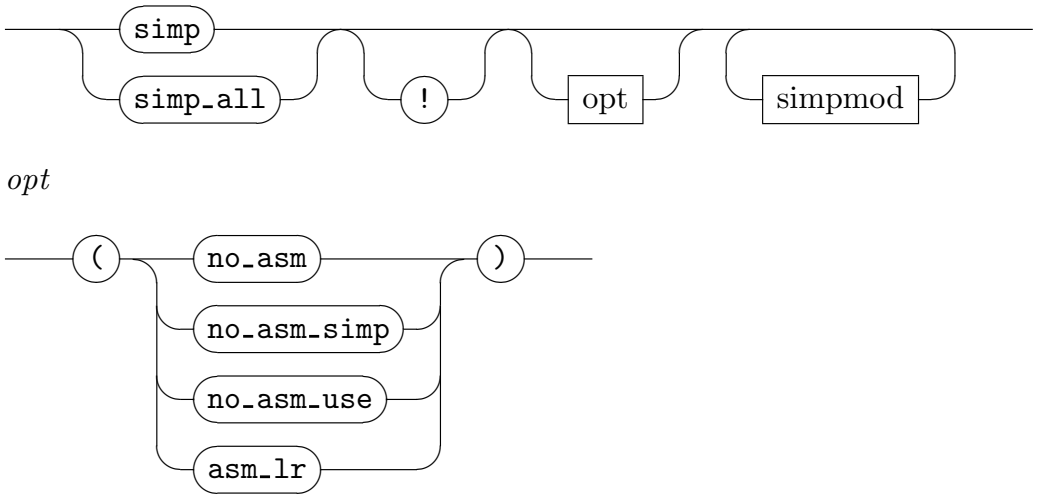
How to Apply the Simplifier?

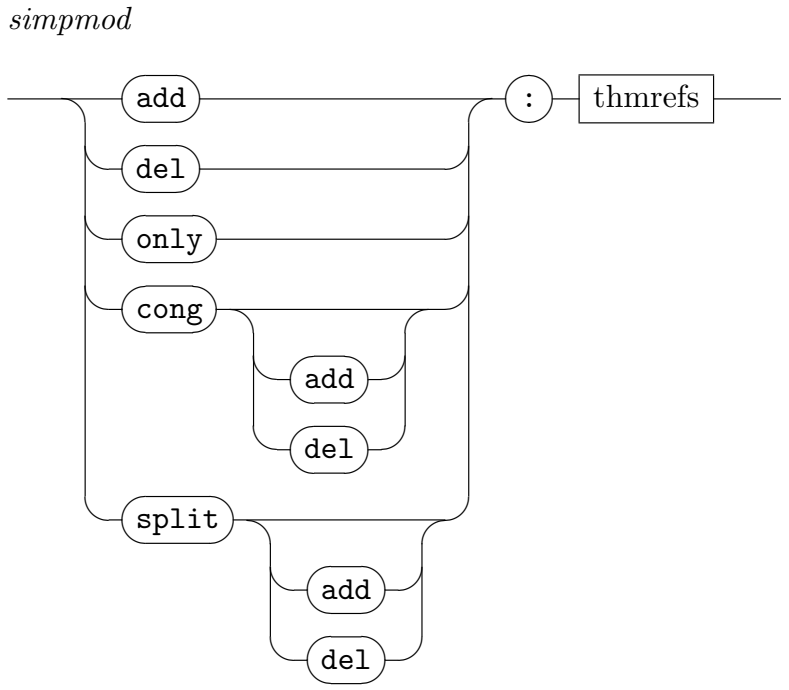
Several versions of the simplifier in the Isabelle engine (ML-level):

- `simp_tac` : `simpset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
- `asm_simp_tac` : `simpset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
  (includes assumptions into `simpset`)
- `asm_full_simp_tac` : `simpset` $\longrightarrow$ `int` $\longrightarrow$ `tactic`
  (rewrites assumptions, and includes them into simpset)

How to Apply the Simplifier?

On the ISAR level, these tactics are accessible as ISAR methods and have the following syntax:





Summary on the Simplifier and Term Rewriting

Simplifier is a powerful proof tool for

- conditional equational formulas
- ACI-rewriting
- quantifier reasoning
- congruence rules
- automatic proofs by case split rules

Fortunately, failure is quite easy to interpret since even intermediate results were computed and the solving process can be traced.

Summary on Last Three Sections

- Although Isabelle is an interactive proof construction, it is a flexible environment with powerful automated proof procedures.
- For classical logic and set theory, tableau-like procedures like `blast_tac` and `fast_tac` decide many tautologies.
- For equational theories (datatypes, evaluating functional programs, but also higher-order logic) `simp_tac` decides many tautologies (and is fairly easy to control).

More Detailed Explanations

$$0 + (x + 0) = x$$

Simplifying $0 + (x + 0)$ to x is something you have learned in school. It is justified by the usual semantics of arithmetic expressions. Here, however, we want to see more formally how such simplification works, rather than why it is justified.

Lists

Lists are a common datatype in functional programming. $[a, b, d, a, b]$ is a list. Actually, this notation is syntactic sugar for $a :: (b :: (d :: (a :: (b :: []))))$. Here, [] is the empty list and :: is a term constructor taking an alement and a list and returning a list. @ stands for list concatenation.

Intuitively, it is clear that $[a, b, d]$ concatenated with $[a, b]$ yields $[a, b, d, a, b]$.

Term constructor is usual terminology in functional programming. In first-order logic, we would speak of a function symbol. In the λ-calculus, we would speak of a (special kind of) constant (this will become clear later).

Functional Programming

For example, the lines

$$\begin{aligned} [] \mathbin{@} X &= X \\ (x :: X) \mathbin{@} Y &= x :: (X \mathbin{@} Y) \end{aligned}$$

define the list concatenation function @.

Rewrite Rules

An equational theory is a formalism based on equational rules of the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$.

A term rewriting system (to be defined shortly) is another formalism, based of rewrite rules. They also have the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$, but they have a different flavor in that $=$ must be interpreted as a directed symbol. One could also write $\rightsquigarrow$ instead of $=$ to emphasize this.

Matching

Given two terms s and t, a unifier is a substitution θ such that $s\theta = t\theta$.
A match is a substitution which only instantiates one of s or t, so $s\theta = t$ or $s = t\theta$ (one should usually clarify in the given context which of the terms is instantiated).

$a = b,\ a = c$

For a rewriting system consisting of rules $a = b$, $a = c$, one cannot rewrite $b = c$ to prove the equality, although it holds:

$$\dfrac{\dfrac{a = b}{b = a}\ \mathit{sym} \qquad a = c}{b = c}\ \mathit{trans}$$

Term Ordering

The biggest problem for term rewriting is (non-)termination. For some crucial rules, this problem is solved by ordered term rewriting. A term ordering is any partial order between ground (i.e., not containing free variables) terms.

One can define a term ordering by giving some function, called norm, from ground terms to natural numbers. Then a term is smaller than another term if the number assigned to the first term is smaller that the number assigned to the second term.

How Ordered Rewriting Solves ACI

Consider an equational theory consisting only of those rules (apart from *refl*, *sym*, *trans*, *subst*). Apart from that, the language may contain arbitrary other constant symbols. For such a language, it is possible to give a term ordering that will assign more weight to the same term on the left-hand-side of a $+$ than on the right-hand side. We can base such a term ordering on a norm. For example, the inductive definition of a norm $|\_|$ might include the line:

$$|s + t| := 2|s| + |t|$$

This means that if $|s| > |t|$, then $|s + t| = 2|s| + |t| > 2|t| + |s| = |t + s|$.
This has two effects:

- Applications of (A) or (I) always decrease the weight of a term

(provided the weight of s is > 0):

$$|(s+t)+r| = 2|s+t| + |r| = 4|s| + 2|t| + |r| \quad > \\ 2|s| + 2|t| + |r| = 2|s| + |t+r| = |s+(t+r)|.$$

- Applications of (C) are only possible if the left-hand side is heavier than the right-hand side.

Isabelle internally provides a term order, and the simplifier will use this in order to avoid non-termination for commutativity and similar rules.

Now, the question arises how ACI normal forms can be computed if commutativity is now longer a problem. The problem is that commutativity and idempotence patterns overlap and for the overlapping cases:

$$x + (x + y) = x + y$$
$$y + (x + z) = x + y + z$$

own rules must be derived. By Isabelle convention, these finitely many rules were stored in own rule sets such as Un_ac which can be accessed in ISAR.

Ambiguous Matching

For higher-order rewriting, it is very problematic to have rules containing terms of the form $F(G\,c)$ on the left-hand side, where F and G are free variables and c is a constant or bound variable. The reason can be seen in an example: Suppose you want to rewrite the term $f(g(h(i\,c)))$ where f, g, h, i are all constants. There are four unifiers of $F\,(G\,c)$ and $f(g(h(i\,c)))$:

$$\{f/F,\ (\lambda x.g(h(i\,x)))/G\},$$
$$\{(\lambda x.f(g\,x))/F,\ (\lambda x.h(i\,x))/G\},$$
$$\{(\lambda x.f(g(h\,x)))/F,\ (\lambda x.i\,x)/G\},$$
$$\{(\lambda x.f(g(h(i\,x))))/F,\ (\lambda x.x)/G\}.$$

$\forall, \exists$ is a Constant

Further examples:

- $(\exists x.Px \vee Qx) = (\exists x.Px) \vee (\exists x.Qx)$
- $(\exists x.P \longrightarrow Qx) = P \longrightarrow (\exists x.Qx)$
- $(\exists x.Px \longrightarrow Q) = (\forall x.Px) \longrightarrow Q$

In these examples, you may assume that first-order logic is our object logic.

On the metalevel, and hence also for the sake of term rewriting, $\forall, \exists$ are constants.

In the notation $(\forall x.Px \wedge Qx)$, the symbols P and Q are variables.

The principle was explained thoroughly before.

References

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proofs*. Academic Press, 1986.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[GM93] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[Klo93] Jan Willem Klop. *Handbook of Logic in Computer Science*, chapter "Term Rewriting Systems". Oxford: Clarendon Press, 1993.

[Pau03] Lawrence C. Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, March 2003.