Rewrite rules do not need to have variables; they can be ground. (i.e. Dec 2013 2(d))

REWRITING pt. 1 + simp

Term Rewriting

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

For example, the rules:

e.g.

$$x * 0 \Rightarrow 0 \qquad x + 0 \Rightarrow x$$

can be used to simplify an expression:

$$x + (x * 0) \longrightarrow x + 0 \longrightarrow x$$

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 $s \longrightarrow t$ 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).

Symbolic Computation

e.g.

Given this set of rules:

$$\begin{array}{llll} 0 + N & \Rightarrow & N & (1) \\ s(M) + N & \Rightarrow & s(M+N) & (2) \\ 0 * N & \Rightarrow & 0 & (3) \\ s(M) * N & \Rightarrow & (M * N) + N & (4) \end{array}$$

($s(x)$ means "successor of x", *i.e.* $1 + x$)

We can rewrite $2 * x$ to $x + x$:

$$\begin{array}{lll} & s(s(0)) * x & \\ \longrightarrow & (s(0) * x) + x & \text{by (4)} \\ \longrightarrow & ((0 * x) + x) + x & \text{by (4)} \\ \longrightarrow & (0 + x) + x & \text{by (3)} \\ \longrightarrow & x + x & \text{by (1)} \end{array}$$

The Power of Rewrites

e.g.

Given this set of rules:

$$\begin{array}{llll} 0 + N & \Rightarrow & N & (1) \\ (0 \le N) & \Rightarrow & \text{True} & (2) \\ s(M) + N & \Rightarrow & s(M+N) & (3) \\ s(M) \le s(N) & \Rightarrow & M \le N & (4) \end{array}$$

We can prove this statement:

$$\begin{array}{lll} & 0 + s(0) \le s(0) + x & \\ \longrightarrow & s(0) \le s(0) + x & \text{by (1)} \\ \longrightarrow & s(0) \le s(0 + x) & \text{by (3)} \\ \longrightarrow & 0 \le 0 + x & \text{by (4)} \\ \longrightarrow & \text{True} & \text{by (2)} \end{array}$$

STOP! bc. we cannot apply any more rules

Rewrite Rule of Inference

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

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

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

e.g. **Example**

Given an expression	$(s(a) + s(0)) + s(b)$
and a rewrite rule	$s(X) + Y \Rightarrow s(X + Y)$
we can find	$t = s(a) + s(0)$ ··· subexpression
and	$\theta = [a/X, s(0)/Y]$
to yield	$\boxed{s(a + s(0)) + s(b)}$

Restrictions

A rewrite rule $\alpha \Rightarrow \beta$ must satisfy the following restrictions:

1. ► $\boxed{\alpha \text{ is not a variable}}$

e.g. For example, $x \Rightarrow x + 0$ is not allowed. If the LHS can match anything, then it's very hard to control.

2. ► $\boxed{vars(\beta) \subseteq vars(\alpha).}$

e.g. 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.

LHS has no variables

More on Notation

- Rewrite rules: $L \Rightarrow R$, as we've seen already.
- Rewrite rule applications: $s \longrightarrow t$
  e.g., $s(s(0)) * x \longrightarrow (s(0) * x) + x$
- Multiple (*zero* or more) rewrite rule applications: $s \longrightarrow^* t$
  e.g., $s(s(0)) * x \longrightarrow^* x + x$
  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 $s \leftrightarrow u_1 \leftrightarrow \ldots \leftrightarrow u_n \leftrightarrow t$

Logical Interpretation

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.*, $a = b \bmod n$), but care is needed to make sure that rewriting corresponds to logically valid steps.

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

Without having A, we cannot talk about B;
Bc. of the nature of implication, we cannot replace A with B in $\neg A \wedge C$.

How to choose rewrite rules?

There are often many equalities to choose from:

$$X + Y = Y + X \qquad X + (Y + Z) = (X + Y) + Z \qquad X + 0 = X$$

$$0 + X = X \qquad 0 + (X + Y) = Y + X \qquad \ldots$$

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

e.g

Rules:

$$
\begin{array}{rl}
x * 0 \Rightarrow 0 & (1) \\
1 * x \Rightarrow x & (2) \\
x^0 \Rightarrow 1 & (3) \\
x + 0 \Rightarrow x & (4)
\end{array}
$$

Example:

$$
\begin{array}{rcl}
 & \underline{a^{2*0}} * 5 + b * 0 & \\
\longrightarrow & \underline{a^0} * 5 + b * 0 & \text{by (1)} \\
\longrightarrow & \underline{1 * 5} + b * 0 & \text{by (3)} \\
\longrightarrow & 5 + \underline{b * 0} & \text{by (2)} \\
\longrightarrow & \underline{5 + 0} & \text{by (1)} \\
\longrightarrow & 5 & \text{by (4)}
\end{array}
$$

Any subexpresson that can be rewritten (i.e. matches the LHS of a rewrite rule) is called a **redex** (*re*ducible *ex*pression).

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

Choices: Which redex to choose? Which rule to choose?

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

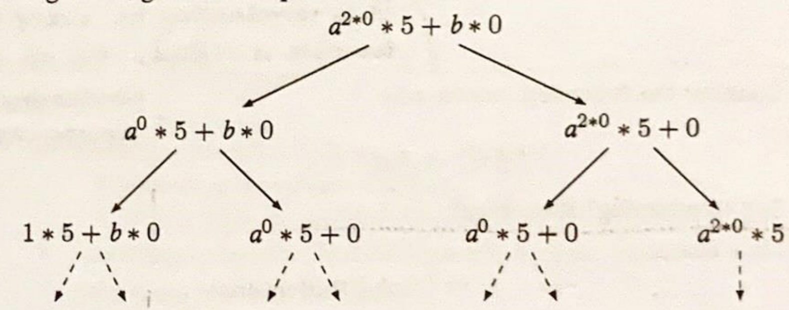
e.g. Examples of rules that *may* cause non-termination:

- Reflexive rules: *e.g.* $0 \Rightarrow 0$
- Self-commuting rewrites: *e.g.* $X * Y \Rightarrow Y * X$, 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).

The Rewrite Search Tree

In general, get a tree of possible rewrites:



- 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?)
    - It doesn't matter if expression rewrites to a canonical normal form

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:

$$
\begin{array}{rl}
x * 0 \Rightarrow 0 & (1) \\
1 * x \Rightarrow x & (2) \\
x^0 \Rightarrow 1 & (3) \\
x + 0 \Rightarrow x & (4)
\end{array}
$$

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))

Examples (from Past Exams)

- ① Consider the following rewrite rule:

  $$f(f(x)) \Rightarrow f(g(f(x)))$$

  Is it terminating? If so, why?

Define the measure of an expr. as the number of nested f application i.e. when you have 2 fs in a row

It is terminating bc. every time the rule is applied, the no. of consecutive f fns decreases. ✓

- ② How about:

  $$-(x+y) \Rightarrow (--x+y)+y$$

  where x and y are variables? Can you show that it is non-terminating?

This rule can be shown to be non-term. by exhibiting an infinite derivation. i.e. instantiating x to $--0+1$ and y to 1 to start w/, we have:

$$-((--0+1)+1) \rightarrow (--(\underbrace{--0}_{x}+1)+1)+1$$
$$\rightarrow (-((\underbrace{----0+1)+1}_{x})+1)+1$$
$$\rightarrow (((--(----0+1)+1)+1)+1)+1$$
$$\rightarrow \dots$$

The Isabelle Simplifier

The methods (tactics) `simp` and `auto`:

1) `simp` does automatic rewriting on the first subgoal, using a database of rules also known as a *simpset*.

2) `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 $P =$ True.
- Many rules are already added to the *default* simpset – so the simplifier often appears quite magical.

Interlude: Rewriting in Isabelle

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

$$\text{subst} \quad : \quad [\![?s = ?t; ?P\ ?s]\!] \Longrightarrow ?P\ ?t$$
$$\text{ssubst} \quad : \quad [\![?t = ?s; ?P\ ?s]\!] \Longrightarrow ?P\ ?t$$

→ whenever we have equation where an expr. s = expr. t and we know $P(s)$ holds, we also know $P(t)$ holds.

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

There is also a tactic that rewrites using a theorem:

apply (subst *theorem*)	: rewrites goal using *theorem*
apply (subst (asm) *theorem*)	: rewrites assumptions using *theorem*
apply (subst (i_1 i_2...) *theorem*)	: rewrites goal at positions $i_1, i_2, \dots$
apply (subst (asm) (i_1 i_2...) *theorem*)	: rewrites assumptions at positions $i_1, i_2, \dots$

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

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 → assms are simplified but are not used in the simplification of a/o or the conclusion
- `auto simp add: … del: …`

assumptions are not simplified but are used in the simplication of the conclusion

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 $[\![P_1; \dots; P_n]\!] \Longrightarrow s = t$ if
   - the lhs s matches some expression and
   - Isabelle can *recursively* prove $P_1, \dots, P_n$ by rewriting.

   $$\text{Example:} \quad \overbrace{[\![a \neq 0; b \neq 0]\!]}^{\textit{prove}} \Longrightarrow \overbrace{b/(a * b)}^{\textit{match}} = 1/a$$

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

   $$a + b \longrightarrow b + a \longrightarrow a + b \longrightarrow \dots$$

   Using $x + y = y + x$ as a rewrite rule is actually okay in Isabelle.

3. Case splitting:

   $$?P\,(\text{case } ?x \text{ of True} \Rightarrow ?f_1 \mid \text{False} \Rightarrow ?f_2)$$
   $$= ((?x = \text{True} \longrightarrow ?P\,?f_1) \wedge (?x = \text{False} \longrightarrow ?P\,?f_2))$$

   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