

Program Verification: Lecture 5

José Meseguer

University of Illinois at Urbana-Champaign (USA)

Subterms

In a Σ -term $f(t_1, \dots, t_n)$, the t_1, \dots, t_n are called its **immediate subterms**, denoted $t_i \triangleleft f(t_1, \dots, t_n)$, $1 \leq i \leq n$. Note that the inverse relation $\triangleleft^{-1} = \triangleright$ is *well-founded* (see STAC 11.1).

A term u is called a **subterm** of t iff $t \triangleright^* u$, and a **proper subterm** of t iff $t \triangleright^+ u$. Note that the relation \triangleright^+ is also well-founded and a strict order.

Notation: $T_{\Sigma(X)}^\circ =_{\text{def}} \bigcup_{s \in \mathcal{S}} T_{\Sigma(X)_s}$. Given a term $t \in T_{\Sigma(X)}^\circ$, we denote by $\text{vars}(t)$ the set of its variables, that is, $\text{vars}(t) = \{x \in \bigcup X \mid t \triangleright^* x\}$.

A term t may contain different **occurrences** of the same subterm u . For example, the subterm $g(a)$ appears twice in the term $f(b, h(g(a)), g(a))$.

Context-Subterm Decomposition of a Term

To indicate **where** a subterm u is located we can replace u by a *hole*, a new constant \square , added at the kind level to the signature Σ , marking **where** the subterm u was **before** we removed it.

For example, we can indicate the two places where $g(a)$ occurs in $f(b, h(g(a)), g(a))$ by $f(b, h(\square), g(a))$ and $f(b, h(g(a)), \square)$. A term with a **single occurrence of a hole** \square is called a **context**.

We write $C[\square]$ to denote a context. Given a context $C[\square]$ and a term u , we can obtain a new term, denoted $C[u]$, by **replacing** the hole \square by the term u . For example, if $C[\square] = f(b, h(\square), g(a))$ and $u = k(b, y)$, then $C[u] = f(b, h(k(b, y)), g(a))$.


Context-Subterm Decomposition of a Term (II)

Of course, if $C[\]$ is the context obtained from a term t by placing a hole \square where subterm u occurred, then we have the term identity $t = C[u]$.

That is, we can always **decompose** a term t into a context and a chosen subterm at a chosen occurrence, where if $t = C[u]$, then the decomposition of t into the context-subterm pair $(C[\], u)$ is succinctly indicated by the more compact notation $C[u]$.

For example, we have, among others, the following decompositions of our term $f(b, h(g(a)), g(a))$:

$$f(b, h([g(a)]), g(a)) = f(b, [h(g(a))], g(a)) = [f(b, h(g(a)), g(a))]$$

where the last decomposition has an “empty context” $[\]$ 

Equations and Equational Theories

Given a sensible order-sorted signature $\Sigma = ((S, <), F, G)$, a Σ -**equation** is an atomic formula $t = t'$, where $t, t' \in T_{\Sigma(X)}^{\circ}$, and where we require that $t = t'$ is **well typed**, in the sense that there are sorts $s, s' \in S$ such that $t \in T_{\Sigma(X),s}$, $t' \in T_{\Sigma(X),s'}$, and $[s] = [s']$.

An **equational theory** is then a pair (Σ, E) , with Σ and order-sorted signature, and E a set of Σ -equations.

In an equational theory (Σ, E) all equations $t = t' \in E$ are implicitly assumed to be **universally quantified** as

$$(\forall x_1 : s_1, \dots, x_n : s_n) t = t'$$

with $\text{vars}(t = t') = \{x_1 : s_1, \dots, x_n : s_n\}$, where, by definition, $\text{vars}(t = t') = \text{vars}(t) \cup \text{vars}(t')$.

Equational Deduction: Replacing Equals by Equals

Equational deduction is the **systematic replacement of equals by equals** using the given equations E .

For example, we may use ring theory equations such as:

(1) $x + y = y + x$, (2) $x * y = y * x$, (3) $(x + y) + z = x + (y + z)$,
 (4) $x + 0 = x$, (5) $x * 1 = x$, (6) $x * (y + z) = (x * y) + (x * z)$, to
 prove the polynomial equality $y + (z + (0 + (1 * x))) = (y + z) + x$
 by the following sequence of replacements of equals by equals:

$$\begin{aligned}
 (\dagger) \quad y + (z + [0 + (1 * x)]) &= y + (z + [(1 * x) + 0]) = y + (z + [1 * x]) = \\
 & y + (z + [x * 1]) = [y + (z + x)] = (y + z) + x
 \end{aligned}$$

where at each point the subterm where an equation is applied is marked by the term decomposition.

Equational Deduction: Replacing Equals by Equals (II)

We can make the above proof of equality (\ddagger) more informative by giving a name, say E , to the above set of equations (1)–(6), and indicating a proof step $t = t'$ by:

- applying an equation **from left to right** by $t \rightarrow_E t'$, and
- applying an equation **from right to left** by $t \leftarrow_E t'$.

With this notation we obtain the more informative proof:

$$\begin{aligned}
 y + (z + [0 + (1 * x)]) &\rightarrow_E y + (z + [(1 * x) + 0]) \rightarrow_E y + (z + [1 * x]) \rightarrow_E \\
 &y + (z + [x * 1]) \rightarrow_E [y + (z + x)] \leftarrow_E (y + z) + x.
 \end{aligned}$$

Term Rewriting

Certain equations, for example equations (3)–(6) in *ALG*, can be applied from left to right as **algebraic simplification rules**, because their righthand side is clearly simpler, so that applying them leads to a simpler expressions.

Algebraic simplification produces a special type of equational proofs, called **algebraic simplification proofs**, where equations are **always applied from left to right**. Here is an algebraic simplification proof with equations in *ALG* for a polynomial expression:

$$\begin{aligned}
 ([x+0]*(y+(z*1)))+x' &\rightarrow_E (x*(y+[z*1]))+x' \rightarrow_E [x*(y+z)]+x' \rightarrow_E \\
 &(((x*y)+(x*z))+x') \rightarrow_E (x*y)+((x*z)+x')
 \end{aligned}$$

This **left to right** process is called **term rewriting**, or **term reduction**.

Rewrite Rules and Term Rewriting Systems

We can make term rewriting explicit by **choosing and orientation** for an equation: we can orient an equation $t = t'$ from left to right as a so-called **rewrite rule** $t \rightarrow t'$, and from right to left as the rewrite rule $t' \rightarrow t$.

Definition

(Rewrite Rules and Term Rewriting Systems). Given a sensible order-sorted signature $\Sigma = ((S, <), F, G)$, a Σ -**rewrite rule** is a sequent $t \rightarrow t'$, where $t, t' \in T_{\Sigma(X)}^{\circ}$, and where we require that the rule $t \rightarrow t'$ is **well typed**, in the sense that there are sorts $s, s' \in S$ such that $t \in T_{\Sigma(X),s}$, $t' \in T_{\Sigma(X),s'}$, and $[s] = [s']$.

A **term rewriting system** is then a pair (Σ, R) , with Σ an order-sorted signature, and R a set of Σ -rewrite rules.

The Rewrite Relation

Definition

Let $\Sigma = ((S, <), F, G)$ be a sensible, kind-complete signature, let (Σ, R) be a term rewriting system, and let $Y = \{Y_s\}_{s \in S}$ be an S -indexed set of variables. Then an R -**rewrite step** is a pair (u, v) , denoted $u \rightarrow_R v$, such that $u, v \in T_{\Sigma(X)}^o$ and there is a rewrite rule $t \rightarrow t' \in R$, a substitution $\theta : \text{vars}(t \rightarrow t') \rightarrow T_{\Sigma(Y)}$, and a term decomposition $u = C[t\theta]$ such that $v = C[t'\theta]$, where, by definition, $\text{vars}(t \rightarrow t') = \text{vars}(t) \cup \text{vars}(t')$.

Since Σ is kind-complete, if $t \rightarrow t' \in R$ and $u = C[t\theta] : [s]$, then we must have $v = C[t'\theta] : [s]$, that is, \rightarrow_R **never produces ill-formed terms**.

We denote by \rightarrow_R^+ the transitive closure of \rightarrow_R , and by \rightarrow_R^* the reflexive-transitive closure of \rightarrow_R .

Rewrite Proofs

Definition

A (Σ, R) -**rewrite proof** is, by definition, either:

- a 0-step rewrite $t \rightarrow_R^* t$ for some term $t \in T_{\Sigma(X)}^\circ$ on some variables Y , or
- a sequence of R -rewrite steps of the form

$$t_0 \rightarrow_R t_1 \rightarrow_R t_2 \dots t_{n-1} \rightarrow_R t_n$$

with $n \geq 1$, witnessing $t_0 \rightarrow_R^+ t_n$.

The Equality Relation and Equational Proofs

The notion of an equational proof, that is, a sequence of steps of replacement of equals by equals using equations E , is a **trivial instance of the notion of a rewrite proof**.

Given an equational theory (Σ, E) , all we need to do is to consider proofs in the term rewriting system $(\Sigma, \vec{E} \cup \overleftarrow{E})$, where, by definition:

- \vec{E} is the set of left-to-right orientations
 $\vec{E} = \{t \rightarrow t' \mid t = t' \in E\}$; and
- \overleftarrow{E} is the set of right-to-left orientations
 $\overleftarrow{E} = \{t' \rightarrow t \mid t = t' \in E\}$.

The Equality Relation and Equational Proofs (II)

Definition

Given an equational theory (Σ, E) with Σ kind-complete and with nonempty sorts (i.e., $\forall s \in S, T_{\Sigma, s} \neq \emptyset$), an E -equality step is, by definition, a $(\vec{E} \cup \overleftarrow{E})$ -rewrite step $u \rightarrow_{(\vec{E} \cup \overleftarrow{E})} v$, denoted $u \leftrightarrow_E v$, where $u, v \in T_{\Sigma(X)}^\circ$ for some variables Y .

\leftrightarrow_E^+ denotes the transitive closure of \leftrightarrow_E ; and \leftrightarrow_E^* the reflexive-transitive closure of \leftrightarrow_E . \leftrightarrow_E^* is called the E -equality relation, and is often abbreviated to $=_E$. It is also called the relation of equality modulo E .

A (Σ, E) -equality proof is by, definition, either a 0-step E -equality $t \leftrightarrow_E^* t$ for some term $t \in T_{\Sigma(X)}^\circ$, or a sequence of E -equality steps of the form $t_0 \leftrightarrow_E t_1 \leftrightarrow_E t_2 \dots t_{n-1} \leftrightarrow_E t_n$, with $n \geq 1$, witnessing $t_0 \leftrightarrow_E^+ t_n$.

Term Rewriting Modulo Axioms

Certain equations are **intrinsically problematic** for term rewriting. For example, the commutativity equation $x + y = y + x$ is intrinsically problematic for rewriting because:

- we do not obtain a simpler term, but only a “mirror image” of the original term; for example, $(x * 7) + (0 * y)$ is rewritten to $(0 * y) + (x * 7)$; and
- even worse, we can easily *loop* when applying this equation, as in the infinite, alternating sequence

$$(x * 7) + (0 * y) \rightarrow_E (0 * y) + (x * 7) \rightarrow_E (x * 7) + (0 * y) \rightarrow_E \dots$$

The solution to this problem is to **build in** certain, commonly occurring equational axioms, such as the above commutativity axioms, so that rewriting takes place **modulo** such axioms.

Term Rewriting Modulo Axioms (II)

For example, we can decompose our equations E into a built-in, commutative part $C = \{x + y = y + x, x * y = y * x\}$ and the rest, say, $E_0 = \{(x + y) + z = x + (y + z), x + 0 = x, x * 1 = x, x * (y + z) = (x * y) + (x * z)\}$, and then rewrite with the equations in E_0 from left to right applying them, not just to the given term t , but to any other term t' which is **provably equal** to t by the commutativity axioms C .

This, more powerful rewrite relation is called **rewriting modulo C** , and is denoted $\rightarrow_{E_0/C}$. For example, we can simplify the expression $((0 + x) * ((1 * y) + 7)) + z$ to $(x * y) + ((x * 7) + z)$ in just four steps with $\rightarrow_{E_0/C}$ as follows:

$$\begin{aligned} ((0+x)*((1*y)+7))+z &\rightarrow_{E_0/C} (x*((1*y)+7))+z \rightarrow_{E_0/C} (x*(y+7))+z \rightarrow_{E_0/C} \\ &((x*y)+(x*7))+z \rightarrow_{E_0/C} (x*y)+((x*7)+z) \end{aligned}$$

Term Rewriting Modulo Axioms (III)

But why stopping with commutativity? How about associativity?
 An associativity (A) equation such as $(x + y) + z = x + (y + z)$ has no looping problems; but parentheses around associative operators are a nuisance and can block the application of equations.

For example, we can simplify to 0 the term $((x + y) + z) + -(y + (z + x))$ in **one** step of rewriting **modulo** the following set AC of associativity and commutativity axioms for $_ + _$ and $_ * _$, $AC = \{x + y = y + x, x * y = y * x, (x + y) + z = x + (y + z), (x * y) * z = x * (y * z)\}$, using the single equation $ALG_1 = \{x + -x = 0\}$ oriented as the rule $x + -x \rightarrow 0$.

$$((x + y) + z) + -(y + (z + x)) \rightarrow_{ALG_1/AC} 0.$$

That is, when rewriting modulo AC: (i) the **order** of the arguments does not matter (because of commutativity, C), and (ii) **parentheses do not matter** (because of associativity, A).

Rewrite Theories

Likewise, we could also build in the unit element axioms $U = \{x + 0 = x, x * 1 = x\}$. Or any combination of C , and/or A , and/or U axioms could be built in.

In fact, the idea of building in a set B of equational axioms, so that we rewrite with a set of rules R modulo B , is **entirely general**, and is associated to the notion of a **rewrite theory**.

Definition

Let Σ be a sensible order-sorted signature. A **rewrite theory** is a triple (Σ, B, R) , where B is a set of Σ -equations, and R is a set of Σ -rewrite rules.

Rewriting with R modulo B can then be formalized as follows.

Rewriting Modulo B

Definition

Let (Σ, B, R) be a rewrite theory such that Σ is sensible and kind-complete. Then an R -**rewrite step modulo B** is a pair $(u, v) \in T_{\Sigma(Y)}^{\circ} \times T_{\Sigma(Y)}^{\circ}$, denoted $u \rightarrow_{R/B} v$, such that there are terms $u', v' \in T_{\Sigma(Y)}^{\circ}$ such that:

$$u =_B u' \rightarrow_R v' =_B v.$$

We call $\rightarrow_{R/B}$ the *one-step R -rewrite relation modulo B* , and denote by $\rightarrow_{R/B}^0$ the relation $=_B$, called the **0-step R -rewrite relation modulo B** , by $\rightarrow_{R/B}^+$ the transitive closure of $\rightarrow_{R/B}$, and by $\rightarrow_{R/B}^{\star}$ the relation $\rightarrow_{R/B}^+ \cup (=B)$.

Rewrite Proofs Modulo B

Definition

An R -rewrite proof modulo B is either:

- a 0 -step R -rewrite modulo B of the form $u \rightarrow_{R/B}^0 v$, so that, by definition, $u =_B v$, for $u, v \in T_{\Sigma(Y)}$, or
- a sequence of R -rewrite steps modulo B of the form

$$v_0 \rightarrow_{R/B} v_1 \rightarrow_{R/B} v_2 \cdots v_{n-1} \rightarrow_{R/B} v_n,$$

$n \geq 1$, witnessing $v_0 \rightarrow_{R/B}^+ v_n$.

The Natural Numbers with a C Equality Predicate

The built-in module NAT protects BOOL. It simultaneously and efficiently supports both Peano (0 and s) and decimal notation. Here we add to it a **commutative** equality predicate.

```
fmod NAT-EQ is protecting NAT .
  op _.=._ : Nat Nat -> Bool [comm] .  *** equality predicate .
  vars N M : Nat .
  eq N .=. N = true .
  eq 0 .=. s(N) = false .
  eq s(N) .=. s(M) = N .=. M .
endfm
Maude> red s(0) .=. s(s(s(0))) .
result Bool: false
=====
Maude> red 7 .=. 13 .
result Bool: false
=====
red 1000000 .=. 1000000 .
result Bool: true
```

Example of Equational Simplification Modulo AU

```

fmod LIST-AU is protecting NAT-EQ .
  sort List . subsort Nat < List .
  op nil : -> List [ctor] .
  op _;_ : List List -> List [assoc id: nil ctor] .
  op _in_ : Nat List -> Bool .
  var N M : Nat . vars L L' : List .
  eq N in nil = false .
  eq N in L ; N ; L' = true .  *** not needed, but more efficient
  eq N in M ; L = if N .=. M then true else N in L fi .
endfm
Maude> red 7 in 3 ; 4 ; 9 .
result Bool: false
=====
Maude> red 7 in 4 ; 3 ; 7 .
result Bool: true

```

Example of Equational Simplification Modulo A

```

fmod LIST-A is protecting NAT-EQ .
  sort List .  subsort Nat < List .
  op nil : -> List [ctor] .
  op _;_ : List List -> List [assoc ctor] .
  op _in_ : Nat List -> Bool .
  var N M : Nat .  vars L L' : List .
  eq nil ; L = L .
  eq L ; nil = L .
  eq N in nil = false .
  eq N in M = N .=. M .
  eq N in N ; L = true .      *** not needed, but more efficient
  eq N in L ; N = true .     *** not needed, but more efficient
  eq N in L ; N ; L' = true . *** not needed, but more efficient
  eq N in M ; L = if N .=. M then true else N in L fi .
endfm
Maude> red 7 in 3 ; 4 ; 9 .
result Bool: false
=====
Maude> red 7 in 4 ; 3 ; 7 .
result Bool: true

```

Example of Equational Simplification Modulo *ACU* (III)

```
fmod MSET-ACU is protecting NAT-EQ .
  sort MSet .
  subsort Nat < MSet .
  op nil : -> MSet [ctor] .
  op _;_ : MSet MSet -> MSet [assoc comm id: nil ctor] .
  op _in_ : Nat MSet -> Bool .
  vars N M : Nat . var S : MSet .
  eq N in nil = false .
  eq N in N ; S = true . *** not needed, but more efficient
  eq N in M ; S = if N =. M then true else N in S fi .
endfm
Maude> red 7 in 3 ; 4 ; 9 .
result Bool: false
=====
Maude> red 7 in 4 ; 3 ; 7 .
result Bool: true
```

Example of Equational Simplification Modulo AC

```
fmod MSET-AC is protecting NAT-EQ .
  sort MSet .          subsort Nat < MSet .
  op nil : -> MSet [ctor] .
  op _;_ : MSet MSet -> MSet [assoc comm ctor] .
  op _in_ : Nat MSet -> Bool .
  vars N M : Nat . var S : MSet .
  eq nil ; S = S .
  eq N in nil = false .
  eq N in M = N .=. M .
  eq N in N ; S = true .  *** not needed, but more efficient
  eq N in M ; S = if N .=. M then true else N in S fi .
endfm
Maude> red 7 in 3 ; 4 ; 9 .
result Bool: false
=====
Maude> red 7 in 4 ; 3 ; 7 .
result Bool: true
```


Example of Equational Simplification Modulo AC

AC finite sets of naturals using identity and idempotency equations.

```
fmod NAT-SET is protecting NAT-EQ .
  sort NatSet .   subsort Nat < NatSet .
  op mt : -> NatSet [ctor] .
  op _ _ : NatSet NatSet -> NatSet [ctor assoc comm] .
  op _/\_ : NatSet NatSet -> NatSet [assoc comm] .
  vars X Y : NatSet .      vars N M : Nat .
  eq mt X = X .
  eq X X = X .
  eq mt /\ X = mt .
  eq N /\ M = if N .=. M then N else mt fi .
  eq N /\ (M X) = (N /\ M) (N /\ X) .
  eq (N X) /\ (M Y) = (N /\ M) (N /\ Y) (X /\ M) (X /\ Y) .
endfm
Maude> red (1 1 2 3 3 3 4 5 5) /\ (3 3 4 4 5 6 6 7 7) .
result NatSet: 3 4 5
```

Caveats on Equational Simplification Modulo U

Equational simplification **modulo identity** can be tricky. For example, the innocent-looking idempotency equation in

```
fmod NAT-SET-NONTERMINATING is protecting NAT .
  sort NatSet .
  subsort Nat < NatSet .
  op mt : -> NatSet [ctor] .
  op _ _ : NatSet NatSet -> NatSet [ctor assoc comm id: mt] .
  var X : NatSet .
  eq X X = X .
endfm
```

is **nonterminating**, since we have,

$$mt =_{ACU} mt \ mt \ \longrightarrow_E \ mt \ =_{ACU} \ mt \ mt \ \longrightarrow_E \ \dots$$

Caveats on Equational Simplification Modulo U (II)

Nontermination can be avoided by giving instead a more careful equation, where we restrict idempotency to pairs of elements (yet, with the same effect, since this ensures that all repeated elements will be eliminated) by means of the (now terminating) equation,

```
var N : Nat .
eq N N = N .
```

Another alternative is to declare:

```
sort NatSet NeNatSet .
subsort Nat < NeNatSet < NatSet .
op mt : -> NatSet [ctor] .
op _ _ : NatSet NatSet -> NatSet [ctor assoc comm id: mt] .
op _ _ : NeNatSet NeNatSet -> NeNatSet [ctor assoc comm id: mt] .
var X : NeNatSet .
eq X X = X .
```

Readings

All the theoretical aspects of the material presented in this lecture are covered in detail in *STAC* 13.1 and 13.2.