Formal Methods: Lecture 4

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Subsignatures

We have already seen an example of a signature extending another, namely $\Sigma(X)$ extends Σ. It is natural to call Σ a subsignature of $\Sigma(X)$. The general notion is as follows.

A signature Σ is a subsignature of a signature Σ', denoted $\Sigma \subseteq \Sigma'$, iff:

- the poset of sorts $(S, \leq)$ of Σ is contained in the poset of sorts $(S', \leq')$ of Σ', that is, $S \subseteq S'$ and $\leq \subseteq \leq'$, and
- the constants and operations of Σ are a subset of those of Σ', that is, for each $s \in S$ and $w \in S^*$ we have

$$\Sigma_{w,s} \subseteq \Sigma'_{w,s}.$$

Constructor Subsignatures

Let $T = (\Sigma, E)$ be an equational theory with equations E that are ground confluent and terminating. Then a subsignature $\Omega \subseteq \Sigma$ is called the constructor subsignature for $T = (\Sigma, E)$ iff:

1. $\Omega \subseteq \Sigma$
2. Ω has the same poset of sorts $(S, \leq)$ as Σ, and
3. Ω is the smallest signature satisfying (1) and (2) above and such that its terms contain all canonical forms, that is, for each sort s we have:

$$Can_{\Sigma,E,s} \subseteq T_{\Omega,s}$$

We call Ω a subsignature of free constructors if for each sort $s \in S$ we actually have an equality $Can_{\Sigma,E,s} = T_{\Omega,s}$.

Constructor Subsignatures (II)

In Maude, we have the option of indicating the constructor subsignature for a given Maude module by declaring the `ctor` attribute for the constants and operators in the constructor subsignature. For example, we can declare,

```
fmod NAT-MIXFIX is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s_ : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s M = s(N + M) .
endfm
```

to indicate the, in this case free, constructor subsignature.

Constructor Subsignatures (III)

But the subsignature of constructors is not always free. For example, in the module,

```
fmod NAT-3 is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s_ : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s M = s(N + M) .
  eq s s s 0 = 0 .
endfm
```

$Can_{\Sigma,E,\texttt{Natural}} = \{0, \texttt{s 0}, \texttt{s s 0}\}$. Therefore,
$Can_{\Sigma,E,\texttt{Natural}} \neq T_{\Omega,\texttt{Natural}}$

Constructor Subsignatures (VI)

Note that a given subsort overloaded operator may be a constructor, while another subsort overloaded version of the same operator may not be. For example, the module

```
fmod INTEGER is
  sorts Zero Natural NzNatural Negative NzNegative Integer .
  subsorts Zero NzNatural < Natural < Integer .
  subsorts Zero NzNegative < Negative < Integer .
  op 0 : -> Zero [ctor] .
  op s_ : Natural -> NzNatural [ctor] .
  op s_ : Integer -> Integer .
  op p_ : Negative -> NzNegative [ctor] .
  op p_ : Integer -> Integer .
  var I : Integer .
  eq p s I = I .
  eq s p I = I .
endfm
```

defines a free constructor subsignature for the integers.

Equational Simplification Modulo Axioms

In Maude, we allow certain equations, namely associativity, commutativity, and identity to be declared as attributes of operators by means of the `assoc`, `comm`, and `id:` attribute declarations in modules and theories.

Mathematically, this means that our theory is of the form $T = (\Sigma, E \cup A)$, with E the equations explicitly given in the module, and with A the equations implicitly declared by attributes such as `assoc`, `comm`, and `id:`, of operators.

Equational Simplification Modulo Axioms (II)

Given equational axioms A of this kind, we say that two terms t, t' are equal modulo the axioms A iff we can show that they are equal by rerplacing equals for equals using the axioms A, i.e., by equational deduction. We will study equational deduction in greater detail later, but we can illustrate its use here with examples for axioms A of our interest.

For example if A is the commutativity axiom for $_ + _$, that is the axiom $N + M = M + N$, then the following terms are all equal modulo commutativity:

$$(s(0)+0)+s(0) = s(0)+(s(0)+0) = s(0)+(0+s(0)) = (0+s(0))+s(0)$$

Equational Simplification Modulo Axioms (III)

Similarly, if A is the associativity axiom for $_ + _$, that is the axiom $A + (B + C) = (A + B) + C$, then the following terms are all equal modulo associativity:

$$0+(s(0)+(s(s(0)))+0)) = 0+((s(0)+s(s(0)))+0) = (0+(s(0)+s(s(0))))+0 =$$

$$(0 + s(0)) + (s(s(0))) + 0) = ((0 + s(0)) + s(s(0))) + 0$$

In general, such axioms A define an equivalence relation $t \equiv_A t'$ between terms. Then, given a term t we denote by $[t]_{\equiv_A}$, of just $[t]$ if A is understood, the set of all terms that can be shown equal to t using the axioms A, that is, $[t]$ denotes and equivalence class modulo A.

Equational Simplification Modulo Axioms (IV)

When performing equational simplification modulo a set of axioms A we are allowed to apply an equation to a term if we can apply it to an equivalent term modulo A.

For example, the equation $N + 0 = N$ can be applied to the term $0 + s(0)$ modulo commutativity of $_ + _$ because using commutativity we can prove $0 + s(0) = s(0) + 0$, and we can directly apply the above equation to $s(0) + 0$. Similarly, we can apply the same equation to the term $s(0) + (0 + s(0))$ modulo associativity of $_ + _$.

Examples of Equational Simplification Modulo A

Lists modulo associativity and identity, with membership:

```
fmod LIST-AID is
  protecting NAT .
  sort List .
  subsort Nat < List .
  op nil : -> List [ctor] .
  op _;_ : List List -> List [assoc id: nil ctor] .
  op _in_ : Nat List -> Bool .
  var N : Nat .  vars L L' : List .
  eq N in L ; N ; L' = true .
  eq N in L = false [owise] .
endfm
reduce in LIST-AID : 7 in 3 ; 4 ; 9 .
result Bool: false
==========================================
reduce in LIST-AID : 7 in 4 ; 3 ; 7 .
result Bool: true
```

Examples of Equational Simplification Modulo A (II)

Lists modulo associativity, with membership. More patterns need to be considered without the identity attribute.

```
fmod LIST-A is
  protecting NAT .  sort List .  subsort Nat < List .
  op nil : -> List [ctor] .
  op _;_ : List List -> List [assoc ctor] .
  op _in_ : Nat List -> Bool .
  var N : Nat .  vars L L' : List .
  eq nil ; L = L .
  eq L ; nil = L .
  eq N in N = true .
  eq N in N ; L = true .
  eq N in L ; N = true .
  eq N in L ; N ; L' = true .
  eq N in L = false [owise] .
endfm
```

```
reduce in LIST-A : 7 in 3 ; 4 ; 9 .
result Bool: false
==========================================
reduce in LIST-A : 7 in 4 ; 3 ; 7 .
result Bool: true
```

Examples of Equational Simplification Modulo A (III)

Multisets modulo associativity, commutativity, and identity.

```
fmod MSET-ACID is
  protecting NAT .
  sort MSet .
  subsort Nat < MSet .
  op nil : -> MSet [ctor] .
  op _;_ : MSet MSet -> MSet [assoc comm id: nil ctor] .
  op _in_ : Nat MSet -> Bool .
  var N : Nat .  var S : MSet .
  eq N in N ; S = true .
  eq N in S = false [owise] .
endfm
reduce in MSET-ACID : 7 in 3 ; 4 ; 9 .
result Bool: false
==========================================
reduce in MSET-ACID : 7 in 4 ; 3 ; 7 .
result Bool: true
```

Examples of Equational Simplification Modulo A (IV)

Multisets modulo associativity and commutativity.

```
fmod MSET-AC is
  protecting NAT .
  sort MSet .            subsort Nat < MSet .
  op nil : -> MSet [ctor] .
  op _;_ : MSet MSet -> MSet [assoc comm ctor] .
  op _in_ : Nat MSet -> Bool .
  var N : Nat .  var S : MSet .
  eq nil ; S = S .
  eq N in N = true .
  eq N in N ; S = true .
  eq N in S = false [owise] .
endfm
reduce in MSET-AC : 7 in 3 ; 4 ; 9 .
result Bool: false
==========================================
reduce in MSET-AC : 7 in 4 ; 3 ; 7 .
result Bool: true
```

Equational Simplification Modulo A

The above examples and `red` commands illustrate equational simplification modulo A, for A any combination of associativity, commutativity, and identity axioms.

Let $T = (\Sigma, E \cup A)$ be a theory whose equations E are admissible as equational simplification rules. Then, we can define a binary relation on terms in $T_{\Sigma(X)}$, denoted $\longrightarrow_{E/A}$, and called one-step equational simplification modulo A, by means of our two previously defined relations $\longrightarrow_E$, and $_ \equiv_A _$, as follows:

$$t \longrightarrow_{E/A} t' \quad \Leftrightarrow \quad t \equiv_A u \longrightarrow_E v \equiv_A t'.$$

Equational Simplification Modulo A (II)

Note that, denoting equivalence classes modulo A by $[t]$, simplification modulo A defines also a relation (with the same notation) on equivalence classes as follows:

$$[t] \longrightarrow_{E/A} [t'] \quad \Leftrightarrow \quad t \longrightarrow_{E/A} t'.$$

Conceptually, this is the best way of thinking of this form of equational simplification:

- we think of equivalence classes $[t]$ modulo A as abstract data structures (for example, strings for A associativity, and multisets for A associativity and comutativity)
- we think of $\longrightarrow_{E/A}$ as acting not on terms, but on such abstract data stuctures (for example, string rewriting, and multiset rewriting).

Another Example of Equational Simplification Modulo A

The following Maude module defines sets of natural numbers by simplifying multisets of natural numbers modulo associativity and commutativity, using identity and idempotency equations.

```
fmod NAT-SET is protecting NATURAL .
  sort NatSet .
  subsort Natural < NatSet .
  op empty : -> NatSet [ctor] .
  op _ _ : NatSet NatSet -> NatSet [ctor assoc comm] . *** set union
  var X : NatSet .
  eq empty X = X .                                     *** identity
  eq X X = X .                                         *** idempotency
endfm
```

Caveats on Equational Simplification Modulo A

Equational simplification modulo identity is trickier. For example, the innocent-looking idempotency equation in

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

is nonterminating, since, denoting by $\equiv_{ACI}$ the congruence modulo associativity, commutativity, and identity, we have,

$$\texttt{empty} \equiv_{ACI} \texttt{empty empty} \longrightarrow_E \texttt{empty} \equiv_{ACI} \texttt{empty empty} \longrightarrow_E \ldots$$

Caveats on Equational Simplification Modulo A (II)

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

```
var N : Natural .
eq N N = N .
```

All Generalizes Modulo A

Under reasonable assumptions on A, all the concepts on equational simplification generalize in a natural way to equational simplification modulo A. We of course define the relation $\longrightarrow^*_{E/A}$ as the reflexive and transitive closure of $\longrightarrow_{E/A}$.

The definitions of confluence and termination are the same, replacing $\longrightarrow_E$ by $\longrightarrow_{E/A}$.

All Generalizes Modulo A (II)

The concepts of canonical term algebra and of constructor subsignature also generalize, but now canonical forms are equivalence classes modulo A, (**Notation:** $Can_{\Sigma,E/A}$) and if Ω are the constructors we have for each sort,

$$Can_{\Sigma,E/A,s} \subseteq T_{\Omega/A,s}$$

where $T_{\Omega/A}$ is the Ω algebra whose elements are A-equivalence classes of terms in T_Ω and with operations defined by: $f([t_1],\ldots,[t_n]) = [f(t_1,\ldots,t_n)]$. We call the constructors free modulo A if for each sort we have

$$Can_{\Sigma,E/A,s} = T_{\Omega/A,s}$$

For example, the constructors in `NAT-SET` are not free modulo associativity and commutativity; for instance, the multiset `0 0 0` is not in canonical form.

Sufficient Completeness

We may claim that a certain signature Ω is the signature of constructors for a canonical term algebra $Can_{\Sigma/E}$, or, in the modulo A case, $Can_{\Sigma,E/A}$. But is this really true? Are the constructors really constructors?

It may be that, by mistake, we have given an incomplete equational definition of some function f in Σ which is not a constructor in Ω. Such non-constructor functions are called defined functions; for example $+$ and $*$ in `NATURAL`.

If all goes well, in the canonical term algebra we should have for each sort s, $Can_{\Sigma,E/A,s} \subseteq T_{\Omega/A,s}$, and in the modulo A case $Can_{\Sigma,E/A,s} \subseteq T_{\Omega/A,s}$. This means that in the canonical form of a ground term having a sort all defined functions should disappear: only constructors should remain.

Sufficient Completeness (II)

For example, consider the reverse function in the list module

```
fmod MY-LIST is protecting NAT .
  sorts NzList List .
  subsorts Nat < NzList < List .
  op _;_ : List List -> List [assoc] .
  op _;_ : NzList NzList -> NzList [assoc ctor] .
  op nil : -> List [ctor] .
  op rev : List -> List .
  eq rev(nil) = nil .
  eq rev(N:Nat ; L:List) = rev(L:List) ; N:Nat .
endfm
```

Are `nil` and `_;_` (plus `0` and `s`) really the constructors of this module as claimed?

Sufficient Completeness (III)

The answer is that they are not, as witnessed by:

```
Maude> red rev(7) .
reduce in MY-LIST : rev(7) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result List: rev(7)
```

The problem is that the above two equations would have been sufficient if we had also declared the `id: nil` attribute for `_;_` but do not fully define `rev` if only the `assoc` attribute is used. We shall say that a ground confluent and terminating theory (Σ, E) (resp. $(\Sigma, E \cup A)$) is sufficiently complete for a subsignature Ω on the same sorts S iff Ω is indeed a constructor signature for its canonical term algebra (resp. modulo A). In future lectures we shall see how to check this.

Sufficient Completeness (IV)

So, suppose we add an extra equation for `rev`

```
fmod MY-LIST is protecting NAT .
  sorts NzList List .
  subsorts Nat < NzList < List .
  op _;_ : List List -> List [assoc] .
  op _;_ : NzList NzList -> NzList [assoc ctor] .
  op nil : -> List [ctor] .
  op rev : List -> List .
  eq rev(nil) = nil .
  eq rev(N:Nat) = N:Nat .
  eq rev(N:Nat ; L:List) = rev(L:List) ; N:Nat .
endfm
```

Is now this module sufficiently complete?

Sufficient Completeness (V)

Indeed we now have

```
Maude> red rev(7) .
reduce in MY-LIST : rev(7) .
result NzNat: 7
```

But it is still not sufficiently complete, since

```
Maude> red nil ; 7 .
reduce in MY-LIST : nil ; 7 .
result List: nil ; 7
```

is not a constructor term, since `_;_` is a constructor on `NzList` but a defined function on `List`.

Sufficient Completeness (VI)

The really sufficiently complete specification is

```
fmod MY-LIST is protecting NAT .   sorts NzList List .
  subsorts Nat < NzList < List .
  op _;_ : List List -> List [assoc] .
  op _;_ : NzList NzList -> NzList [assoc ctor] .
  op nil : -> List [ctor] .
  op rev : List -> List .
  eq rev(nil) = nil .
  eq rev(N:Nat) = N:Nat .
  eq rev(N:Nat ; L:List) = rev(L:List) ; N:Nat .
  eq nil ; L:List = L:List .
  eq L:List ; nil = L:List .
endfm
```

Sufficient Completeness (VII)

```
Maude> red nil ; 7 .
reduce in MY-LIST : nil ; 7 .
result NzNat: 7
```

Readings and Exercises

Finish studying in detail Chapter 4 of the Maude 2.2 Manual. Study Sections 3.6 and 3.7 of Ölveczky's Lecture Notes.

Ex.6.1. Write four specifications computing your favorite functions on strings, multisets, and sets, so that you end up using all the different combinations of: (1) associativity; (2) associativity and identity; (3) associativity and commutativity; and (4) associativity, commutativity, and identity.