Executability Conditions

Given a rewrite theory \((\Sigma, B, R)\), which executabilty conditions should be placed on the rules \(R\) to effectively use it for equational simplification modulo \(B\) in the equational theory \((\Sigma, B \cup eq(R))\), in which the rules \(t \rightarrow t' \in R\) are now understood as equations \(t = t' \in eq(R)\)?

We will see that there are essentially four conditions needed:

1. each \(t \rightarrow t' \in R\) should be such that \(\text{vars}(t') \subseteq \text{vars}(t)\)
2. \(R\) is sort-decreasing
3. \(R\) is confluent modulo \(B\)
4. \(R\) is terminating modulo \(B\) (highly desirable but not essential), and

and will consider some variants of such conditions.
No Extra Variables in Lefthand Sides

Consider the rule $0 \rightarrow x \ast 0$. This rule is problematic we have to guess how to instantiate the variable $x$ in $x \ast 0$ before applying it, and there is an infinite number of instantiations.

Instead, the rule $x \ast 0 \rightarrow 0$ can be applied without problems, since the same substitution obtained by matching for the lefthand side can be reused to generate the righthand side replacement.

Therefore, we should require:

$$(1) \text{ for each } t \rightarrow t' \in R, \text{ any variable } x \text{ occurring in } t' \text{ must also occur in } t.$$
Sort Decreasingness

A second important requirement is:

(2) sort-decreasingness: for each $t \rightarrow t' \in R$, sort $s \in S$, and substitution $\theta$ we should have $t\theta : s \Rightarrow t'\theta : s$.

Prove by well-founded induction on the context $C$ below which a rewrite $C[t\theta] \rightarrow_R C[t'\theta]$ takes place, that under condition (2), if $u \rightarrow_R v$, then $u : s \Rightarrow v : s$.

To see why without sort-decreasingness things can go wrong, let $\Sigma$ have sorts $C$ and $D$ with $C < D$, a constant $c$ of sort $C$, a constant $d$ of sort $D$, and a subsort-overloaded unary function $f : C \rightarrow C$, $f : D \rightarrow D$. Let $B = \emptyset$ and $R = \{c \rightarrow d, f(f(x : C)) \rightarrow f(x : C)\}$. With the second rule $f(f(c))$ rewrites to $f(c)$, and then to $f(d)$ with the first rule. But if we apply the first rule to $f(f(c))$ we get $f(f(d))$, which cannot be further rewritten because sort information has been lost!
Checking Sort-Decreasingness

Sort decreasingness can be easily checked, since we do not need to check it on the (infinite) set of all substitutions \( \theta \). If 
\[
\{x_1 : s_1, \ldots, x_n : s_n\} = vars(t \rightarrow t'),
\]
we only need to check it on the (typically finite) set of substitutions of the form 
\[
\{(x_1 : s_1, x'_1 : s'_1), \ldots, (x_n : s_n, x'_n : s'_n)\}
\]
with \( s'_i \leq s_i, 1 \leq i \leq n \), called the sort specializations of the variables \( \{x_1 : s_1, \ldots, x_n : s_n\} \).

For example, for sorts \( Nat < Set \), with \( \_ \cup \_ \) set union, the rule 
\( x \rightarrow x \cup x \), with \( x : Set \), is not sort-decreasing, since for the sort specialization \( \{(x : Set, x' : Nat)\} \) we have 
\[
ls(x') = Nat < Set = ls(x' \cup x').
\]

**Exercise.** For \( \Sigma \) preregular, prove that the rules \( R \) are sort decreasing iff for each sort specialization \( \rho \) and for each \( t \rightarrow t' \) in \( R \) we have: \( ls(t\rho) \geq ls(t'\rho) \).
Determinism

A third requirement is determinism: if a term $t$ is simplified by $R$ modulo $B$ to two different terms $u$ and $v$, and $u \neq_B v$, then $u$ and $v$ can always be further simplified by $R$ modulo $E$ to a common term $w$.

This implies (Exercise!) that if $t \rightarrow_{R/B}^* u$ and $t \rightarrow_{R/B}^* v$, and $u$ and $v$ cannot be further simplified by $R$ modulo $B$, then we must have $u =_B v$. This is the idea of determinism: if rewriting with $R$ modulo $B$ yields a fully simplified answer, then that answer must be unique modulo $B$.

That is, the final result of a reduction with the rules $R$ modulo $B$ should not depend on the particular order in which the rewrites have been performed.
Determinism = Confluence

Determinism is captured by: (3) confluence. The rules $R$ of $(\Sigma, B, R)$ are confluent modulo $B$ iff for each $t \in \bigcup T_{\Sigma(Y)}$, whenever $t \rightarrow_{R/B}^* u$, $t \rightarrow_{R/B}^* v$, there is a $w \in \bigcup T_{\Sigma(Y)}$ such that $u \rightarrow_{R/B}^* w$ and $v \rightarrow_{R/B}^* w$. This can be described diagrammatically (dashed arrows denote existential quantification):

\[
\begin{array}{c}
t \\
| \quad | \\
| \quad | \\
R/B \\
| \quad | \\
u \quad v \\
| \quad | \\
* \\
R/B \\
| \quad | \\
R/B \\
| \quad | \\
w \\
* \\
R/B
\end{array}
\]

We call $R$ (3') ground confluent modulo $B$ if the above is only required for $t \in \bigcup T_{\Sigma}$. 
Joinability and the Church-Rosser Property

Call two terms \( t, t' \in \bigcup T_{\Sigma(Y)} \) joinable with \( R \) modulo \( B \), denoted \( t \downarrow_{R/B} t' \), iff \((\exists w \in \bigcup T_{\Sigma(Y)})\ t \rightarrow^*_{R/B} w \land t' \rightarrow^*_{R/B} w\).

**Exercise.** Prove that if \((\Sigma, E \cup B)\) satisfies the conditions of an order-sorted equational theory and the rules \( \vec{E} \) are confluent modulo \( B \), then the following equivalence, called the Church-Rosser property, holds for any two terms \( t, t' \in T_{\Sigma(Y)}\):

\[
t =_{E \cup B} t' \iff t \downarrow_{E/B} t'.
\]

where we abbreviate \( t \downarrow_{\vec{E}/B} t' \) to just \( t \downarrow_{E/B} t' \).
Termination

It is highly desirable that rewriting with $R$ modulo $B$ terminates.

**Definition**

Let $(\Sigma, B, R)$ be a rewrite theory. $R$ is called terminating or strongly normalizing modulo $B$ iff $\rightarrow_{R/B}$ is well-founded. $R$ is called weakly terminating or normalizing modulo $B$ iff any $t \in \bigcup T_{\Sigma(Y)}$ has a $R/B$-normal form, i.e., $\exists v \in \bigcup T_{\Sigma(Y)}$ s.t. $t \rightarrow^*_{R/B} v \land \forall w \in \bigcup T_{\Sigma(Y)}$ s.t. $v \rightarrow_{R/B} w$.

(Notation: $t \rightarrow^!_{R/B} v$).

Therefore, a highly desirable fourth requirement is:

(4) the rules $R$ are terminating modulo $B$, or at least the weaker requirement (4') that the rules $R$ are (ground) weakly terminating modulo $B$. 
Conditions on the Axioms $B$

Even with requirements (1)–(4) all satisfied, some further requirements should be placed on axioms $B$ so that they can be effectively “built in.”

- There should be a $B$-matching algorithm, that is, an algorithm such that, given $\Sigma$-terms $t$ and $t'$, gives us a complete set of substitutions $\theta$ such that $t \theta =_B t'$, or fails if no such $\theta$ exists. If $t \theta =_B t'$ holds, we say that $t'$ $B$-matches the pattern $t$.

- The variables in the axioms $B$ should all be at the kind level, i.e., of the form $x : [s]$, for $[s]$ a kind in $(S, <)$, so that the equations $B$ apply in their fullest possible generality.

- The equations $B$ should be $B$-preregular, in the sense that, given a $B$-equivalence class $[t]_B$, the set $\{s \in S \mid t' \in [t]_B \land t' : s\}$ has a minimum element, denoted $ls([t]_B)$. (Maude automatically checks $B$-preregularity for $B \subseteq ACU$).
The Canonical Term Algebra

Suppose \((\Sigma, E \cup B)\) is oriented as the rewrite theory \((\Sigma, B, \vec{E})\) and satisfies the executability conditions (1)–(4), or at least the slightly weaker (1)–(2), and (3')–(4').

Then, every term \(t \in \bigcup T_\Sigma\) can be simplified to a unique normal form \(\text{can}_{E/B}(t)\) modulo \(B\), called its canonical form, so that \(t \rightarrow^!_{E/B} \text{can}_{E/B}(t)\).

Furthermore, by the Church-Rosser property we have the following extremely useful equivalence for any \(t, t' \in \bigcup T_\Sigma\) (resp. \(t, t' \in \bigcup T_{\Sigma(Y)}\) if \((\Sigma, B, \vec{E})\) is confluent):

\[
t =_{E\cup B} t' \iff t \downarrow_{E/B} t' \iff \text{can}_{E/B}(t) =_B \text{can}_{E/B}(t').
\]

Therefore, to know if \(t, t'\) are provably equal in \((\Sigma, E \cup B)\), reduce them to canonical form and test if \(\text{can}_{E/B}(t) =_B \text{can}_{E/B}(t')\), which is decidable if \(B\) has a \(B\)-matching algorithm.
The Canonical Term Algebra (II)

This suggests considering the terms in $E/B$-canonical form as the values of an algebra.

Consider the example of an unsorted signature $\Sigma$ with a constant 0, a unary successor function $s$, and a binary addition function $\_ + \_$, and the equations: $E = \{ x + 0 = x, x + s(y) = s(x + y) \}$.

It is easy to check that the term rewriting system $(\Sigma, \vec{E})$ is confluent and terminating. It is also easy to check that the set of ground terms in $\vec{E}$-canonical form is the set $\text{Can}_{\Sigma/E} = \{ 0, s(0), s(s(0)), \ldots, s^n(0), \ldots \}$, that is the natural numbers in Peano notation.

This is a set of values, but for which algebra? Well, we can agree that the result of each operation on such values is, by definition, its $E/B$-canonical form. This is what the Maude $\text{red}$ command does!
The Canonical Term Algebra (III)

Here is the general definition:

**Definition**

Let \((\Sigma, E \uplus B)\) satisfy conditions (1)–(2), and (3‘)–(4‘). Then the \(S\)-indexed set \(\text{Can}_{\Sigma/E,B} = \{\text{Can}_{\Sigma/E,B,s}\}_{s \in S}\), where for each \(s \in S\) we define \(\text{Can}_{\Sigma/E,B,s} = \{[\text{can}_{E/B}(t)]_B \in T_{\Sigma,[s]/\equiv_B} \mid t \in T_{\Sigma,[s]} \land \exists t' \in [\text{can}_{E/B}(t)]_B, t' : s\}\), can be given a \(\Sigma\)-algebra structure called the **canonical term algebra** associated to \((\Sigma, E \uplus B)\) and denoted \(\mathcal{C}_{\Sigma/E,B} = (\text{Can}_{\Sigma/E,B}, \mathcal{C}_{\Sigma/E,B})\), where the structure map \(\mathcal{C}_{\Sigma/E,B}\) assigns to each \(f : w \rightarrow s\) in \(\Sigma\) the function \(f_{\mathcal{C}_{\Sigma/E,B}} : \text{Can}_w^{\Sigma/E,B} \rightarrow \text{Can}_{\Sigma/E,B,s}\) defined:

- for \(w = \text{nil}\), by \(f_{\mathcal{C}_{\Sigma/E,B}}(\emptyset) = \text{can}_{E/B}(f)\), and
- for \(w = s_1 \ldots s_n, n \geq 1\), by the function
  \[
  f_{\mathcal{C}_{\Sigma/E,B}} = \lambda([t_1]_B, \ldots, [t_n]_B) \in \text{Can}_{\Sigma/E,B,s_1} \times \ldots \times \text{Can}_{\Sigma/E,B,s_n} \cdot [\text{can}_{E/B}(f(t_1, \ldots, t_n))]_B.
  \]
The Idea of Sufficient Completeness

Consider the equations \( E = \{ x + 0 = x, x + s(y) = s(x + y) \} \) and observe that the set \( \text{Can}_{\Sigma/E} \) is precisely the set \( T_{DL} \) of terms in the signature \( \Sigma_{DL} \) with symbols 0 and \( s \). That is, the addition symbol has completely disappeared! This is as it should be, since the equations \( E = \{ x + 0 = x, x + s(y) = s(x + y) \} \) provide a complete definition of the addition function on natural numbers. Note that we have a strict inclusion \( \Sigma_{DL} \subset \Sigma \).

In general, if \( (\Sigma, E \cup B) \) satisfies (1)–(2) and (3′)–(4′), we can use operations in a subsignature \( \Omega \subseteq \Sigma \) as data constructors, so that the remaining operations in \( \Sigma - \Omega \) are functions operating on data built with the data constructors \( \Omega \) and returning as result another data value built with the constructors \( \Omega \).

The functions \( f \in \Sigma - \Omega \) are then completely defined if for each \( t \in \bigcup T_{\Sigma} \), we have \( \text{can}_{E/B}(t) \in \bigcup T_{\Omega} \).
Subsignatures

Before defining sufficient completeness we need to make more precise the notion of subsignature.

**Definition**

An order-sorted signature $\Omega = ((S', <'), G)$ is called a subsignature of an order-sorted signature $\Sigma = ((S, <), F)$, denoted $\Omega \subseteq \Sigma$, iff:

1. $S' \subseteq S$ and $<' \subseteq <$, and
2. for each $(w', s') \in List(S') \times S'$ there is a subset inclusion $G_{w',s'} \subseteq F_{w',s'}$, which we abbreviate with the notation $G \subseteq F$. 
Sufficient Completeness Defined

Definition

Let \((\Sigma, B, R)\) be a rewrite theory that is weakly ground terminating, and let \(\Omega \subseteq \Sigma\) be a subsignature inclusion where \(\Omega\) has the same poset of sorts as \(\Sigma\), that is, \(\Sigma = ((S, <), F)\), \(\Omega = ((S, <), G)\), and \(G \subseteq F\). We say that the rules \(R\) are sufficiently complete modulo \(B\) with respect to the constructor subsignature \(\Omega\) iff for each \(s \in S\) and each \(t \in T_{\Sigma,s}\) there is a \(t' \in T_{\Omega,s}\) such that \(t \rightarrow^!_{R/B} t'\).
More on Sufficient Completeness

If $\Sigma$ is kind-complete, then the above requirement that for each $t \in T_{\Sigma,s}$ there is a $t' \in T_{\Omega,s}$ such that $t \rightarrow^{1}_{R/B} t'$ should apply only to the sorts $s \in [s]$ in each connected component, but not to the kinds $[s]$. I.e., the sufficient completeness for $R$ modulo $B$ should required for a signature $\Sigma$ before kind-completing it to $\hat{\Sigma}$.

This is because, since terms that have a kind $[s]$ but not a sort $s$, correspond to undefined or error expressions, such as $p(0)$ for $p$ the predecessor function on natural numbers, it is perfectly possible that a completely well-defined function on the right sorts cannot be simplified away when applied to the wrong arguments.
More on Sufficient Completeness (II)

If \((\Sigma, B, E)\) has \(\Omega \subseteq \Sigma\) as a constructor subsignature with \(E\) confluent and weakly terminating modulo \(B\), we say that the constructors \(\Omega\) are free modulo \(B\) in \((\Sigma, B, E)\) iff for each sort \(s\) which is not a kind we have \(\text{Can}_{\Sigma/E,B,s} = T_{\Omega/B,s}\).

Therefore, if we have identified for our rewrite theory \((\Sigma, B, R)\) a subsignature of \(\Omega\) of constructors, a fifth and last requirement should be:

\(5\) the rules \(R\) are sufficiently complete modulo \(B\).
Examples of Sufficient Completeness Modulo $B$

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?
Examples of Sufficient Completeness Modulo $B$ (II)

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.

In future lectures we shall see how sufficient completeness can be automatically checked under reasonable assumptions.
So, suppose we add an extra equation for \texttt{rev}

\begin{verbatim}
  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
\end{verbatim}

Is now this module sufficiently complete?
Examples of Sufficient Completeness Modulo $B$ (IV)

Indeed we now have

Maude> red rev(7) .
reduce in MY-LIS

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.
Examples of Sufficient Completeness Modulo $B(V)$

The really sufficiently complete specification, making the constructors free modulo assoc, is

```plaintext
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.
  eq nil ; L:List = L:List.
  eq L:List ; nil = L:List.
endfm

Maude> red nil ; 7.
reduce in MY-LIST : nil ; 7.
result NzNat: 7
```
The following example shows an equational theory whose constructors are not free.

```plaintext
fmod NAT/3 is
  sorts Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
  eq s(s(s(0))) = 0 .
endfm
```