

Chapter 1: Term Rewriting

We are all by now quite used to the idea of modelling systems algebraically, in terms of (i) a signature representing in some sense the *interface* of some system, and (ii) an algebra, representing the *interpretation* of the signature. In general, we are quite happy to accept the idea that algebras are defined in terms of sets of *equations*. In this chapter, we consider some of the issues that arise when we represent algebras using equations, and then use software tools to apply these equations.

1.1. The Basic Problems: an Illustrative Example

Consider the Booleans with the following signature:

```
signature Boolean is
  sort Bool
  constants true, false : Bool
  operation not : Bool -> Bool
  operation or : Bool Bool -> Bool
end
```

Note that this is a "Maude-like" notation but it is not Maude! A typical set of equations for this might be:

```
vars X, Y, Z : Bool

not(not(X)) = X
not(true) = false
not(false) = true

X or X = X
X or Y = Y or X
X or true = true
X or false = X
(X or Y) or Z = X or (Y or Z)
```

Given an expression like `(not(true) or false) or (true or not(false))`, how would we typically go about evaluating it? We would search within the expression for *patterns* that match one side of the equations above, and we would then use that equation to modify the expression (typically by simplifying it). For example in the above case we might do:

```
(not(true) or false) -> not(true) /* since X or false = X */
not(true) -> false
```

Equally, we could have done:

```
(not(true) or false) -> false or false /* since not(true) = false */
false or false -> false
```

Now suppose we look at the other half of the expression:

```
(true or not(false)) -> (not(false) or true) /* since X or Y = Y or X */
(not(false) or true) -> true /* since X or true = true */
```

The mechanisation of expression evaluation based on this natural (for humans) method is called *term rewriting* - since we find and replace, or *rewrite*, terms using rules. However, there are some critical issues that lead to problems if we naively treat equations as rewrite rules.

1.1.1. Termination

In the second example above, where we rewrote `true or not(false)` we used the equation `X or Y = Y or X` to get `not(false) or true`. What would stop a machine, without intelligence, simply applying the same equation again to return to `true or not(false)`, resulting in a non-terminating sequence of rewriting. The answer is nothing - if we are careless enough to leave in equations like `X or Y = Y or X`. In the example above, the equation `(X or Y) or Z = X or (Y or Z)` will cause the same [potential] problem.

In general, we wish to avoid term rewriting systems that have this problem - we want our systems to be *terminating*.

1.1.2. Confluence

In the first example above, another [potential] problem occurs. There are two rules we can initially apply, leading to different intermediate terms. (The same is actually true of the second example: which other rule could we have applied?) In this case there is no problem: the second rule we apply in both cases leads to the same [final] result. However, consider the following set of rules:

A \rightarrow B
 AB \rightarrow AAB

In this example, we have a number of options when considering the sequence AAAB:

AAAB \rightarrow AABB \rightarrow AB BB \rightarrow BBBB
 AAAB \rightarrow AAAAB \rightarrow AAAAAB \rightarrow AAAAAAB \rightarrow . . .

Not only do these two sequences of rule applications not lead to the same result, the second one does not terminate. We say that systems that have the [desirable] property of converging on the same result, regardless of the choice of rules we apply, are *confluent* or *convergent*, or *Church-Rosser* (after the mathematician *Alonzo Church*, and his graduate student Berkeley Rosser). Commonly, only one sequence of rewrites is in some sense "correct". Again this is only a problem when we use an unintelligent machine: a human would simply avoid the incorrect rewriting sequence. (Of course in a complicated case, it may well take them some time to find the correct path!)

1.2. Implications: Equations, Rewrite Rules and some Terminology

There is a well-developed theory underlying rewriting. For the most part, we will not need to go into this in great detail, other than to note that term rewriting is provably equivalent to *Equational Logic*, and that a range of useful introductory papers can be found [here](#). However, a few aspects are interesting and important.

In general, and conceptually, the process of term rewriting continues until no further rules can be applied (or for ever, in the case of a non-terminating system). When we reach a point where we can apply no more rewrite rules, we have reached a *normal form*. If there is only one *unique* normal form, then our system of rewrite rules is confluent. In general, determining if a rewrite system is terminating and/or confluent is undecidable: fortunately, in the specific cases we are [often] interested in in practice, there are some useful rules to help us decide.

We have implied above that when we start to use software to evaluate expressions, there is some [subtle] difference between an *equation* and a *rewrite rule*. This is the case: an equation

$X = Y$

is a relation and is consequentially "non-directional": there are no restrictions on X or Y, and the equation $X=Y$ is either true or false.

In contrast, a rewrite rule

$X \rightarrow Y$

is a "directional" action: X can be replaced by Y, but *not* the other way around. Furthermore, there are some restrictions on X and Y:

- X cannot be a variable.
- Y cannot contain variables not in X. Note that the converse is not the case: X can contain variables not in Y.

Intuitively, these rules are to ensure that expressions "become simpler" during the rewriting process.

1.2.1. The Effect of the Difference Between Equations and Rewrite Rules

It may have occurred to you by this stage that the difference between equations and rewrite rules may have a significant impact on the implementation of systems: it is clear that we cannot just take a set of [arbitrary] equations and turn them into rewrite rules, expecting them to "work". Most obviously, associative and commutative equations must be removed, which will obviously have an effect. Also, the "one-way" nature of rewrite rules will have an effect. Both of these combine together to mean that when a set of equations is turned into a set of rewrite rules, some sequences of rewrites that were possible for an intelligent human using equations are no longer available to a machine using rewrite rules.

1.2.2. Completion Algorithms

The second problem - that of sets of rewrite rules not being as capable in some sense as the original equations - can be addressed by adding in further rewrite rules. These must obviously be chosen carefully, and this is typically done by applying *completion algorithms* of which the best known is the Knuth-Bendix algorithm. Such algorithms often have inconvenient properties (like no guarantee of termination). Knuth-Bendix has been implemented in *Maude*.

1.2.3. Associativity and Commutativity

The problem of associative and commutative operators is generally solved by "cheating" - that is, building in special mechanisms for dealing with them, and allowing operators to be marked with the respective properties. This is the approach that Maude takes. The usual terminology is to refer to rewriting in the absence of any commutative or associative operators as rewriting in the *free theory*. The addition of both associativity and commutativity is generally called *AC rewriting*. Incidentally, if you can get away without any AC rewriting, you will usually find things go quicker as it has a significant impact on efficiency.

1.3. In Practice...

How does this generally affect us in practice? If we are careful, not much. Typically, the sets of equations we come up with will (provided we are careful about commutative and associative operators) form a terminating and confluent set of term rewriting rules without any modifications. This in my opinion is because although we are intellectually aware we are dealing with sets of equations, our mind set is that of the programmer.

- When we write down equations, we are not normally thinking in terms of their bi-directionality, but of functionally mapping the left-hand side to the right-hand side: just as in a set of rewrite rules. This is the way programs work, and the way programmers think.
- We do not generally write down equation sets where more than one can be applied at a time. Because of our programming background and experience, we think in terms of a single series of steps which do not [non-deterministically] branch. Therefore, at any one time, there is usually only *one* equation that is applicable at a time.

Maude recognises this implicitly by providing two forms of module: *fmod* modules ("functional modules") where rewrite rules appear as equations, and systems must be terminating and confluent; and *mod* modules ("term rewriting modules"), where a \rightarrow symbol is used to denote rules, and systems *need not* be terminating and confluent. Needless to say, we are interested in *fmod* modules.

However, it is still possible that we may end up with non-terminating, non-confluent systems:

- By mistake - simply making errors in our systems of equations; and
- By messing about with "system algebras" - those algebras pre-defined to deal with, for example, vectors of bits. Care is needed here: for example, many second years in a recent exercise seemed to think subtraction was a commutative operation!

If you wish, you can [attempt to] check if your systems are terminating by using a *Maude tool*.