Quick Introduction to Term Rewriting

Paul Klint

2007-10-21 15:48:15 +0200 (Sun, 21 Oct 2007)

Table of Contents

Motivation

Term rewriting is a surprisingly simple computational paradigm that is based on the repeated application of simplification rules. It is particularly suited for tasks like symbolic computation, program analysis and program transformation. Understanding term rewriting will help you to solve such tasks in a very effective manner.

Introduction

Consider the following, *back-of-an-envelope*, calculation:

```
(9 - 5)² * (7 + 4) =
4²  * (7 + 4) =
16 * (7 + 4) =
16 * 11 =
176
```

This is a perfect example of *term rewriting*: the initial expression $(9 - 5)^2 * (7 + 4)$ is simplified in a number of steps using the rules of elementary arithmetic. The result is the number `176`.

Many forms of simplification or symbolic manipulation can be expressed in this way. Recall the simplification rule

```
(a + b)² = a² + 2ab + b²
```

from high school algebra or the rule to calculate the derivative of the sum of two functions u and v:

```
d(u + v)/dx = du/dx + dv/dx
```

In both cases, there is a complex *left-hand side* that can be simplified into the expression appearing at the *right-hand side*. These expressions are called *term*s and this explains the name term rewriting. The precise form of such a term may differ and is explained below. Observe that some of the items

on the left-hand side re-appear at the right-hand side (e.g., a, b, u, v); they are called *variables* and are used during the simplification.

A simple view on term rewriting is shown in Figure 1.1, "The rewriting process" (page 2). Given a set of rewrite rules and an initial term T, the rewriting algorithm is applied and will yield a simplified (or normalized) term T' as answer. T' is called the *normal form* of T. Later (in the section called "The term rewriting algorithm" (page 3) we will further explain the internals of the term rewriting algorithm.

Figure 1.1. The rewriting process
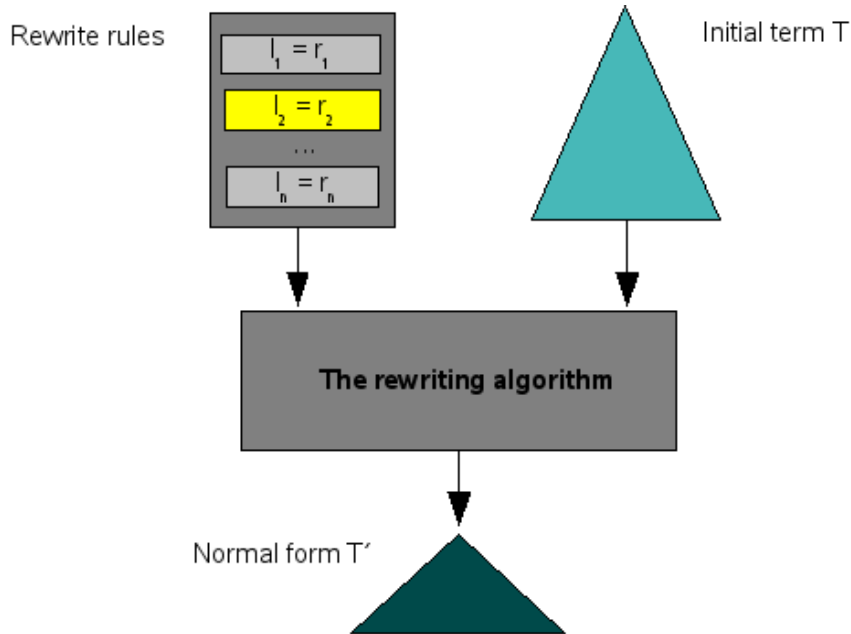
In order to apply such a simplification rule (or better: *rewrite rule*), several ingredients play a role:

- an initial expression (a term) that is to be simplified.

- a subexpression where the rule might be applied (this is called the *redex*); there must be a *match*, between the redex and the left-hand side of the rule we want to apply; finding a match means that we can find values for the variables in the left-hand side of the rule that make the left-hand side and the redex identical.

- next, the redex in the initial expression is replaced by the right-hand side of the rule, after replacing the variables by the values found during the match.

Term rewriting is a simplification process that starts with one or more rewrite rules and an initial term. This initial term is gradually reduced to a term that cannot be further simplified: this is the outcome of the rewriting process and is called the *normal form*.

In each step, a redex is searched for in (the possibly already simplified) initial term and it is replaced by the rewrite rule that can be applied.

Basic concepts

In order to understand term rewriting three basic concepts are essential: terms, substitution and matching.

Terms

Traditionally, terms are defined in a strict prefix format:

- A single variable is a term, e.g. `X`, `Y` or `Z`.

- A function name applied to zero or more arguments is a term, e.g., `add(X,Y)`.

Using this definition, complex hierarchical structures of arbitrary depth can be defined.

Occasionally, we will relax this definition of terms a bit and use a more flexible notation. Instead of `add(X,Y)` we will, for instance, also write `X + Y`.

Substitution

A substitution is an important auxiliary notion that we need on our road towards understanding term rewriting. A substitution is an association between variables and terms. For instance,

```
{ X -> 0, Y -> succ(0) }
```

is a substitution that maps the variable `X` to the term `0` (a function without arguments), and the variable `Y` to the term `succ(0)`.

Substitution can be used to create new terms from old ones. Using the above substitution and applying it to the term `mul(succ(X), Y)` will yield the new term `mul(succ(0), succ(0))`. The basic idea is that variables are replaced by the term they are mapped to by the substitution.

Matching

Matching has as goal to determine whether two terms T_1 and T_2 can be made equal. More precisely, whether a substitution exists that can make T_1 equal to T_2. For instance, the two terms `mul(succ(X), Y)` and `mul(succ(0), succ(0))` match since we can use the substitution { X -> 0 Y -> succ(0) } to make them identical. If no such substitution can be found, the two terms cannot be matched.

The term rewriting algorithm

We can now assemble the ingredients we have described so far into a comprehensive description of the rewriting algorithm ((page 4)).This algorithm still leaves some aspects unspecified:

- In what way is the redex selected?

- In what order are the rules applied?

Various methods are possible for selecting the redex:

- From the top (root) of the term to the bottom or from the leaves to the root.

- From left to right or from right to left.

We will mostly be using a leaves-to-root and left-to right order. This is called the *left-most innermost reduction strategy*.

Also various methods exist for selecting the rules to be tried:

- Textual order.

- Specificity order (rules with more precise left-hand sides are tried before rules with more general left-hand sides).

- No order.

We will be using no ordering of the rules.

Figure 1.2. The rewriting algorithm
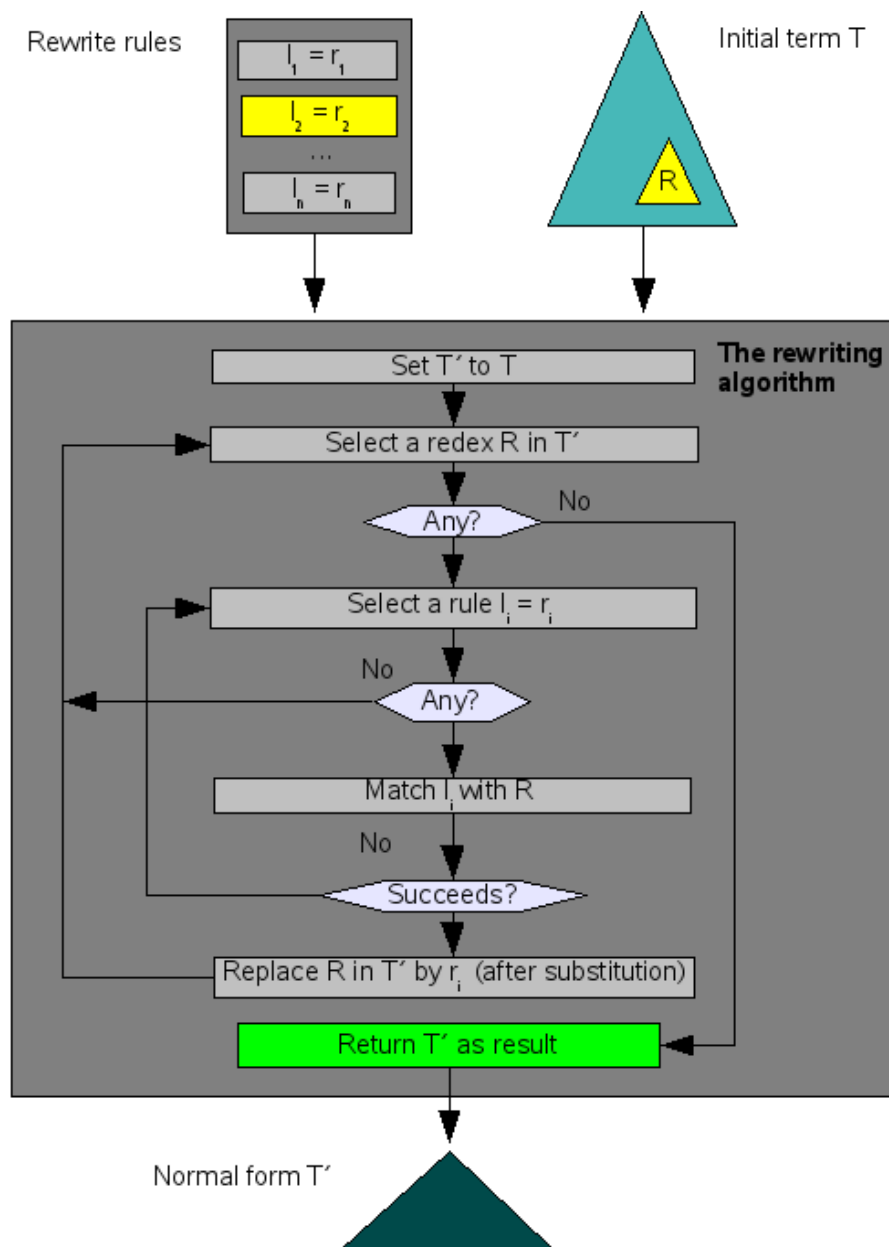
Examples

Let´s consider two simple examples of term rewriting systems for natural numbers and booleans. More elaborate examples that fall outside the scope of this article are:

- Typechecking of programs.

- Interpretation (= execution) of programs.

- Code generation.

- Fact extraction from source code for the benefit of quality assurance or defect detection.

- Software transformation for the benefit of translation or code improvement.

See the section called "Further Reading" (page 7) for pointers to such examples.

Numerals

How can we specify natural numbers with 0, successor, addition and multiplication using term rewriting? A usual approach is as follows:

```
[add1] add(0, X) = X

[add2] add(succ(X), Y) = succ(add(X, Y))

[mul1] mul(0, X) = 0

[mul2] mul(succ(X), Y) = add(mul(X, Y), Y)
```

Here, `0` represents the constant zero, and `add` and `mul` the arithmetic operations addition and multiplication. Natural numbers are represented as `0`, `succ(0)`, `succ(succ(0))`, and so on. In other words, *N* applications of the successor function `succ` to the constant `0`, represent the number *N*. In the above rules, `X` and `Y` are used as variables.

For mere mortals, the above rules could also be written in a more readable form:

```
[add1'] 0 + X = X

[add2'] succ(X) + Y = succ(X + Y)

[mul1'] 0 * X = 0

[mul2'] succ(X) * Y = mul(X * Y) + Y
```

Now let's follow the simplification of `add(succ(succ(0)), succ(succ(0)))` as shown in Figure 1.3, "Rewriting an arithmetic term" (page 5). Each redex is shown as a yellow box with a thick border. An arrow connects the redex with its replacement that is surround by a rectangle with a thin border.

The name of the rule that is applied in each step is shown on the right of each vertical arrow. As expected, the answer is `succ(succ(succ(succ0))))` or, in other words, 2 + 2 equals 4.

Figure 1.3. Rewriting an arithmetic term
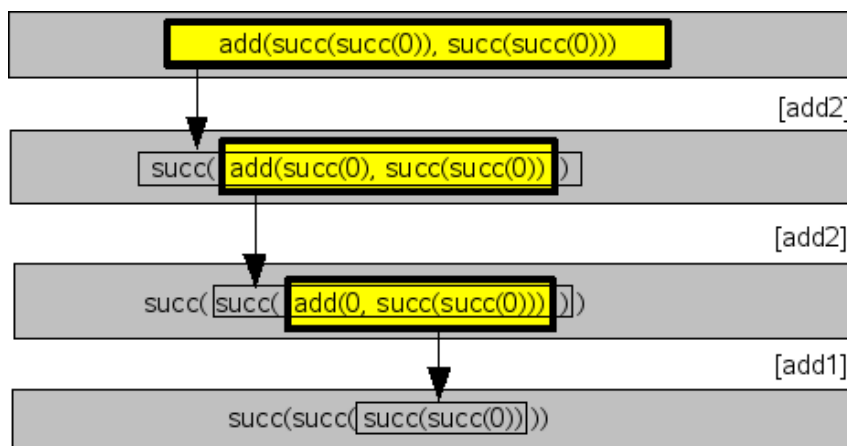
Booleans

How can we define Boolean expressions with constants `true` and `false` and the operations `and`, `or` and `not`? Consider the following definition that strictly follows the truth table definitions of Boolean operators:

```
[or1] or(true, true) = true
[or2] or(true, false) = true
[or3] or(false, true) = true
[or4] or(false, false) = false

[and1] and(true, true) = true
[and2] and(true, false) = false
[and3] and(false, true) = false
[and4] and(false, false) = false

[not1] not(true) = false
[not2] not(false) = true
```
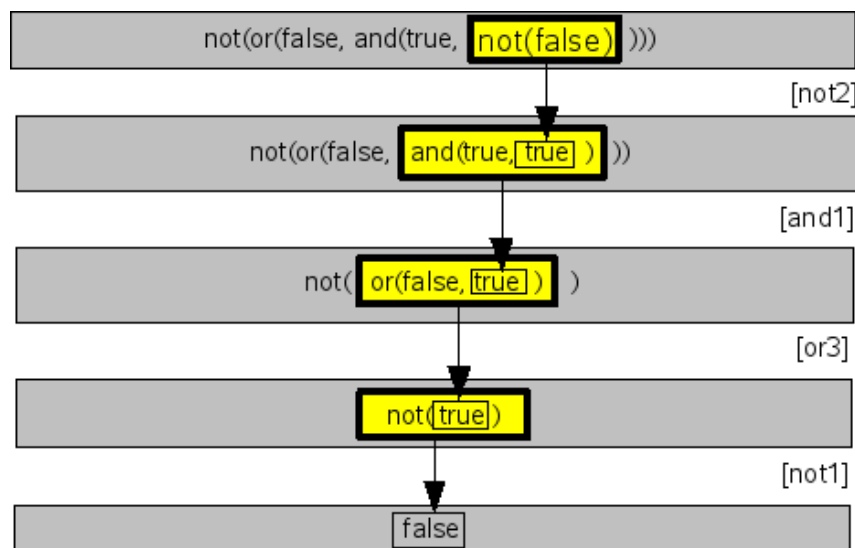
Consider the simplification of the Boolean expression `not(or(false, and(true,not(false))))` as shown in Figure 1.4, "Rewriting as Boolean term" (page 6).

Figure 1.4. Rewriting as Boolean term

Extensions of term rewriting

Term rewriting has been extended in various ways to make it even more practical. These extensions include:

- *User-defined syntax*. Instead of writing `and(true, false)` we would rather like to write `true & false`. We want to relax the strict prefix format of functions and use arbitrary notation for them.

- *Conditional rules*. So far we have only seen rewrite rules that are applied when they *can* be applied. To conditional rewrite rules, one or more conditions are attached that are first evaluated in order to determine whether the rule should be applied at all.

- *Default rules*. So far we have used no ordering for the rewrite rules. Default rules are only applied when no other rules can be applied an capture the idea of a *catch-all rule* that covers the cases that are not addressed by other, more specific, rules.

- *Lists and list matching*. Terms can be extended with a notion of lists. This is particularly handy when representing repeated items in a structure such as a list of identifiers in a declaration, or a list of statements in a procedure body. The matching between two terms has to be extended to cover the case of matching lists of terms rather than a single term. In this way, one can, for instance, define patterns that match a list of statements that contains a statement of a particular form.

- *Traversal functions*. In many problem domains it may occur that a large structure has to be searched while only a few elements are relevant for the operation at hand. Consider counting goto statements in C or Cobol programs: we are only interested to do something when we encounter a goto statement and we just want to travel over the other statements kinds. Traversal functions automate this kind of behaviour and can hugely reduce the number of rewrite rules that are needed in industrial size applications.

All these -- and more -- extensions are provided by the ASF+SDF formalism that is supported by The Meta-Environment [http://www.meta-environment.org].

The role of term rewriting in The Meta-Environment

The Meta-Environment supports a language called ASF+SDF that supports, amongst others, defining and executing rewrite rules. One of the distinguishing features of ASF+SDF is that rewrite rules may be written using arbitrary (user-defined) syntax. Continuing the simple example of the Booleans above, this means that we are not limited to writing `and(true,false)` but can use any syntax we want for the `and` function, and for all other functions of our liking.

This does not yet sound very exciting, right? But consider that we are working on something serious like analyzing Java programs. In that case, we have to write rewrite rules that describe the kind of analysis we want to do, say, finding calls to and data exchanges with untrusted components. For all Java statements that are involved in this analysis, we write rules that extract the desired information. The nice thing is that we can write these rules using ordinary Java text and that we don't have to resort to some complicated abstract syntax tree representation as is usual in compiler-based systems. In ASF+SDF the text *is* the tree!

How do the pieces now fit together? The source text of the Java program is first parsed (parsing is explained elsewhere) and converted to a term. Next, we can apply an analysis function to the Java program and get the results we want. It is as simple and exciting as that.

Further Reading

- Several articles on th website (www.meta-environment.org [http://www.meta-environment.org] give detailed examples of term rewriting applications.

- The book [BN98] gives an introduction to the theory of term rewriting.

- The book [Ter03] gives a comprehensive overview of research in this area.

- The Meta-Environment [http://www.meta-environment.org] is the tool suite par excellence for exploring applications of term rewriting.

Bibliography

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press1998.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press2003.