A Term Rewriting Scheme for Function Symbols with Variable Arity

Manfred Widera, Christoph Beierle
Praktische Informatik VIII
FernUniversität Hagen
58084 Hagen

Abstract

Term rewriting is used for programming tasks consisting of transformations of terms to simpler equivalent terms. In this context associative and commutative functions appear quite often. However, their transformations may be hard to understand when associativity and commutativity are not explicit in the transformation rules but coded into an additional set of equations. In this paper we present a notation for rewriting rules that makes associativity and commutativity explicit in the rewriting rules. As they are widely used in functional programming languages we allow the use of functions with variable number of arguments. Furthermore, our concept supports the commutative choice of certain arguments in a function call.

Contents

1 Introduction

Term rewriting systems as described in detail e.g. in [1] are widely studied as a tool of analyzing equivalence of terms. Ideally, they provide a simple formalism to reduce semantic equivalence to syntactic equality of the normal forms.

To use term rewriting systems (trs) in that way one usually needs a set of equations that describe an equivalence theory. From that set of equations a set of rewriting rules can be generated automatically quite often [6].

Furthermore, term rewriting systems seem to be adequate to describe equivalence transformations on an abstract level and give an efficient implementation, provided that the used trs are confluent and terminating. In contrast to the first view one is not only interested in the question whether two terms are equivalent but one wants to use the equivalence theory to transform given terms to equivalent ones with certain properties. E.g. the equation

$$x + (-x) = 0$$

not only states its left hand side and its right hand side as equivalent, but when performed from left to right on an arbitrary term the result is an equivalent term with no unnecessary sums of inverse numbers.

A special problem in term rewriting is given by associative and commutative functions. The solutions on this problem in literature e.g. [7], [4] usually spot on the first application type of term rewriting and split the set of equations into two sets R and T where R is transformed to a set of rewriting rules. T is handled between the rewriting steps, e.g. in the unification algorithm and usually contains equations for associativity and commutativity of several functions.

In the context of programming by trs this splitting makes it harder to understand the transformations that are performed by a pair (R, T), since the influence of the theory coded in T is not explicitly visible in the rules of R. E.g. consider the following system:

$$T := \{f(x, f(y, z)) = f(f(x, y), z)\},\ R := \{f(x, f(y, g(x))) = y\}$$

where f could be the addition $+$ and g could be unary minus $-$. The transformation

$$f(a_1, f(a_2, f(a_3, f(a_4, f(g(a_2), f(a_5, a_6)))))) \to f(a_1, f(a_3, f(a_4, f(a_5, a_6))))$$

can be performed by the system, but this is hard to see for the user, because of the necessary transformations

$$\begin{array}{rl} & f(a_1, f(a_2, f(a_3, f(a_4, f(g(a_2), f(a_5, a_6)))))) \\ =_T & f(a_1, f(\underline{f(a_2, f(f(a_3, a_4), g(a_2)))}, f(a_5, a_6))) \end{array}$$

before and

$$f(a_1, f(\underline{f(a_3, a_4)}, f(a_5, a_6))) =_T f(a_1, f(a_3, f(a_4, f(a_5, a_6))))$$

after the transformation.

In studies about theoretical properties of term rewriting in the AC theory, associative and commutative operators are handled as if they were transformed to functions of variable arity by an operation called flattening in [5]. This paper introduces a formalism that allows to use the view of flattened functions directly in rewriting rules, thus making them understandable more easily. Expressing function applications as in LISP the transformation above will be written as

$$(f\ a_1\ a_2\ a_3\ a_4\ (g\ a_2)\ a_5\ a_6) \rightarrow_R (f\ a_1\ a_3\ a_4\ a_5\ a_6)$$

A further benefit of our new formalism is the opportunity to restrict the use of associativity and commutativity to rules that depend on these properties and thereby to increase the performance of a trs.

The rest of the paper is organized as follows: In Sec. 2 we define the extended rule syntax providing tools for expressing functors of variable arity, commutative choice and the blocking of empty rule execution. The relation to the standard syntax of trs is given in Sec. 3 on a couple of examples. Section 4 discusses the influence of the extended syntax on the analysis of termination and confluence and in Sec. 5 we give a conclusion.

2 The new Rule Syntax

As mentioned before the main idea for expressing associativity is to permit functions with variable arity. Syntactic patterns for expressing a commutative choice are defined afterwards. In a last step we address the newly arisen problem of empty rule execution and provide a tool to solve it.

2.1 Functions of Variable Arity

Let f be a function symbol denoting an associative function. Then the execution order of nested applications of f is irrelevant. Thus, we can write nested occurrences of f as one application to some f' that takes an arbitrary number of arguments and applies f to them in an arbitrary but fixed order, e.g. from left to right. We use LISP notation for applications of

f', i.e. an application is written as parentheses enclosing the function symbol followed by the arguments separated by spaces. Instead of writing e.g.

$$(((a + (b + c)) + (d + (e + f))) + g)$$

we then could write $(+'\ a\ b\ c\ d\ e\ f\ g)$ with

$$\begin{aligned} (+'\ x\ y) &= x + y \\ (+'\ y_1 \dots y_k\ x) &= (+'\ y_1 \dots y_k) + x\ (k \geq 2) \end{aligned}$$

Note that we usually identify f' with f and write f for both the original function and the flattened function of variable arity.

Though the number of arguments to a function can become arbitrarily high, a rewriting rule will always affect just a certain finite number of them, or it will perform the same operation on all of them. We therefore need syntactical constructs that allow us to group arguments that are either the destination to a uniform change or are not affected by the rule at all. These constructs are defined in the following:

- For $i \in \{1, 2, \dots\}$ the pattern $<i \dots>$ represents an argument list of variable arity. A pattern $<i \dots>$ is always instantiated with the sequence of its elements without parentheses. When showing substitutions in examples we will use square brackets [] to make the start and the end of a list visible. The concatenation of lists is just written as $<i \dots><j \dots>$. Note that the whole expression $<i \dots>$ is handled as a single syntactic keyword. Several of these expressions can occur in one rule and have to be distinguished from each other by i. i is called *list counter*. Two occurrences $<i \dots>$ and $<j \dots>$ with $i = j$ denote the same list.

  Example 2.1 *When the term* $(+\ 1\ 2\ 3)$ *has to be unified with* $(+\ \ <1 \dots>\ \ x\ \ <2 \dots>)$ *there are three possibilities:*

  1. $\{<1 \dots> \leftarrow [], x \leftarrow 1, <2 \dots> \leftarrow [2, 3]\}$
  2. $\{<1 \dots> \leftarrow [1], x \leftarrow 2, <2 \dots> \leftarrow [3]\}$
  3. $\{<1 \dots> \leftarrow [1, 2], x \leftarrow 3, <2 \dots> \leftarrow []\}$

  This type of pattern can be used e.g. for explicitly flattening (c.f. [5]) a function.

  Example 2.2 *Consider the rule*

  $$(+\ \ <1 \dots>\ \ (+\ \ <2 \dots>)\ \ <3 \dots>) \rightarrow (+ <1 \dots> <2 \dots> <3 \dots>)$$

  and the term $(+\ a\ b\ c\ (+\ d\ e)\ f)$. *The rule above is applicable with the substitution* $\{<1 \dots> \leftarrow [a, b, c], <2 \dots> \leftarrow [d, e], <3 \dots> \leftarrow [f]\}$. *This application results in the term* $(+\ a\ b\ c\ d\ e\ f)$.

- The pattern $< a_i \ldots e_i >$ is used instead of $< i \ldots >$ if the individual list elements are changed in a uniform manner. $< a_i \ldots e_i >$ is a syntactical keyword, too, where the reader can think of a_i as the first list element and of e_i as the last one. There is only one common list counter i for $<i \ldots>$ and $<a_i \ldots e_i>$, and constructs of different kinds must not share the same index.

  There is an alternative form $<t(a_i) \ldots t(e_i)>$ where $t(x)$ stands for any term containing the variable x. This form expresses a list where all elements are instances of the pattern $t(x)$. The pattern $<t(a_i) \ldots t(e_i)>$ matches every list $(v_1\ v_2 \ldots v_k)$ where each v_j is of the form $t(u_j)$, i.e.

  $$\forall j \in \{1, \ldots, k\}\ \exists u_j\,.\, v_j = \sigma_j(t(x)) \text{ with } \sigma_j = \{x \leftarrow u_j\}$$

  Please note the following correspondence between the patterns $<a_i \ldots e_i>$ and $<t(a_i) \ldots t(e_i)>$: If some substitution τ assigns the list $(v_1\ v_2 \ldots v_k)$ to $<t(a_i) \ldots t(e_i)>$ it implicitly contains the assignment of $(u_1\ u_2 \ldots u_k)$ to $<a_i \ldots e_i>$ as well as the assignment of $(v'_1\ v'_2 \ldots v'_k)$ $(v'_j := \sigma_j(t'(x)))$ to $<t'(a_i) \ldots t'(e_i)>$ and vice versa. This ensures that all patterns $<a_i \ldots e_i>$ and $<t'(a_i) \ldots t'(e_i)>$ occurring on the right hand side of a rule are well-defined whenever any of the patterns $<t(a_i) \ldots t(e_i)>$ or $<a_i \ldots e_i>$ occurs on the left hand side of the rule.

  Example 2.3 *Consider the rule*

  $$(*\ (+\ \ <a_1 \ldots e_1>)\ t) \rightarrow (+\ \ <(*\ a_1\ t) \ldots (*\ e_1\ t)>)$$

  that eliminates sums inside of products by a restricted form of distributivity: E.g. it is applicable on the input term $(*\ (+\ t_1\ t_2\ t_3)\ t')$ *with the substitution* $\{< a_1 \ldots e_1 > \leftarrow [t_1, t_2, t_3], t \leftarrow t'\}$ *and reduces it in one step to the term*

  $$(+\ (*\ t_1\ t')\ (*\ t_2\ t')\ (*\ t_3\ t'))$$

  As in the example, a rule usually contains some $< a_i \ldots e_i >$ together with the corresponding $<t(a_i) \ldots t(e_i)>$, one on each side of the rule.

2.2 Handling Commutativity

In the context of functions with variable number of arguments the different uses of commutativity can be divided as follows:

1. A rule r changes a fixed number of arguments of a commutative functor. Due to the commutativity there are several (but finitely many) rules that result from the permutations of the arguments changed by r.

Example 2.4 *For the rule*

$$(+\ {<}1\ldots{>}\ x\ {<}2\ldots{>}\ (-\ x)\ {<}3\ldots{>}) \rightarrow (+\ {<}1\ldots{>}{<}2\ldots{>}{<}3\ldots{>})$$

a second rule results from exchanging x and $(-\ x)$:

$$(+\ {<}1\ldots{>}\ (-\ x)\ {<}2\ldots{>}\ x\ {<}3\ldots{>}) \rightarrow (+\ {<}1\ldots{>}{<}2\ldots{>}{<}3\ldots{>})$$

2. All arguments with certain properties have to be chosen from the argument list of a commutative functor.

   Example 2.5 *Consider a rule that finds the common denominator of a sum of fractions. E.g. the following transformation should be possible:*

   $$\frac{1}{a \cdot b} + \frac{1}{c \cdot a} \rightarrow c \cdot \frac{1}{a \cdot b \cdot c} + b \cdot \frac{1}{a \cdot b \cdot c}$$

   Then it is necessary to extract all common factors of two denominators no matter in which order they appear.

In the following we do not consider the case (1) any more since it can be solved by additional rules without extending the rule syntax any further. To handle case (2) two new notations representing patterns that may occur on the left hand side of a rule are introduced:

- For a function symbol f we write ${<}comm\ f{>}$, if f represents a commutative function and the commutativity should be utilized for the current rule.
- Inside of the argument list of a commutative functor the pattern ${<}\,choose\ {<}\,xxx\,{>}{>}$ (where ${<}\,xxx\,{>}$ stands for ${<}\,i\ldots{>}$, ${<}\,a_j\ldots e_j\,{>}$ or ${<}\,t(a_j)\ldots t(e_j)\,{>}$) represents the choice of a maximal subset with certain restrictions.

  Example 2.6 *Consider the following rule generating a common denominator of two fractions:*

  $$\begin{aligned}(+\ (/\ ({<}comm\ *{>}\ {<}choose\ \ {<}1\ldots{>}{>}\ {<}2\ldots{>}))&\\ (/\ ({<}comm\ *{>}\ {<}choose\ \ {<}1\ldots{>}{>}\ {<}3\ldots{>})))&\\ \rightarrow (+\ (*\ \ {<}3\ldots{>}\ (/\ (*\ \ {<}1\ldots{>}\ {<}2\ldots{>}\ {<}3\ldots{>})))&\\ (*\ \ {<}2\ldots{>}\ (/\ (*\ \ {<}1\ldots{>}\ {<}2\ldots{>}\ {<}3\ldots{>}))))&\end{aligned}$$[1]

  By the two occurrences of ${<}1\ldots{>}$ in the two different choose patterns we express that only the common elements of these argument lists should be collected. The remaining elements of the lists are put into ${<}2\ldots{>}$ or ${<}3\ldots{>}$ respectively.

[1]There is a further construct necessary to guarantee termination of this rule. It is introduced in Subsec. 2.3.

The possible *choose conditions* on the elements unifiable with the choose list are the occurrences in several argument lists as in Ex. 2.6 and matching a certain pattern $t(x)$ in the case of $<\mathit{choose}\ \ <t(a_1)\dots t(e_1)>>$.

The use of the commutative choice as described above is restricted as follows:

- The first argument of an operator defined by $<\mathit{comm}\dots>$ must be a choose-list $<\mathit{choose}\ \ <xxx>>$.
- Besides exactly one choose list a commutative operator contains at most one more (maybe extended) construct. Without a second construct, unification with a term is only possible if all arguments fit into the choose list. If a second "absorption"-construct exists, it must be unifiable with the remaining arguments not caught by the choose list.
- If an argument list of variable arity appears in the position of the choose list inside a commutativity construct, every appearance of this list at the left hand side of the rule must take place at the choose list position inside a commutativity construct and all occurrences must be of the same form ($<i\dots>$, $<a_i\dots e_i>$ or $<t(a_i)\dots t(a_i)>$ with identical $t(x)$).
- A commutative operator cannot appear inside an extended construct from Sec. 2.1, i.e. with

  $$t(x) := (<\mathit{comm}\ *> <\mathit{choose}\ \ <2\dots>>\ \ x)$$

  the following left hand side of a rule is not valid:

  $$(+\ \ <1\dots> <t(a_3)\dots t(e_3)> <4\dots>) \rightarrow \dots$$

The use of the commutative choice is demonstrated by the following example:

Example 2.7 *Consider the rule in Ex. 2.6 and the term* $(+\ (/\ (*\ x\ z))\ (/\ (*\ y\ z)))$. *The rule is applicable with the substitution* $\sigma := \{<1\dots>\leftarrow [z], <2\dots>\leftarrow [x], <3\dots>\leftarrow [y]\}$. *The result of this rewriting step is the term*

$$(+\ (*\ y\ (/\ (*\ z\ x\ y)))\ (*\ x\ (/\ (*\ z\ x\ y))))$$

Calculating a matcher between a term and the left hand side of a rule containing commutative choice is possible as follows:

- For all lists that have to match with a commutative construct, the arguments are sorted according to a fixed total order on ground terms providing efficient tests for $<, >, =$.

- All lists, which define the same choose-list, are processed together. During a linear sweep, all common elements matching the given pattern, are collected.

Compared to ordinary term matching, the matching algorithm becomes more complex by this extension, because the terms to be unified can no longer be processed in a linear order, but all commutative constructs must be processed together. However, it should perform quite well compared to other matching algorithms that can handle associativity and commutativity since the use of associativity and commutativity with its computational overhead can be restricted to those rules where it is really needed.

This method of handling commutativity is an extension to *ordered rewriting* as described e.g. in [3]. The arguments of a binary commutative operator $\bullet$ can be ordered as a special case by the following rule (providing some extra work to obtain termination by introducing a new function symbol marking those appearances that have not been sorted yet):

$$(<\!comm\ \bullet\!>\ <\!choose\,<\!1\ldots\!>\!>) \ \rightarrow\ (\bullet\ \ <\!1\ldots\!>)$$

This rule processes the commutative operator $\bullet$ of variable arity and especially of arity 2 correctly in the sense that terms consisting of the operator $\bullet$ and the same set of arguments are transformed to terms with the arguments in the same order.

2.3 The Problem of Empty Rule Execution

The introduction of functions with variable arity causes a new problem to occur: the problem of empty application of a rule. It occurs when an argument pattern of a function with variable arity is instantiated with a list of certain length (usually 0 or 1) and the application of the rule cannot prevent itself from being applicable at the same position again.

Example 2.8 *Consider the rule*

$$(*\ \ <\!(/\ a_1)\ldots(/\ e_1)\!>\ <\!2\ldots\!>) \rightarrow (*\ (/\ (*\ \ <\!a_1\ldots e_1\!>))\ \ <\!2\ldots\!>)$$

that takes some denominators from the beginning of the factor list of a product and combines them to a single denominator.[2] *Now let us consider the term* $t := (*\ (/\ 2)\ x)$*. The rule is applicable with substitution* $\sigma := \{<(/\ a_1)\ldots(/\ e_1)> \leftarrow [(/\ 2)], <2\ldots> \leftarrow [x]\}$*. A single reduction step on* t *yields* $t' := (*\ (/\ (*\ 2))\ x)$*. The problem here is not the product containing just one factor, because that can be corrected easily by the rule*

$$(*\ x) \rightarrow x$$

[2]One would normally use the commutative choice to select *all* denominators, but this is suppressed here for simplification.

The problem rather is the fact that the rule is applicable again on t' with $\sigma' := \{<(/\ a_1)\ldots(/\ e_1)> \leftarrow [(/\ (*\ 2))], <2\ldots> \leftarrow [x]\}$ and that this kind of reduction could go on forever.

Applications of this kind are avoidable, if certain constructs of variable arity are not allowed to be instantiated with a list of certain length. For this purpose we introduce the pattern

$$<<[\mathit{notarity}\ (v_1\ n_1)\ldots(v_k\ n_k)]>>$$

where the v_i are list indices that must occur in the rule the construct is used for and the n_i are non-negative integers. Such a construct can be added to the end of the left hand side of a rule. It prevents the rule from being applied with a substitution that assigns to every list with list index $j \in \{v_1, \ldots, v_k\}$ (i.e. $<j\ldots>$, $<a_j \ldots e_j>$ or $<t(a_j)\ldots t(e_j)>$) an argument list of length n_i.

If one pattern of variable arity should block the rule execution for several arities these constraints have to be expressed with several instances of the *notarity* pattern.

Example 2.9 *To get termination for the rule in Ex. 2.8, one must modify it to make sure that at least two fractions are combined. Thus, we have to exclude the arities* 0 *and* 1 *for the list* $<1\ldots>$*:*

$$\begin{aligned}(*\ \ <(/\ a_1)\ldots(/\ e_1)>\ <2\ldots>)\ \ <<[\mathit{notarity}\ (1\ 0)]>>&\\ <<[\mathit{notarity}\ (1\ 1)]>> \rightarrow (*\ (/\ (*\ \ <a_1\ldots e_1>))\ \ <2\ldots>)&\end{aligned}$$

An implementation of this kind of restriction is easy, because it is sufficient to check a generated substitution after unifying the left hand side of the rule and the term. This check can be added to the end of the unification algorithm.

3 The Relation Between Normal and Extended Term Rewriting

In this section we describe the meaning of the different syntactic constructs defined before from a different point of view. It is a central observation in this context that most of the new constructs yield rules which represent sets of rules not using the syntactical extensions. Thus, it is possible to define some function φ, mapping every trs T employing the extended syntax to a (usually infinite) trs $T' := \varphi(T)$ consisting of rules in standard syntax.

The definition of φ is first motivated by a set of examples explaining for every extended construct how it can be represented by standard rules.

We will start with a simplification of addition as a typical application for $<i\ldots>$. The simplification is given by the rule

$$(+\ <1\ldots>\ x\ <2\ldots>\ (-\ x)\ <3\ldots>) \rightarrow (+\ <1\ldots><2\ldots><3\ldots>)$$

This rule represents the following infinite set of simple rules:

$$\begin{aligned}
(+\ x\ (-\ x)) &\rightarrow (+)^3 \\
(+\ u_1\ x\ (-\ x)) &\rightarrow (+\ u_1) \\
(+\ x\ u_1\ (-\ x)) &\rightarrow (+\ u_1) \\
(+\ x\ (-\ x)\ u_1) &\rightarrow (+\ u_1) \\
(+\ u_1\ u_2\ x\ (-\ x)) &\rightarrow (+\ u_1\ u_2) \\
(+\ u_1\ x\ u_2\ (-\ x)) &\rightarrow (+\ u_1\ u_2) \\
(+\ u_1\ x\ (-\ x)\ u_2) &\rightarrow (+\ u_1\ u_2) \\
(+\ x\ u_1\ u_2\ (-\ x)) &\rightarrow (+\ u_1\ u_2) \\
(+\ x\ u_1\ (-\ x)\ u_2) &\rightarrow (+\ u_1\ u_2) \\
(+\ x\ (-\ x)\ u_1\ u_2) &\rightarrow (+\ u_1\ u_2) \\
&\vdots
\end{aligned}$$

Lists like $<a_i\ldots e_i>$ are used e.g. for expressing distributivity. The following rule gives a simplified sketch of the principle:

$$(*\ x\ (+\ <a_1\ldots e_1>)) \rightarrow (+\ <(*\ x\ a_1)\ldots(*\ x\ e_1)>)$$

It represents the following infinite set of simple rules:

$$\begin{aligned}
(*\ x\ (+)) &\rightarrow (+) \\
(*\ x\ (+\ u_1)) &\rightarrow (+\ (*\ x\ u_1)) \\
(*\ x\ (+\ u_1\ u_2)) &\rightarrow (+\ (*\ x\ u_1)\ (*\ x\ u_2)) \\
(*\ x\ (+\ u_1\ u_2\ u_3)) &\rightarrow (+\ (*\ x\ u_1)\ (*\ x\ u_2)\ (*\ x\ u_3)) \\
&\vdots
\end{aligned}$$

As the next example, we show the combination of the product of several fractions to one fraction. We introduce the needed constructs step by step. We start with a rule, which uses a construct of the form $<t(a_1)\ldots t(e_1)>$, in this case $t(x) = (/\ x)$:

$$(*\ <(/\ a_1)\ldots(/\ e_1)><2\ldots>) \rightarrow (*\ (/\ (*\ <a_1\ldots e_1>))\ <2\ldots>)$$

[3] This kind of degenerated function application is typical for rules with argument lists of variable arity. They are easily resolvable by further rules, e.g. $(+) \rightarrow 0$

This rule represents the following set of standard rules:

$$\begin{array}{rll}
(*) & \rightarrow (*\ (/)) & \star \\
(*\ (/\ u_1)) & \rightarrow (*\ (/\ (*\ u_1))) & \star \\
(*\ v_1) & \rightarrow (*\ (/)\ v_1) & \star \\
(*\ (/\ u_1)\ (/\ u_2)) & \rightarrow (*\ (/\ (*\ u_1\ u_2))) & \\
(*\ (/\ u_1)\ v_1) & \rightarrow (*\ (/\ (*\ u_1))\ v_1) & \star \\
(*\ v_1\ v_2) & \rightarrow (*\ (/)\ v_1\ v_2) & \star \\
(*\ (/\ u_1)\ (/\ u_2)\ (/\ u_3)) & \rightarrow (*\ (/\ (*\ u_1\ u_2\ u_3))) & \\
(*\ (/\ u_1)\ (/\ u_2)\ v_1) & \rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1) & \\
(*\ (/\ u_1)\ v_1\ v_2) & \rightarrow (*\ (/\ (*\ u_1))\ v_1\ v_2) & \star \\
(*\ v_1\ v_2\ v_3) & \rightarrow (*\ (/)\ v_1\ v_2\ v_3) & \star \\
& \vdots &
\end{array}$$

This set contains rules working on none or exactly one fraction. In the rule set above, such rules are marked with $\star$. These rules provide no functionality, but violate the termination property of the trs. To solve this problem, we delete these rules by blocking the arities 0 and 1 for the list with list index 1:

$$(*\ \ {<}(/\ a_1)\ldots(/\ e_1){>}\ {<}2\ldots{>})\ \ {<<}[\mathit{notarity}\ (1\ 0)]{>>}$$
$$\qquad\qquad {<<}[\mathit{notarity}\ (1\ 1)]{>>} \rightarrow (*\ (/\ (*\ \ {<}a_1\ldots e_1{>}))\ \ {<}2\ldots{>})$$

After this change, the set of represented rules just contains those rules that work on at least two fractions:

$$(*\ (/\ u_1)\ (/\ u_2)) \rightarrow (*\ (/\ (*\ u_1\ u_2))) \tag{1}$$
$$(*\ (/\ u_1)\ (/\ u_2)\ (/\ u_3)) \rightarrow (*\ (/\ (*\ u_1\ u_2\ u_3))) \tag{2}$$
$$(*\ (/\ u_1)\ (/\ u_2)\ v_1) \rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1) \tag{3}$$

$$(*\ (/\ u_1)\ (/\ u_2)\ (/\ u_3)\ (/\ u_4)) \rightarrow (*\ (/\ (*\ u_1\ u_2\ u_3\ u_4))) \tag{4}$$
$$(*\ (/\ u_1)\ (/\ u_2)\ (/\ u_3)\ v_1) \rightarrow (*\ (/\ (*\ u_1\ u_2\ u_3))\ v_1) \tag{5}$$
$$(*\ (/\ u_1)\ (/\ u_2)\ v_1\ v_2) \rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1\ v_2) \tag{6}$$
$$\vdots$$

Now there is one more problem to cope with: Fractions occurring behind at least one non-fraction factor are not processed by this rule. This problem can be handled by exploiting the commutativity of $*$:

$$({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}(/\ a_1)\ldots(/\ e_1){>>}\ {<}2\ldots{>})$$
$$\qquad {<<}[\mathit{notarity}\ (1\ 0)]{>>}\ \ {<<}[\mathit{notarity}\ (1\ 1)]{>>}$$
$$\qquad\qquad \rightarrow (*\ (/\ (*\ \ {<}a_1\ldots e_1{>}))\ \ {<}2\ldots{>})$$

By this change, the set of represented rules is extended and now contains every permutation of arguments. For instance, for the two rules (1) and (2) in the previous rule set we get now the extended set:

$$
\begin{aligned}
(*\ (/\ u_1)\ (/\ u_2)) &\rightarrow (*\ (/\ (*\ u_1\ u_2))) \\
(*\ (/\ u_2)\ (/\ u_1)) &\rightarrow (*\ (/\ (*\ u_1\ u_2))) \\
(*\ (/\ u_1)\ (/\ u_2)\ v_1) &\rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1) \\
(*\ (/\ u_2)\ (/\ u_1)\ v_1) &\rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1) \\
(*\ (/\ u_1)\ v_1\ (/\ u_2)) &\rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1) \\
(*\ (/\ u_2)\ v_1\ (/\ u_1)) &\rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1) \\
(*\ v_1\ (/\ u_1)\ (/\ u_2)) &\rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1) \\
(*\ v_1\ (/\ u_2)\ (/\ u_1)) &\rightarrow (*\ (/\ (*\ u_1\ u_2))\ v_1)
\end{aligned}
$$

The other rules of the previous set are extended by their permutations in the same manner, yielding a complete solution for the problem of combining several denominators.

The applicability of every rule is implicitly restricted by the following conditions:

- Non of the v_i fits the pattern $(/\ t)$. Those terms are reserved for the unification with the $(/\ u_i)$.
- The rule only applies if the terms bound to u_i or v_i, respectively, are ordered according to an arbitrary fixed order on the set of terms. (This is not typical for term rewriting systems, since the applicability of a rule depends on the right hand side of it, but the restriction only exists in the explanation given for the example, not in the implementation by sorting the argument terms.)

Now we can define φ as follows:

Definition 3.1 *For a given extended rule R the set $\varphi(R)$ is defined by the following algorithm:*

1. *Every extended list $< i \ldots >$, $< a_i \ldots e_i >$ or $< t(a_i) \ldots t(e_i) >$ in a rule R^i is (one by one) eliminated by generating an infinite set of rules R^i_k containing $k \in \mathbb{N}$ variables instead of the extended list i.*
2. *Delete those rules from the list generated in (1) that violate one of the* notarity *constraints.*
3. *If R contains constructs for commutative choice these are transformed construct by construct and rule by rule as follows: For a given rule R^i and a certain commutative operator f generate rules containing all permutations of the argument list of f.*

For the definition of φ we assume the existence of a fair enumeration of the generated rules.

The function φ can be extended to a function on rule sets by

$$\varphi(T) := \{S \mid R \in T \land S = \varphi(R)\}\,.$$

For the function φ the following lemma hold:

Lemma 3.2 *Let T be a trs containing the extended syntax and let $T' := \varphi(T)$ be the standard trs represented by T. If $\rightarrow_T$ and $\rightarrow_{T'}$ are the one step reduction relations of T and T', respectively, we have*

$$t \rightarrow_T t' \Longleftrightarrow t \rightarrow_{T'} t'\,.$$

Proof: The lemma follows directly from the fact that the transformation algorithm from Def. 3.1 does not change the set of possible reductions. □

Lemma 3.2 directly carries over to the transitive closures $\rightarrow^+$ and reflexive transitive closures $\rightarrow^*$

4 Effect of the Syntactical Extensions on Confluence and Termination

Termination and confluence are the two main properties of term rewriting systems. This section describes how the checks for termination and confluence can be performed for term rewriting systems using the extensions described before.

4.1 Termination

The following corollary from Lemma 3.2 states that for checking some extended trs T for termination it is sufficient to check $\varphi(T)$ instead.

Corollary 4.1 *Let T be an extended trs. T is terminating iff $\varphi(T)$ is terminating.*

Proof: By Lemma 3.2 every reduction sequence that is possible in one of the trs is also possible in the other. Therefore, if there is an infinite sequence in T this sequence can also be found in $\varphi(T)$ and vice versa. Thus, non-termination of T and $\varphi(T)$ are equivalent. Hence termination is also equivalent for T and $\varphi(T)$. □

$T' := \varphi(T)$ is usually infinite for a given extended trs T. The following lemma states that it suffices to check a finite subset T'' of T' for termination.

Lemma 4.2 *For a given extended rule R let m be the number of different variables and n the number of different extended lists in R. Let q be the maximal number of occurrences of the same variable or extended list on the left hand side of R. For every extended list with list index j let g_j be the maximal arity for list j which can block the application of R ($g_j = 0$ if no* notarity *constraint on list j is given) and let $min_j := max\{2, g_j\}$.*

If all rules in the rule set $T'_R \subset T_R := \varphi(R)$ with maximal functor arity $S_1 = q \cdot (m + \sum_{k=1}^{n} min_k)$ are proved terminating by a termination order $\preceq$ then $\preceq$ is also a termination order for T_R.

Proof: The definition of S_1 guarantees that an arity of S_1 is sufficient to assign a value to every variable and more than the maximal blocking arity to every extended list (or 2 values if no blocking constraint to the list is given). Enlarging the arity any further just brings more elements into an extended list. But since these elements are transformed according to a common pattern, this cannot violate the termination property any more. $\square$

One should note that the standard rules represented by a rule containing the commutative choice need not to be checked for assignments for which they are not applicable due to order restrictions. If we restrict ourselves to termination orders not depending on the order of the arguments this does not cause any problems. Since termination orders depending on the argument order do not seem to make much sense in the context of commutativity this is not a heavy restriction.

4.2 Local Confluence

As before in the case of termination we first prove that (local) confluence of an extended trs T is equivalent to (local) confluence of $\varphi(T)$.

Corollary 4.3 *Let T be an extended trs. Then T is (locally) confluent iff $\varphi(T)$ is (locally) confluent.*

Proof: By Lemma 3.2 we have for any terms s and s':

$$\begin{aligned} s \rightarrow_T s' &\iff s \rightarrow_{\varphi(T)} s' \\ s \rightarrow^*_T s' &\iff s \rightarrow^*_{\varphi(T)} s' \end{aligned}$$

Taking several instances of these two properties we get:

$$((t \to_T t_1) \wedge (t \to_T t_2)) \Rightarrow (\exists t' : (t_1 \to_T^* t') \wedge (t_2 \to_T^* t')) \iff$$
$$((t \to_{\varphi(T)} t_1) \wedge (t \to_{\varphi(T)} t_2)) \Rightarrow (\exists t' : (t_1 \to_{\varphi(T)}^* t') \wedge (t_2 \to_{\varphi(T)}^* t'))$$

and

$$((t \to_T^* t_1) \wedge (t \to_T^* t_2)) \Rightarrow (\exists t' : (t_1 \to_T^* t') \wedge (t_2 \to_T^* t')) \iff$$
$$((t \to_{\varphi(T)}^* t_1) \wedge (t \to_{\varphi(T)}^* t_2)) \Rightarrow (\exists t' : (t_1 \to_{\varphi(T)}^* t') \wedge (t_2 \to_{\varphi(T)}^* t'))$$

□

As for the termination test, for the test of local confluence we can also restrict ourselves to a finite subset of $\varphi(T)$.

Lemma 4.4 *Let T be an extended trs with rules R_1 and R_2, m_1, m_2 the number of different variables in R_1 and R_2, respectively, n_1, n_2 the number of extended lists and q_1 and q_2 the maximal number of occurrences of a variable or extended list on the left hand side of the rule R_i, respectively. For every extended list with list index j in rule i let $g_{i,j}$ be the maximal arity for which list j can block R_i. ($g_{i,j} = 0$ if R_i does not contain a* notarity *constraint on list j.) $min_{i,j} := max\{2, g_{i,j}\}$.*

If some rules in $T' := \varphi(T)$ represented by R_1 and R_2 yield a critical pair violating local confluence then there exist rules represented by R_1 and R_2 with this property in the finite subset $T'' \subset T'$ just containing functors of arity $\leq S_2 := q_1 \cdot q_2 \cdot (m_1 + \sum_{k=1}^{n_1} min_{1,k} + m_2 + \sum_{k=1}^{n_2} min_{2,k})$.

The idea in this lemma is similar to that in Lemma 4.2. The definition of S_2 is more complicated than that of S_1 above. An element can play a certain role in both of the rules (a role means being assigned to a certain variable or a certain extended list) and we must make sure that for any given pair of roles in R_1, R_2 there is an argument that can play those rules.

Proof: What we need is a maximal arity that fulfills the following properties:

1. The arity chosen is large enough to exceed the maximal arity of the list number i that blocks R_2 for every i at the same time where every extended list can get at least 2 elements.

2. There are enough further elements to exceed the blocking arity of every extended construct of R_1 at the same time as well. These further elements can be spread over the extended constructs of R_2 arbitrarily. Especially, they can be all assigned to the same construct.

According to its definition S_2 meets these conditions:

1. The subterm m_2 covers one occurrence of every variable in R_2 and the sum $\sum_{k=1}^{p_2} min_{2_k}$ covers one occurrence of every extended construct. The factor q_2 makes sure that even multiple occurrences of the same variable or construct can be saturated.
2. Analogously the term $q_1 \cdot (m_1 + \sum_{k=1}^{p_1} min_{1_k})$ saturates all variables and extended constructs of R_1. The additional factor q_2 assures that this minimal number of arguments to R_1 can be assigned to any of the extended constructs of R_2, no matter how often this construct occurs.

By the symmetric definition of S_2 this also holds after exchanging R_1 and R_2.

If we now increase the maximal functor arity any more we just add further elements that are processed in the same manner as at least 2 already existing elements. Therefore they cannot violate the property of local confluence if it was not violated before. □

Note that we chose the maximal function arity in T'' only depending on the left hand sides of the rules. Since the functor arity of a term can increase during normalization we have to use the whole trs $T' = \varphi(T)$ for normalizing critical pairs.

This normalization can easily be done since it is always sufficient to use only a finite subset of rules from T'. The arity needed to normalize a given term can be extracted from that term, and for every arity there are only finitely many rules which can be generated on the fly.

A special handling of the result is necessary, if R_1 and R_2 are represented by the same extended rule containing the commutative choice. In that case we usually get normal forms that are not (syntactically) equivalent, but equivalent modulo permutation of variables that represent elements of the *same* extended list (choice list or absorption list). This does not cause a violation of the local confluence.

Example 4.5 *Consider the extended rule*

$$\begin{array}{r}(< comm * > < choose\ \ < (/\ a_1) \ldots (/\ e_1) >> <2 \ldots>) \\ \to (*\ (/\ (*\ \ < a_1 \ldots e_1 >))\ \ <2 \ldots>)\end{array}$$

It represents among others the following rules:

$$\begin{array}{ll}(*\ (/\ x_1)\ y_1\ (/\ x_2)) & \to (*\ (/\ (*\ x_1\ x_2))\ y_1) \\ (*\ (/\ x_2)\ y_1\ (/\ x_1)) & \to (*\ (/\ (*\ x_1\ x_2))\ y_1)\,.\end{array}$$

These two rules yield the critical pair (after renaming of variables)

$$(*\ (/\ (*\ u_1\ u_2))\ v) = (*\ (/\ (*\ u_2\ u_1))\ v)\,.$$

Both terms are already in normal form. They are not (syntactically) equal, but equal modulo permutation of the u_i corresponding to the choose list.

The statements of this section are summarized in the following theorem:

Theorem 4.6 *To check an extended trs T for termination and local confluence it is sufficient to check a finite subset of $T' = \varphi(T)$.*

4.3 Relation to Knuth-Bendix Completion.

It is important to note that the techniques shown above are usable for checking the local confluence of a term rewriting system T containing the extended constructs introduced in this paper. On the other hand it seems quite hard to find a completion procedure in the sense of the Knuth-Bendix algorithm [6] that works directly on T (rather than on $\varphi(T)$) because one would have to find common patterns in new rules that form a new extended rule. Thus, it is an open question, whether the original completion algorithm can be extended to handle term rewriting systems using the constructs presented in this paper directly.

5 Conclusion and Future Work

In this paper we have addressed the idea of using term rewriting systems in a programming related style. By the extension of the rule syntax presented here we have overcome the problem that associativity and commutativity are usually handled by separating equations for these theories from the rule set. Our approach has the benefit of allowing to express the use of associativity and commutativity in the rules explicitly.

A prototype of a term rewriting interpreter for these extended constructs has been implemented in the LISP dialect Scheme from which we adopted the syntax of functions with variable arity. The implementation details can be found in [2]. The usefulness of the new syntax has been proven in applications of performing local normalizations on Scheme programs [9] and the normalization of type terms for a type inference system [8].

From the theoretical point of view we have shown that termination and confluence of trs employing the new extensions can be checked by reducing it to finite trs in standard syntax. From experiences with checking these properties on several trs we expect that it may be possible to perform these checks directly on a subset of the extended syntax. Discovering this as well as tools for extending the Knuth-Bendix completion algorithm [6] are left for future work.

References

[1] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 245–320. Elsevier Science Publishers B.V., 1990.

[2] Dieter Froning. Lokale Normalisierung von Scheme-Programmen. Diplomarbeit, FernUniversität Hagen, April 1998.

[3] J. P. Jouannaud. Rewrite proofs and computations. International Summer School Marktoberdorf "Logic of computation", Institut für Informatik, Technische Universität München, 1995.

[4] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4):1155–1194, November 1986.

[5] Deepak Kapur and G. Sivakumar. A total, ground path ordering for proving termination of ac-rewrite systems. In Hubert Comon, editor, *Proceedings of the 8th International Conference on Rewriting Techniques and Applications (RTA-97), LNCS 1232*, pages 142–156, 1997.

[6] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297, 1967.

[7] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, April 1981.

[8] Manfred Widera and Christoph Beierle. A complete type system for functional languages. Informatik Berichte 240, FernUniversität Hagen, September 1998.

[9] Manfred Widera and Christoph Beierle. Local normalization of functional programs. Informatik Berichte 248, FernUniversität Hagen, January 1999.