Local normalization of functional programs

Manfred Widera, Christoph Beierle
Praktische Informatik VIII
FernUniversität Hagen
58084 Hagen
email: {Manfred.Widera, Christoph.Beierle}@FernUni-Hagen.de

Abstract

This article presents methods for local normalization of functional programs written in LISP. The intension is a preprocessing step to reduce the set of used language constructs. This helps to reduce the complexity of subsequent program analysis. The main spots are list expressions, arithmetic expressions, especially with operators $+$, $-$, $*$, $/$, and boolean expressions. To reach this goal an extension to term rewriting systems is presented that can cope with operators of arbitrary arity as they are used in LISP.

Contents

1 Introduction

An important problem of program analysis is caused by the number of different ways to express the same program semantics. Examples are given by transformations of arithmetic expressions, yielding equivalent but syntactically different expressions, circumscriptions of predicates (e.g. $\neg(a > b) \wedge \neg(a = b)$ for $a < b$) and abbreviations of nested list accesses (e.g. $(caddr\ l)$ for $(car\ (cdr\ (cdr\ l)))$ in the functional language Scheme).
For syntax oriented program analysis all these different possibilities cause more complexity and larger case distinctions. Normalization of programs can reduce the expense of program analysis by transforming the program to a form which allows the analysis system to rely on stronger conditions on the program code.
Furthermore, normalizing programs or parts of programs can be useful, if some analyzing procedure has to check different terms for equivalence. By a powerful normalizing system this problem can hopefully be reduced to the problem of syntactic equivalence.

Our actual goal is the normalization of programs written in LISP, where examples are given in the LISP-dialect Scheme. As a tool for program transformation we use the formalism of term rewriting systems, which let us give a short and easily understandable description of the transformation steps.

The paper is organized as follows: Section 2 explains some novel extensions to the concept of term rewriting systems, which we introduce in order to handle the special properties of LISP. Term rewriting systems for normalizing list terms, arithmetic terms and boolean terms are given in sections 3, 4 and 5 respectively. Section 6 shows, how types of terms can change during normalization and how complex terms (terms with subterms of different type) can be normalized under the possibility of type changes. A short summary and conclusions are given in section 7.

2 Extension of term rewriting

Term rewriting systems are very important in analyzing equivalence of terms. Ideally, they provide a simple formalism to reduce semantic equivalence to syntactic equality of the normal forms.
Normalization by term rewriting systems is based on unification (resp. on matching) of a rule and the term to normalize. Main properties of term rewriting systems (trs) are termination which guarantees that the trs provides a decision procedure and confluence which is important for the completeness of the given decision procedure. The basics on term rewriting systems can be found e.g. in [DJ90].
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.

Since we want to use term rewriting systems to normalize LISP programs efficiently, we extend the standard formalism in various ways. This section provides these extensions and discusses their influence on unification and the methods for proving termination and confluence.

2.1 Syntactic extensions

The extensions described in this subsection provide solutions to the following problems of trs in the context of LISP programs:

- Handling functions of variable arity.
- Commutativity in functions of variable arity.
- Appropriate restrictions of the applicability of rules.[1]

2.1.1 Functors of variable arity

Some function symbols in LISP denote a whole class of functions which just differ in their arity. To cope with this specialty of notation, rewrite rules for these classes of functions are introduced. The rules change a fixed number of arguments or argument lists only. The remaining arguments are not affected.

Notations:

- For $i \in \{1, 2, \ldots\}$ the pattern $< i \ldots >$ represents an argument list of variable arity. A pattern $< i \ldots >$ 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 $< i \ldots >< j \ldots >$. Note that the whole expression $< i \ldots >$ 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 $<i \ldots>$ and $<j \ldots>$ with $i = j$ denote the same list.

  Example:

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

[1] Principles of restricting the applicability of rules are known as conditional rewriting as described in [DJ90]. These approaches have to be adopted to the intended applications.

transforms nested sums (e.g. $(+\ a\ b\ c\ (+\ d\ e)\ f))$ to a single sum containing all elements of the previous sums in the same order (e.g. $(+\ a\ b\ c\ d\ e\ f))$. The one step reduction of $(+\ a\ b\ c\ (+\ d\ e)\ f)$ in the given example is achieved by the pattern matching $\{<1\ldots>\leftarrow [a,b,c], <2\ldots>\leftarrow [d,e], <3\ldots>\leftarrow [f]\}$.

- 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 a 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 $< t(a_i)\ldots t(e_i) >$ and $<a_i \ldots e_i>$: 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: The rule
  $$(*\ (+\ \ <a_1 \ldots e_1>)\ t) \to (+\ \ <(*\ a_1\ t)\ldots(*\ e_1\ t)>)$$
  eliminates sums inside of products by distributivity: E.g. it transforms the input term $(*\ (+\ t_1\ t_2\ t_3)\ t')$ in one step to $(+\ (*\ t_1\ t')\ (*\ t_2\ t')\ (*\ t_3\ t'))$ with the substitution $\{<a_1 \ldots e_1>\leftarrow [t_1,t_2,t_3], t \leftarrow t'\}$.
  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.

2.1.2 Handling commutativity

In term rewriting systems with fixed arity for every functor, commutativity can often be circumscribed by specifying several rules describing the same transformation which differ only in application of the commutativity. E.g. addition of inverse numbers is expressible with two rules:

$$x + (-x) \to 0$$
$$(-x) + x \to 0$$

If operators of variable arity are possible, commutativity can cause the following situation: From a *list* of arguments we have to choose a maximal sub*set* of arguments with special properties (e.g. common elements of several argument lists). An example is given by the (simplified) rule for calculating the common denominator of two fractions:

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

As a first approach, one could try the following rule:

$$\begin{array}{l}
(+\ (/\ (*\ \ <1\ldots>\ <2\ldots>))\ (/\ (*\ \ <1\ldots>\ <3\ldots>)))\to \\
\qquad\qquad\qquad (+\ (*\ \ <3\ldots>\ \ (/\ (*\ \ <1\ldots>\ <2\ldots>\ <3\ldots>))) \\
\qquad\qquad\qquad\qquad\qquad\qquad (*\ \ <2\ldots>\ \ (/\ (*\ \ <1\ldots>\ <2\ldots>\ <3\ldots>))))
\end{array}$$

Unfortunately it only applies in a special case: All common factors have to occur in the same order at the beginning of both denominators. Since the number of common factors, which can occur in an arbitrary order at arbitrary positions, is not restricted, a finite number of rules cannot handle all cases.

Two new notations representing patterns that may occur on the left hand side of the rule are introduced to handle this case:

- For a function symbol f we write $<\mathit{comm}\ f>$, if f represents a commutative function and the commutativity should be used.
- Inside of the argument list of a commutative functor the pattern $<\mathit{choose}\ <\mathit{xxx}>>$ where $<\mathit{xxx}>$ may stand for $<i\ldots>$, $<a_j\ldots e_j>$ or $<t(a_j)\ldots t(e_j)>$ represents the choice of a maximal subset, which may be restricted to common arguments of several lists or to arguments fitting a certain pattern $t(x)$.

Using the new notations, the example can be handled by the following rule:

$$\begin{array}{l}
(+\ (/\ (<\mathit{comm}\ *>\ <\mathit{choose}\ \ <1\ldots>>\ <2\ldots>)) \\
\qquad\qquad\qquad\quad (/\ (<\mathit{comm}\ *>\ <\mathit{choose}\ \ <1\ldots>>\ <3\ldots>))) \\
\qquad\qquad\qquad \to (+\ (*\ \ <3\ldots>\ (/\ (*\ \ <1\ldots>\ <2\ldots>\ <3\ldots>))) \\
\qquad\qquad\qquad\qquad\qquad\qquad (*\ \ <2\ldots>\ (/\ (*\ \ <1\ldots>\ <2\ldots>\ <3\ldots>))))
\end{array}$$ [2]

This rule e.g. performs a one step reduction on the term $(+\ (/\ (*\ x\ z))\ (/\ (*\ y\ z))$ with the substitution $\sigma := \{<1\ldots> \leftarrow [z], <2\ldots> \leftarrow [x], <3\ldots> \leftarrow [y]\}$ and the result

[2] There is a further construct necessary to guarantee termination of this rule. It is introduced in subsection 2.1.3.

$(+\ (*\ y\ (/\ (*\ z\ x\ y)))\ (*\ x\ (/\ (*\ z\ x\ y))))$, i.e.

$$\frac{1}{x \cdot z} + \frac{1}{y \cdot z} \rightarrow y \cdot \frac{1}{z \cdot x \cdot y} + x \cdot \frac{1}{z \cdot x \cdot y}$$

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

- A commutative operator cannot appear inside an extended construct from section 2.1.1, i.e. with

  $$t(x) := (< comm * > < choose\ \ <2 \ldots >>\ \ x)$$

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

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

- The first argument of an operator defined by $< comm \ldots >$ must be a choose-list $< choose\ \ < xxx >>$.

- 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.

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

- For all lists that have to match with a commutative construct, the arguments are sorted according to a term ordering 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.

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.

This method of handling commutativity is an extension to *ordered rewriting* as described e.g. in [Jou95]. 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):

$$(<\mathit{comm}\ \bullet> \ <\mathit{choose}\ <1\ldots>>) \ \rightarrow \ (\bullet \ \ <1\ldots>)$$

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.

2.1.3 Syntactic restrictions of rule-applicability

Originally, term rewriting systems are based on equations with all occurring variables being all-quantified. I.e. a rule is applicable, whenever the left hand side matches a subterm of the term to be transformed.

Since even in simple cases exceptions from the all-quantification are necessary (e.g. division by 0), term rewriting systems were extended by conditions on the applicability of rules (conditional rewriting). Unfortunately there are often heavy restrictions on the type of possible conditions. Normally, equality of terms, which is checkable by the term rewriting system itself, is admitted [DJ90].
Dershowitz and Okada [DO90] handle tests by "built-in predicates", as well. Such tests are used in this subsection to define some special conditions, which seem to be useful in the context of program transformation.

The following conditions are useful in normalizing LISP programs:

1. Negative patterns can be given to certain variables. Substitutions to such a restricted variable x are only valid, if the term t assigned to x does not match the negative pattern p.
2. From argument lists of variable arity from subsection 2.1.1 certain list lengths (especially 0) are excluded. I.e. the rule only applies if the restricted argument list is not unified with some list of excluded length.

To express these conditions, the following notations are introduced:

If x is a variable and no unifier should assign a term t matching the negative pattern p to x, we write

$$<<[\mathit{exclude}\ p\ x]>>$$

As an example consider the following unrestricted rule to generate a common denominator of several fractions:

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

which corresponds to

$$\frac{1}{x} + \frac{1}{y} \rightarrow \frac{y}{x * y} + \frac{x}{x * y}$$

in standard notation.
Among other transformations this rule performs the following unintended transformation:

$$(+\ (/\ 3)\ (/\ 3) \rightarrow (+\ (*\ 3\ (/\ (*\ 3\ 3)))\ (*\ 3\ (/\ (*\ 3\ 3))))$$

or in infix notation

$$\frac{1}{3} + \frac{1}{3} \rightarrow \frac{3}{3 * 3} + \frac{3}{3 * 3}$$

By using a negative pattern, the rule becomes more restrictive:

$$(+\ (/\ x)\ (/\ y))\ \ {<<}[\mathit{exclude}\ x\ y]{>>} \rightarrow (+\ (*\ y\ (/\ (*\ x\ y)))\ (*\ x\ (/\ (*\ x\ y))))$$

This rule processes only terms containing at least two *different* denominators, i.e. terms where a transformation is actually needed.

Restrictions according to 2. handle the following problem: After introducing argument lists of variable length, one must make sure in certain cases, that the instances assigned to some of these lists are not empty or of a certain length. This is necessary to block the "empty application" of a rule, which may violate termination. An example is the following rule for combining products of fractions which does not terminate:

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

This rule is applicable e.g. to the term $t := (*\ (/\ 2)\ x)$ with the 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

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

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 goes on forever.

Applications of this kind are avoidable, if certain argument lists of variable length are not allowed to be instantiated with a list of certain length.

For this purpose we introduce the pattern

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

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 $<j\ldots>$, $<a_j\ldots e_j>$ or $<t(a_j)\ldots t(e_j)>$ with $j = v_i$ 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.

To get termination for our example rule above, one must modify it to make sure that at least two fractions are combined:

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

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.

2.1.4 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

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

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:

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

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:

$$\begin{array}{l}
(* \;\; <\!(/ \; a_1) \ldots (/ \; e_1)\!> \; <\!2 \ldots\!>) \;\; <\!<\![\mathit{notarity}\ (1\ 0)]\!>\!> \\
\qquad\qquad\qquad <\!<\![\mathit{notarity}\ (1\ 1)]\!>\!> \to (* \; (/ \; (* \;\; <\!a_1 \ldots e_1\!>)) \;\; <\!2 \ldots\!>)
\end{array}$$

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

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

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

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 $*$:

$$\begin{array}{l}
(<\!\mathit{comm}\ *\!> \; <\!\mathit{choose}\;\; <\!(/ \; a_1) \ldots (/ \; e_1)\!>\!> \; <\!2 \ldots\!>) \\
\qquad\qquad <\!<\![\mathit{notarity}\ (1\ 0)]\!>\!> \;\; <\!<\![\mathit{notarity}\ (1\ 1)]\!>\!> \\
\qquad\qquad\qquad\qquad \to (* \; (/ \; (* \;\; <\!a_1 \ldots e_1\!>)) \;\; <\!2 \ldots\!>)
\end{array}$$

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 2.1 *For a rule R we define $\varphi(R)$ as follows:*

- $R_0 := \{R\}$
- *For $j \in \mathbb{N}$ define $R_{j+1} := R_j \setminus \{R'\} \cup \{R_k\}_k$ where R' contains an extended construct with index i (i.e. $< i \ldots >$, $< a_i \ldots e_i >$ or $< t(a_i) \ldots t(e_i) >$) and $\{R_k\}_k$ is generated from R' by the following algorithm:*
    1. *For every $l \in \mathbb{N}$ generate a rule R_l that contains l variables (or terms) instead of the extended construct with index i.*
    2. *Delete those rules from the set generated in (1) that violate one of the* notarity *constraints in R'.*

3. *If the extended construct with index i occurs as* choose-*construct of a commutative choice in R' then for every rule $\tilde{R}$ in the set generated in (2) all those rules are generated which result from a permutation of the variables introduced for the extended construct with index i.*

- $\varphi(R) = \lim_{i\to\infty} R_i$.[4]

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

$$\varphi(T) := \bigcup_{R\in T} \varphi(R)\,.$$

For the function φ the following lemma holds:

Lemma 2.2 *Let T be a trs containing the extended syntax and let $T' := \varphi(T)$ be the standard trs represented by T. If $\to_T$ and $\to_{T'}$ are the one step reduction relations of T and T', respectively, we have*

$$t \to_T t' \Longleftrightarrow t \to_{T'} t'\,.$$

Proof: The lemma follows directly from the fact that the transformation algorithm from Def. 2.1 does not change the set of possible reductions. □

Lemma 2.2 directly carries over to the transitive closure $\to^+$ and the reflexive transitive closure $\to^*$.

In a further example we want to describe the *exclude* pattern by one rule that transforms a fraction into a fraction whose denominator contains a product, provided the original denominator was not already a product:

$$(/\ x)\ \ll [\mathit{exclude}\ (*\ <1\ldots>)\ x] \gg\ \to (/\ (*\ x))$$

The condition $\ll [\mathit{exclude}\ (*\ <1\ldots>)\ x] \gg$ blocks the rule application if a product term is bound to x. If this is not the case the rule is applied as if the condition was not present.

The *exclude* pattern has no effect on the number of standard rules represented by an extended one. Therefore we cannot eliminate its occurrences by a modification of φ but we have to carry over the definition of *exclude* to the standard rules.

[4] The limes is necessary because every step $i \to i+1$ introduces infinitely many new rules that may contain further extended constructs.

2.1.5 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.

Termination: The following corollary from Lemma 2.2 states that for checking some extended trs T for termination it is sufficient to check $\varphi(T)$ instead.

Corollary 2.3 *Let T be an extended trs. T is terminating iff $\varphi(T)$ is terminating.*

Proof: By Lemma 2.2 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 2.4 *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. □

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.

Remark 2.5 *The last remaining problem is the proof of termination of a standard trs containing* exclude *patterns. The following two observations often yield termination proofs for trs containing this pattern:*

1. *The* exclude *pattern cannot violate termination, i.e. if we can prove termination of a trs $\tilde{T}$ which results from T by deleting every* exclude *condition then T also terminates. This is true because this kind of constraint causes the constrained rule to be applicable less often. Furthermore, testing the constraint consist of simply calculating a unifier which always terminates.*

2. *Those rules for which termination cannot be proved otherwise often introduce new function applications for parameters and use* exclude *to block the rule application on inappropriate parameters. Since the number of appropriate parameters decreases by at least one with every rule application, this can be used for a termination proof.*

Example 2.6 *Consider the following two rules:*

$$\begin{array}{rl} (+\ \ <1\ldots>\ \ (/\ y)\ \ <2\ldots>) & \to (+\ \ <1\ldots>\ \ (*\ (/\ y))\ \ <2\ldots>) \\ (/\ x)\quad <<[exclude\ (*\ \ <1\ldots>)\ x]>> & \to (/\ (*\ x)) \end{array}$$

The first rule assures that no denominator occurs in a sum but all denominators in sums occur in (one factor) products. By the second rule every denominator consists of a (one factor) product. Here termination can be proved by the observation that every rule application decreases the number of terms not meeting the conditions above by one.

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

Proof: By Lemma 2.2 we have for any terms s and s':

$$\begin{array}{rl} s \to_T s' & \Longleftrightarrow s \to_{\varphi(T)} s' \\ s \to_T^* s' & \Longleftrightarrow s \to_{\varphi(T)}^* s' \end{array}$$

Taking several instances of these two properties we get:

$$\begin{array}{l} ((t \to_T t_1) \wedge (t \to_T t_2)) \Rightarrow (\exists t' : (t_1 \to_T^* t') \wedge (t_2 \to_T^* t')) \Longleftrightarrow \\ \qquad\qquad ((t \to_{\varphi(T)} t_1) \wedge (t \to_{\varphi(T)} t_2)) \Rightarrow (\exists t' : (t_1 \to_{\varphi(T)}^* t') \wedge (t_2 \to_{\varphi(T)}^* t')) \end{array}$$

and

$$((t \rightarrow_T^* t_1) \wedge (t \rightarrow_T^* t_2)) \Rightarrow (\exists t' : (t_1 \rightarrow_T^* t') \wedge (t_2 \rightarrow_T^* t')) \Longleftrightarrow$$
$$((t \rightarrow_{\varphi(T)}^* t_1) \wedge (t \rightarrow_{\varphi(T)}^* t_2)) \Rightarrow (\exists t' : (t_1 \rightarrow_{\varphi(T)}^* t') \wedge (t_2 \rightarrow_{\varphi(T)}^* t'))$$

□

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

Lemma 2.8 *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 2.4. 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 2.9 *Consider the extended rule*

$$\begin{aligned}(< comm * > < choose\ < (/\ a_1) \ldots (/\ e_1) >> <2 \ldots>)&\\ \rightarrow (* \ (/\ (* \ < a_1 \ldots e_1 >)) \ <2 \ldots>)&\end{aligned}$$

It represents among others the following rules:

$$\begin{aligned}(* \ (/\ x_1) \ y_1 \ (/\ x_2)) &\ \rightarrow (* \ (/\ (* \ x_1 \ x_2)) \ y_1)\\ (* \ (/\ x_2) \ y_1 \ (/\ x_1)) &\ \rightarrow (* \ (/\ (* \ x_1 \ x_2)) \ y_1)\,.\end{aligned}$$

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.

Remark 2.10 *Negative patterns for the unification given by $<exclude \ldots>$ meet the conditions of conditional term rewriting systems as given in [DO90]. (Beside external predicates* *there are no conditions, that could violate the definition of* standard conditional rewriting system.*) If a termination order exists for an extended term rewriting system, [DO90, Theorem 6.1] yields the requested statement: Testing critical pairs is sufficient to prove confluence.*

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

Theorem 2.11 *To check an extended trs T for termination and local confluence it is sufficient to check a finite subset of $T' = \varphi(T)$.*

Note that in the absence of *exclude* constructs the proofs for termination and confluence of the finite subset of $\varphi(T)$ can use the usual tools. For handling *exclude* constraints one has to take into account the Remarks 2.5 and 2.10.

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 [KB67] 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.

2.2 Semantics oriented extensions

2.2.1 Definition of the extensions

Beside the syntactic transformations as given by term rewriting systems, simple calculations using predefined functions are helpful as well. E.g. it is possible to overcome the need of representing $\mathbb{N}$ in a way tractable by term rewriting systems. Instead of that one could use compiled functions. To handle this we introduce the notion of function application in this subsection.[5]

Moreover we need a construct to restrict the applicability of a rule to special types of arguments, because most functions are restricted in their input type. Function applications are only possible, if arguments of the right type are given. This avoids runtime errors of the term rewriting interpreter.

[5]The extensions presented in this subsection have to be used very carefully, because they cause some theoretical problems. E.g. local confluence seems to become undecidable by those extensions even for terminating term rewriting systems.

Notation: $<\mathit{test}\ p?\ x>$ unifies x exactly with all terms fulfilling the predicate $p?$. $p?$ must be defined for all terms and return a boolean value. The language to implement $p?$ depends on the implementation of the term rewriting interpreter. A prototype [Fro98] uses the LISP dialect Scheme.
$<\mathit{eval}\ (f\ x\ y\ \ldots)>$ represents the result of applying f to the arguments bound to $x, y, \ldots$. The function application can only occur at the right hand side of a rule which should contain the corresponding $<\mathit{test}\ \ldots>$ constructs at the left hand side for all arguments. The definition of appropriate input types and the implementation of the test predicates has to be done manually as a subtask of generating a term rewriting system.
The arity of functions used in $<\mathit{eval}\ldots>$ must be fixed. The extensions described in subsection 2.1.1 are not allowed inside of $<\mathit{eval}\ldots>$.

Example: The rule

$$(+\ (*\ <\mathit{test\ number}?\ a>\ x)\ (*\ <\mathit{test\ number}?\ b>\ x)\ <1\ldots>)$$
$$\to (+\ (*\ <\mathit{eval}\ (+\ a\ b)>\ x)\ <1\ldots>)$$

transforms $(+\ (*\ 3\ n)\ (*\ 5\ n)\ m)$ to $(+\ (*\ 8\ n)\ m)$. The rule is not applicable to the term $(+\ (*\ 3\ n)\ (*\ k\ n)\ m)$, because $<\mathit{test\ number}?\ b>$ does not permit binding k to b.

2.2.2 Effects of the semantics oriented extensions to termination and confluence

Termination: There are several different ways how the extensions presented in this subsection can violate termination of a trs:

1. The predicate in some $<\mathit{test}\ldots>$ does not terminate.
2. The evaluation of $<\mathit{eval}\ldots>$ does not terminate for certain arguments that are allowed according to the tests at the left hand side.
3. Termination is violated in the usual way, i.e. there is a non-terminating sequence of rule applications.

To avoid termination problems according to 1. and 2., termination must be proved for every *test*-predicate and every input argument as well as for every *eval*-function and every set of valid input arguments. Moreover one has to prove the functions being defined for all valid input arguments to avoid runtime errors.
Missing termination according to 3. is provable by an appropriate termination order. E.g. one has often function applications for a function $f : \mathbb{M}^k \to \mathbb{M}$ with an arbitrary set $\mathbb{M}$ and $k \geq 2$. In this case a termination ordering should incorporate the number of elements of type $\mathbb{M}$.

Confluence: For term rewriting systems defined by using some of the semantics oriented extensions we cannot expect confluence to be still decidable:
Local confluence states that the results of two different rule applications to a term have a common normal form. When we permit any function, local confluence is no more provable just by regarding the critical pairs, but we have to prove local confluence for any variable assignment. This is impossible, because the functions used in the evaluation construct can cause any kind of special cases, implying the necessity of a complete function analysis. Only this can detect that e.g. the following function needs a special treatment for $x = 42$:

```
(lambda (x y)
   (if (= x 42) 5 (* x y)))
```

This example shows that when used without care the semantics oriented extensions destroy the main benefit of term rewriting systems of having well understood methods of proving the main properties.
But we can recommend the use of these extensions for transformations that one could describe using term rewriting systems but that can be performed with more efficiency by compiled functions. We will describe this on the example of integer addition:

$$\begin{aligned}(succ\ (pred\ x)) &\rightarrow x\\ (pred\ (succ\ x)) &\rightarrow x\\ (add\ x\ zero) &\rightarrow x\\ (add\ x\ (pred\ y)) &\rightarrow (add\ (pred\ x)\ y)\\ (add\ x\ (succ\ y)) &\rightarrow (add\ (succ\ x)\ y)\end{aligned}$$

If we represent integers in some machine or programming language specific representation, the following extended rule could be used:

$$(add\ \ <test\ integer?\ x>\ <test\ integer?\ y>) \rightarrow <eval\ (+\ x\ y)>$$

In addition to that, the remaining rules that do not depend on the representation of integers must be converted to the changed representation of constants:

$$(add\ x\ zero) \rightarrow x$$

is transformed to

$$(add\ x\ 0) \rightarrow x$$

Rules for handling nested pairs of successor and predecessor need not to be converted, because the representation is no longer based on iterated successor or predecessor application to 0.

This simple example shows how the semantics based extensions can be useful in spite of their theoretical problems, if one first defines a term rewriting system without these extensions and improves its efficiency using the new extensions after termination and confluence have been proved for the original term rewriting system.

3 Normalizing list expressions

List expressions are built by the following operators:

- car
- cdr
- cons
- list

Furthermore there are abbreviations for sequences up to four calls to *car* and *cdr*. But it is trivial to transform these into expressions just containing the operators listed above:

$$\begin{aligned}
(caar\ x) &\to (car\ (car\ x)) \\
(cadr\ x) &\to (car\ (cdr\ x)) \\
(cddr\ x) &\to (cdr\ (cdr\ x)) \\
&\ \vdots
\end{aligned}$$

Expressions built by the other operators can be transformed to a normal form, that is unique for equivalent list expressions:

$$\begin{aligned}
(car\ (cons\ x\ y)) &\to x \\
(cdr\ (cons\ x\ y)) &\to y \\
(list) &\to () \\
(list\ x) &\to (cons\ x\ ()) \\
(list\ x\ \ <1\ldots>)\quad <<[notarity\ (1\ 0)]>> &\to (cons\ x\ (list\ \ <1\ldots>))
\end{aligned}$$

The whole trs for list transformation consisting of the elimination of abbreviations and the following normalization is denoted TRS_L.

Theorem 3.1 *The term rewriting system TRS_L is terminating.*

Proof: The statement follows directly from a recursive path ordering with the following partial order on the set of operators:

- $c\{a,d\}^{k\geq 2}r > car,\ cdr$
- $list > cons$

□

Theorem 3.2 *The term rewriting system TRS_L is confluent.*

Proof: Proving confluence is trivial since there are no critical pair to be built. The resulting local confluence together with termination as proved in theorem 3.1 yields confluence. □

The result of normalizing an arbitrary list expression is an equivalent expression, containing no function other than *cons*, *car* and *cdr*. The use of *car* or *cdr* on a *cons* expression is replaced by its result.
The representation is unique for every list, no matter how the list was presented originally.

4 Normalizing arithmetic expressions

4.1 General considerations

In this section we want to normalize arithmetic terms. We restrict ourselves to terms just containing the functions $+$, $-$, $*$ and $/$. If other functions occur in a term this does not cause further opportunities of simplification to arise, but these functions are handled just as constants (except of simplifying their arguments).

To formulate our term rewriting systems more easily we first of all replace k-ary occurrences of $-$ and $/$ by unary occurrences of these functions in connection with $+$ and $*$ respectively. Moreover the unary functions for successor and predecessor are replaced by addition of 1 and -1, respectively. This pre-normalization is done by the following term rewriting system TRS_{A_1}:

$$\begin{aligned}(1+\ x) &\rightarrow (+\ 1\ x)\\ (-1+\ x) &\rightarrow (+\ -1\ x)\\ (-\ a\ \ <1\ldots>)\ \ <<[\mathit{notarity}\ (1\ 0)]>> &\rightarrow (+\ a\ (-\ (+\ \ <1\ldots>)))\\ (/\ a\ \ <1\ldots>)\ \ <<[\mathit{notarity}\ (1\ 0)]>> &\rightarrow (*\ a\ (/\ (*\ \ <1\ldots>)))\end{aligned}$$

The normalization of arithmetic expressions cannot be solved satisfactorily by a single term rewriting system. Hence we use a multi level rewriting system as described in [Bau85]. We identify several steps of normalizing arithmetic expressions. Most of these steps are performed by a (maybe multi level) term rewriting systems. There is only one step (reducing of a fraction) that leaves the framework of term rewriting. The single steps are put together in an order making sure that the result of every step fulfills the conditions on the input of the next step.

The following subtasks are performed:

1. Eliminating double fractions, determining a common denominator and multiplying out the term.
2. Multiplying out and simplification of sums (adding of inverse numbers, adding 0, calculating sums of constants.).
3. Factoring out subexpressions and simplification of products (reducing, multiplication with 1 or 0).

We will present the solutions to these subtasks in the order in which they are performed.

4.1.1 Eliminating nested fractions

The elimination of nested fractions is done by the following term rewriting system TRS_{A_2}:

$$(/\ (/\ x)) \to x \tag{1}$$

$$(-\ (-\ x)) \to x \tag{2}$$

$$(*\ <1\ldots>\ (*\ <2\ldots>)\ <3\ldots>) \to (*\ <1\ldots><2\ldots><3\ldots>) \tag{3}$$

$$(+\ <1\ldots>\ (+\ <2\ldots>)\ <3\ldots>) \to (+\ <1\ldots><2\ldots><3\ldots>) \tag{4}$$

$$(/\ (*\ <a_1\ldots e_1>)) \to (*\ <(/\ a_1)\ldots(/\ e_1)>) \tag{5}$$

$$(-\ (+\ <a_1\ldots e_1>)) \to (+\ <(-\ a_1)\ldots(-\ e_1)>) \tag{6}$$

$$\begin{aligned}&(*\ <1\ldots>\ (+\ <a_2\ldots e_2>)\ <3\ldots>) \to\\ &\qquad (+\ <(*\ <1\ldots>\ a_2\ <3\ldots>)\ldots(*\ <1\ldots>\ e_2\ <3\ldots>)>)\end{aligned} \tag{7}$$

$$\begin{aligned}&(/\ (+\ <a_1\ldots e_1>\ (/\ y)\ <a_2\ldots e_2>)) \to\\ &\qquad (*\ y\ (/\ (+\ <(*\ y\ a_1)\ldots(*\ y\ e_1)>\ 1\ <(*\ y\ a_2)\ldots(*\ y\ e_2)>)))\end{aligned} \tag{8}$$

$$\begin{aligned}&(/\ (+\ <a_1\ldots e_1>\ (*\ <2\ldots>\ (/\ y)\ <3\ldots>)\ <a_4\ldots e_4>)) \to\\ &\quad (*\ y\ (/\ (+\ <(*\ y\ a_1)\ldots(*\ y\ e_1)>\ (*\ <2\ldots><3\ldots>)\ <(*\ y\ a_4)\ldots(*\ y\ e_4)>))\end{aligned} \tag{9}$$

Theorem 4.1 *TRS_{A_2} is terminating.*

Proof: Consider the a recursive path ordering with the following order on the set of operators:

$$- \ > \ / \ > \ * \ > \ + \ > \begin{cases} 0 \\ 1 \end{cases}$$

This is a termination order for TRS_{A_2}. $\square$

Confluence does not hold for TRS_{A_2}, as the following example (in standard notation) shows:

$$\frac{1}{x+\frac{1}{y+\frac{1}{z}}} \overset{(8)}{\longrightarrow} \frac{1}{x+z\cdot\frac{1}{z\cdot y+1}} \overset{(9)}{\longrightarrow} (zy+1)\cdot\frac{1}{(zy+1)\cdot x+z} \overset{(7)}{\longrightarrow} (zy+1)\cdot\frac{1}{zyx+1\cdot x+z} \overset{(7)}{\longrightarrow}$$
$$zy\cdot\frac{1}{zyx+1\cdot x+z}+1\cdot\frac{1}{zyx+1\cdot x+z}$$

$$\frac{1}{x+\frac{1}{y+\frac{1}{z}}} \overset{(8)}{\longrightarrow} (y+\frac{1}{z})\cdot\frac{1}{(y+\frac{1}{z})x+1} \overset{(7)}{\longrightarrow} (y+\frac{1}{z})\cdot\frac{1}{yx+\frac{1}{z}x+1} \overset{(9)}{\longrightarrow} (y+\frac{1}{z})\cdot z\cdot\frac{1}{zyx+x+z\cdot 1}$$
$$\overset{(7)}{\longrightarrow} yz\cdot\frac{1}{zyx+x+z\cdot 1}+\frac{1}{z}z\cdot\frac{1}{zyx+x+z\cdot 1}$$

The two different resulting terms are in normal form with respect to TRS_{A_2}.

Since we use a multi level term rewriting system, confluence of the single stages is not necessary. For all critical pairs of this stage, the normal forms of the whole system are equivalent with respect to the system's restrictions (equivalence modulo commutativity, ...). In the given example the common normal form is reached by the stage of product simplification that is presented in Sec. 4.3.

The use of this stage becomes obvious by the following theorem:

Theorem 4.2 *The output of TRS_{A_2} does not contain any nested fractions.*

Proof: A nested fraction can occur in two different ways:

- The denominator of the main fraction consists of a sum which contains a fraction as factor of one of the addend.
- The denominator of the main fraction consists of a product with a fraction as factor.

If a fraction in a denominator occurs as addend of a factor, the rule for multiplying out yields one of the cases above.

Case 2 is solved by the rules 5 and 1: First the main fraction is decomposed into several fractions which denominators consist of the factors of the original denominator. This directly yields two sequent division operators that can be eliminated.

Case 1 is handled by the rules 8 and 9 which eliminate one nested fraction a time by multiplying numerator and denominator with its reciprocal.

Altogether given an arbitrary expression the output of TRS_{A_2} is an equivalent expression not containing any nested fractions, i.e. no argument of an /-operator contains another /-operator. □

By elimination of nested fractions the set of valid input values may change: If the inner denominator becomes 0 for some variable assignment, the original expression is not defined. After normalization, the whole fraction becomes 0. Since the old set of valid arguments is contained in the new one and runtime error are never the expected output of a program we can tolerate the described change of semantics.

4.1.2 Unifying the denominators

In this subsection we show how the different denominators occurring in an expression can be unified. The goal is the decomposition into numerator and denominator both consisting of a polynomial.
The decomposition of expressions into numerator and denominator is possible only if the expression contains at least one division operator /. The term rewriting system TRS_{A_3} performs some simplification and generates a common fraction from the different fractions occurring in the expression. The term rewriting system TRS_{A_4} generates a common denominator for all fractions occurring in different addends.

TRS_{A_3}:

$$
\begin{aligned}
(-\ 0) &\to 0 \\
(-\ (-\ x)) &\to x \\
(+\ <1\ldots>\ 0\ <2\ldots>) &\to (+\ <1\ldots><2\ldots>) \\
(+\ <1\ldots>\ x\ <2\ldots>\ (-\ x)\ <3\ldots>) &\to (+\ <1\ldots><2\ldots><3\ldots>) \\
(+\ <1\ldots>\ (-\ x)\ <2\ldots>\ x\ <3\ldots>) &\to (+\ <1\ldots><2\ldots><3\ldots>) \\
(+) &\to 0 \\
(+\ x) &\to x \\
(-\ (+\ <a_1\ldots e_1>)) &\to (+\ <(-\ a_1)\ldots(-\ e_1)>) \\
(+\ <1\ldots>\ (+\ <2\ldots>)\ <3\ldots>) &\to (+\ <1\ldots><2\ldots><3\ldots>)
\end{aligned}
$$

$$
\begin{aligned}
&(*\ <1\ldots>\ (+\ <a_2\ldots e_2>)\ <3\ldots>) \to \\
&\qquad\qquad (+\ <(*\ <1\ldots>\ a_2\ <3\ldots>)\ldots(*\ <1\ldots>\ e_2\ <3\ldots>)>)
\end{aligned}
$$

$$
\begin{aligned}
&(<comm\ *><choose\ <(/\ a_1)\ldots(/\ e_1)>><2\ldots>)\quad <<[notarity\ (1\ 0)]>> \\
&\qquad\qquad <<[notarity\ (1\ 1)]>>\ \to (*\ <2\ldots>\ (/\ (*\ <a_1\ldots e_1>)))
\end{aligned}
$$

Theorem 4.3 *The term rewriting system TRS_{A_3} is terminating.*

Proof: We prove the statement by a recursive path order with the following partial order on the set of function symbols:

- $+ \; < \; -$
- $+ \; < \; *$
- $/ \; < \; *$

Termination of the last rule according to this termination order becomes obvious by the observation that the commutative choice cannot violate termination and that the rule without commutativity can be handled as

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

extended by some non-fraction factors or as

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

by using associativity. □

Theorem 4.4 *The term rewriting system TRS_{A_3} is confluent.*

Proof: All critical pairs have a common normal form. This implies local confluence, and confluence follows from this together with termination as proved in theorem 4.3. □

The following term rewriting system TRS_{A_4} performs the multiplications of numerator and denominator necessary to obtain a common denominator, if several fractions occur in different addends of a sum. TRS_{A_4} just works on those addends already containing a fraction. The remaining addends will be multiplied with the common denominator when it is factored out. The applicability of a rule for generating a common denominator may not be given because of a number of special cases of missing products. Handling all these cases independently results in a great number of similar rules. We handle these cases differently by introducing products where there are missing and thus obtaining the general case.

TRS_{A_4}:

$$\begin{aligned} (+ \;\; <1\ldots> \;\; (/\ y) \;\; <2\ldots>) &\rightarrow (+ \;\; <1\ldots> \;\; (*\ (/\ y)) \;\; <2\ldots>) \\ (/\ x) \quad <<[\mathit{exclude}\ (* \;\; <1\ldots>)\ x]>> \quad &\rightarrow (/\ (*\ x)) \end{aligned}$$

$$
\begin{gathered}
(+\ <1\ldots>\ (*\ <2\ldots>\ (/\ (<\mathit{comm}\ *>\ <\mathit{choose}\ \ <3\ldots>>\ <4\ldots>))\ \ <5\ldots>) \\
<6\ldots>\ (*\ <7\ldots>\ (/\ (<\mathit{comm}\ *>\ <\mathit{choose}\ \ <3\ldots>>\ <8\ldots>))\ \ <9\ldots>) \\
<10\ldots>)\ \ \ <<[\mathit{notarity}\ (4\ 0)\ (8\ 0)]>>\ \ \rightarrow \\
(+\ <1\ldots>\ (*\ <2\ldots>\ <8\ldots>\ (/\ (*\ <3\ldots>\ <4\ldots>\ <8\ldots>))\ \ <5\ldots>) \\
<6\ldots>\ (*\ <7\ldots>\ <4\ldots>\ (/\ (*\ <3\ldots>\ <4\ldots>\ <8\ldots>))\ \ <9\ldots>)\ \ <10\ldots>)
\end{gathered}
$$

Theorem 4.5 *The term rewriting system TRS_{A_4} terminates when applied to any term t with the following properties:*

1. *t does not contain nested fractions.*
2. *Every subterm of t is fully multiplied out.*
3. *Every product contains at most one fraction.*

Proof: In this case we do not prove termination by a termination order but directly by considering the transformation process for a given term t.
Without loss of generality let t contain at least one fraction. (Otherwise t is already in normal form.) Inspecting the rules shows that no transformation step can change the number of fractions in t and no transformation yields a term violating the conditions on the input terms of TRS_{A_4}.
All fractions occur in addends of the same sum (may be as factors of a product), because nested fractions and several sums are in contradiction to the preconditions.
The common denominator of all fractions in t built as product of all different factors (several identical factors are not necessary since the original denominators were fully multiplied out) of the original denominators does not change during normalization:

- The first rule does not change any denominators in t.
- The second rule does not change the common denominator, because it just puts denominators not consisting of a product into a product with one element. Before and after that transformation step the original denominator must be one of the factors of the common denominator.
- The last rule works on exactly two denominators which must consist of a product. If t contains only one fraction, we have nothing to prove, because the rule does not apply. So let t contain at least two fractions. For every two fractions b_1 and b_2 the set $H_{1,2}$ of factors in a common denominator of b_1 and b_2 is a subset of the set H of factors in the common denominator of all fractions in t.
  Let H_i be the set of factors in the denominator of b_i ($i \in \{1, 2\}$). Assume the application of rule three, where b_1 is unified with $(/\ (<\mathit{comm}\ *>\ <\mathit{choose}\ \ <3\ldots>>\ <4\ldots>))$

and b_2 with $(/\ (<\mathit{comm}\ *>\ <\mathit{choose}\ \ <3\ldots>>\ <8\ldots>))$. The argument list $<4\ldots>$ contains exactly the factors from $H_{1,2} \setminus H_2$, and the argument list $<8\ldots>$ contains the elements of $H_{1,2} \setminus H_1$. Thus the result of the transformation step is a term where b'_1 and b'_2 derived from b_1 and b_2 respectively contain exactly the factors in $H_{1,2}$. Because of $H_{1,2} \subseteq H$ the common denominator of all fractions in t did not change during this transformation step.

We will now prove every sequence $t = t_1 \rightarrow t_2 \rightarrow \ldots$ of normalization steps to be finite since the number of applications is already bound by the input term t for every rule:

- The first rule reduces the number of denominators not occurring inside a product by 1 per application. No rule releases a denominator from a product and hence the number of applications of rule 1 is bound by the number of denominators originally not occurring inside a product and therefore bound by the number n of fractions occurring in t.

- The number of applications of the second rule is also bound by n, because every denominator has to be transformed into product form at most once and no rule takes back this transformation.

- For the number of applications of the third rule one should note that to every fraction b the number of factors missing in b's denominator to reach the common denominator is known. This number cannot increase during normalization, because the common denominator to be reached is fixed and the number of factors in b's denominator cannot decrease.
  The number of fractions in t is fixed during normalization and therefore we can conclude from the considerations above that the overall number of factors missing in any of the denominators cannot increase.
  When rule 3 is applied, at least one of the argument lists $< 4 \ldots >$ and $< 8 \ldots >$ is non-empty. Hence the number of missing factors decreases by at least 1 for at least one fraction. Since the overall number of missing factors must be a finite natural number, the application of the third rule can be performed for a finite number of times only.

From the above we know that any of the finitely many rules can be performed only a finite number of times for every given term t. Thus the term rewriting system TRS_{A_4} is terminating under the given preconditions. □

Theorem 4.6 *The term rewriting system TRS_{A_4} is confluent for all terms fulfilling the conditions of theorem 4.5.*

Proof: There is a common normal form for every critical pair between the rules of TRS_{A_4}. This implies local confluence and together with the termination as for the valid input terms stated by theorem 4.5 follows confluence. (Note that the term rewriting system partially orders the set of valid input terms and that this partial ordering is well-founded because of the termination. This is sufficient in the prove of Knuth and Bendix that local confluence and termination imply confluence [KB67, Theorem 4].) $\square$

Theorem 4.7 *Let t be a term fulfilling the conditions of theorem 4.5, t' the result of TRS_{A_3} on t and t'' the result of TRS_{A_4} on t'.*
t'' does not contain two different denominators, i.e. there are no subexpressions $(/\ n_1)$ and $(/\ n_2)$ with $n_1 \neq n_2$.

Proof: We prove the theorem by contradiction: Assume t'' contains subexpressions $(/\ n_1)$ and $(/\ n_2)$ with $n_1 \neq n_2$.
n_1 cannot be a subexpression of n_2 and vice versa, because t did not contain any nested fractions and neither TRS_{A_3} nor TRS_{A_4} could introduce those. Therefore $(/\ n_1)$ and $(/\ n_2)$ are either factors of the same products or factors of different products (maybe consisting of exactly one factor) being different addends of the same sum. This is true, because t' is fully multiplied out and the sums introduced as factors by TRS_{A_4} do not contain fractions, because they are taken from denominators and neither t' nor the intermediate terms during normalization by TRS_{A_4} contain any nested fractions.
If the first case is true, this must hold for t' as well, because TRS_{A_4} does not contain any rules introducing multiple fractions as factors of the same product. In this case, the last rule TRS_{A_3} must have been applicable to t'. That is a contradiction to the assumption t' is normal form with respect to TRS_{A_3}.
In the second case the last rule of TRS_{A_4} is still applicable to t'' in contradiction to "t'' is normal form with respect to TRS_{A_4}". $\square$

Altogether the result of this stage for an input expression without nested fractions is an expression in which all occurrences of / have the same operand.

4.1.3 Decomposition of numerator and denominator

The output of the stage before has usually neither numerator nor denominator fully multiplied out. The following term rewriting system TRS_{A_5} eliminates nested sums and products respectively, multiplies out all products and puts negation inside of fractions:

$$(-\ (/\ x)) \to (/\ (-\ x))$$
$$(+\ <1\ldots>\ (+\ <2\ldots>)\ <3\ldots>) \to (+\ <1\ldots>\ <2\ldots>\ <3\ldots>)$$
$$(*\ <1\ldots>\ (*\ <2\ldots>)\ <3\ldots>) \to (*\ <1\ldots>\ <2\ldots>\ <3\ldots>)$$

$$(* \ {<}1\ldots{>} \ (+ \ {<}a_2\ldots e_2{>}) \ {<}3\ldots{>}) \to$$
$$(+ \ {<}(* \ {<}1\ldots{>} \ a_2 \ {<}3\ldots{>})\ldots(* \ {<}1\ldots{>} \ e_2 \ {<}3\ldots{>}){>})$$

Afterwards the unique denominator can be factored out.

Theorem 4.8 *The term rewriting system TRS_{A_5} is terminating.*

Proof: Consider the recursive path order with the following partial order on the set of operators:

- $/ \ < \ -$
- $+ \ < \ *$

This is a termination order for TRS_{A_5}. □

Theorem 4.9 *The term rewriting system TRS_{A_5} is confluent.*

Proof: Every critical pair has a common normal form. Together with Theorem 4.8 this implies confluence. □

The next term rewriting system TRS_{A_6} assures for every fraction to occur in a product (if necessary by multiplying 1) and that all these products consist of exactly two factors. One of these factors is the common denominator and the other may contain another product without any fractions:

$$(+ \ {<}1\ldots{>} \ (/ \ x) \ {<}2\ldots{>}) \to (+ \ {<}1\ldots{>} \ (* \ (/ \ x) \ 1) \ {<}2\ldots{>})$$
$$(* \ (/ \ x)) \to (* \ (/ \ x) \ 1)$$

$$({<}\mathit{comm} \ *{>} \ {<}\mathit{choose} \ \ {<}(/ \ a_1)\ldots(/ \ e_1){>}{>}{<}2\ldots{>})$$
$$\ll[\mathit{notarity} \ (2 \ 0)]\gg \quad \ll[\mathit{notarity} \ (2 \ 1)]\gg \quad \ll[\mathit{notarity} \ (1 \ 0)]\gg \to$$
$$(* \ {<}(/ \ a_1)\ldots(/ \ e_1){>} \ (* \ {<}2\ldots{>}))$$

In the third rule $< a_1 \ldots e_1 >$ contains at most one element after the normalization of the stages before. Because of the constraint $\ll [\mathit{notarity} \ (1 \ 0)] \gg$ it must be exactly one element. The only reason for using an argument list of variable arity is, that $<\mathit{choose}\ldots>$ by definition enforces the use of such a list.

To prove termination of TRS_{A_6} we define a partial order on the set of terms and prove that it is a termination order:

Definition 4.10 *The partial order $<_6$ is defined as follows: Let t_1 and t_2 be terms. $t_1 <_6 t_2$ holds, iff:*

1. *For every variable the number of occurrences in t_1 and t_2 is equal.*
2. *For every variable the number of occurrences as addend of a sum is equal in t_1 and t_2.*
3. *There are as many subterms of the form $(*\ y)$ for every variable y in t_1 as in t_2 and vice versa.*
4. *To every position t_{1_i} of t_1 in the form of a product with more than one factor not of the form $(/\ t)$, there is a position t_{2_i} of t_2 in form of a product fulfilling the following conditions:*
   - *t_{2_i} contains at least as many factors not of the form $(/\ t)$ as t_{1_i}.*
   - *t_{2_i} contains every variable as factor at most as often as t_{1_i}.*

   The function mapping t_{1_i} to t_{2_i} is injective, i.e. from $t_{1_p} \neq t_{1_q}$ one can conclude $t_{2_p} \neq t_{2_q}$.
5. *t_1 has the form $(/\ t)$ with a term t or x with a variable x, if t_2 has the same form and vice versa.*
6. *Let t be a term, $SB(t)$ the number of subterms of t matching the pattern $(/\ t')$ and occurring as an addend in t, $PE(t)$ the number of subterms of the form $(*\ (/\ t'))$ and $PN(t)$ the number of products with more than one factor not of the form $(/\ t')$. One of the following conditions holds:*
   (a) $SB(t_1) < SB(t_2)$
   (b) $(SB(t_1) = SB(t_2)) \wedge (PE(t_1) < PE(t_2))$
   (c) $(SB(t_1) = SB(t_2)) \wedge (PE(t_1) = PE(t_2)) \wedge (PN(t_1) < PN(t_2))$

Lemma 4.11 *$<_6$ is well-founded.*

Proof: For a given term t $SB(t)$, $PE(t)$ and $PN(t)$ are natural numbers. The order $<$ on the natural numbers is well-founded. By choosing a term t' with $t' <_6 t$ $PE(t')$ $PN(t')$ can only be larger than $PE(t)$ or $PN(t)$ respectively if a higher order value ($SB(t)$ for $PE(t)$ and $PN(t)$, $PE(t)$ for $PN(t)$) decreases. This is possible only a finite number of times. Thus $<_6$ is well-founded. □

Lemma 4.12 *$<_6$ is invariant under variable assignment.*

Proof: Regard an assignment of variable x in t_1 and t_2, where $t_1 <_6 t_2$. The results are denoted t_1' and t_2' respectively. No assignment violates the condition 1.. For the other conditions we have to regard the following cases of assignment:

- A term s matching the pattern $(/\ s')$ is assigned to x. This only influences the conditions 2. and 3., if s' contains sums or terms of the form $(*\ (/\ t))$ as subterms. Even in this case, the conditions are not violated because of 1..

  Regarding condition 4. there is no change for the not assigned variables as factors. Since x occurs in any product t_{1_i} in t_1 at least as often as in the associated Product t_{2_i} in t_2 the instantiation changes in t_{2_i} at most as many factors not of the form $(/\ t)$ as in t_{1_i}. After the assignment t_{1_i} still contains at most as many factors not of the form $(/\ t)$ as t_{i_2}. If s' itself contains products as subterms, every product is introduced in both terms equally often because of 1.. Therefore we can built new pairs of equal products from the new ones and condition 4. remains valid.

  Because of condition 2. and the fact that new sums are introduced in the same form and number as subterms of s' according to 1., we have $SB(t_1') - SB(t_1) = SB(t_2') - SB(t_2)$ which implies:

  - $SB(t_1) < SB(t_2) \Rightarrow SB(t_1') < SB(t_2')$
  - $SB(t_1) = SB(t_2) \Rightarrow SB(t_1') = SB(t_2')$

  analogously we have $PE(t_1') - PE(t_1) = PE(t_2') - PE(t_2)$ because of 3. and 1. and can conclude

  - $PE(t_1) < PE(t_2) \Rightarrow PE(t_1') < PE(t_2')$
  - $PE(t_1) = PE(t_2) \Rightarrow PE(t_1') = PE(t_2')$

  The remaining validity of 4. especially implies that for every product in t_1' having more than one factor not of the form $(/\ t)$ there is a corresponding product in t_2'. This implies $PN(t_1') < PN(t_2')$.
  Altogether 6. remains valid after the assignment.

  From checking all conditions we can conclude $t_1' <_6 t_2'$.

- A term s not of the form $(/\ s')$ is assigned to x.
  Since 2. holds for t_1 and t_2 (especially for x), the same number of variables being addends are instantiated in t_1 and t_2. Moreover because of 1. the term s is inserted into t_1 and t_2 equally often. This holds especially for all sums being subterms of s with there variables occurring as addends. Therefore 2. holds for t_1' and t_2'.

  Subterms of the form $(*\ y)$ with a variable y are not changed for $x \neq y$. For $x = y$ there is no change either, if another variable is assigned to x. If any other term is assigned to

x, the occurrences of $(*\ x)$ are not in the scope of 3. any more. The form and type of subterms of matching the pattern $(*\ (/\ z))$ and being inserted by s are identical in t'_1 and t'_2 because of 1.. Thus condition 3. remains valid.

As far as condition 4. is concerned the new products in s are no problem, because the same number and kind of products is inserted into t_1 and t_2 according to 1.. From these new products one can build pairs of identical products fulfilling the condition 4..
For the products already occurring in t_1 or t_2 respectively the only problem are assignments transforming factors from a form other than $(/\ t)$ to that form. This can just happen by assigning $(/\ t)$ to a variable x occurring as a factor and is therefore impossible in the actual case.

Condition 5. causes no problem if t_1 and t_2 are of the form $(/\ t)$, because this fact cannot change by any assignment. If t_1 and t_2 are variables, it must be the same one in both cases because of 1.. an assignment of this variable transforms both terms to the same form.

The change of $SB(t_1)$ and $SB(t_2)$ is only affected by introducing new sums. According to 1. this is done for an identical number for both terms, implying $SB(t'_1) - SB(t_1) = SB(t'_2) - SB(t_2)$. This implies:

- $SB(t_1) < SB(t_2) \Rightarrow SB(t'_1) < SB(t'_2)$
- $SB(t_1) = SB(t_2) \Rightarrow SB(t'_1) = SB(t'_2)$

$PE(t_1)$ and $PE(t_2)$ can only change, if s contains subterms of the form $(*\ (/\ t))$. Because of 1. an identical number of subterms of this form are inserted into both terms and therefore $PE(t'_1) - PE(t_1) = PE(t'_2) - PE(t_2)$ holds and implies:

- $PE(t_1) < PE(t_2) \Rightarrow PE(t'_1) < PE(t'_2)$
- $PE(t_1) = PE(t_2) \Rightarrow PE(t'_1) = PE(t'_2)$

As shown condition 4. holds for t'_1 and t'_2. Thus especially for any product t'_{1_j} in with more than one factor not of the form $(/\ y)$ there is a product t'_{2_j} in t'_2 fulfilling the same condition. Therefore $PN(t'_1) < PN(t'_2)$ holds which implies condition 6. to be valid. In this case we have $t'_1 <_6 t'_2$, too.

From checking all cases we can conclude $<_6$ the theorem to hold. □

Lemma 4.13 *$<_6$ is monotonic.*

Proof: Let f be a function symbol and $r_1, \dots, r_{i-1}, r_{i+1}, \dots r_n$ terms. We regard $t'_1 = (f\ r_1 \dots r_{i-1}\ t_1\ r_{i+1} \dots r_n)$and $t'_2 = (f\ r_1 \dots r_{i-1}\ t_2\ r_{i+1} \dots r_n)$, where t_1 and t_2 fulfill $t_1 <_6 t_2$. We insert t_1 and t_2 into equal terms, and for t_1 and t_2 condition 1.. Thus this is also true for

t_1' and t_2'. Condition 5. holds for t_1' and t_2', because only f determines whether these have the form $(/\ y)$, and the case of t_1' or t_2' being a single variable is impossible.

2. can only cause problems, if $f = +$ and t_1 or t_2 is a variable. Only none of t_1 and t_2 or both can be a variable because of 5. and because of 1. these variables must be the same. Therefore 2. holds for t_1' and t_2'.

A change of 3. can only take place for $f = *$ with arity 1 and t_1 or t_2 being a single variable. Because of 5. and 1. both terms must be consist of the same variable in that case which implies 3. to remain valid.

Also for 4. a change is only possible if $f = *$ and one of the terms t_1 and t_2 consists of a variable or matches the pattern $(/\ t)$. Because of 5. the other term has the same form, and in case of variables, they are identical because of 1.. This keeps condition 4. valid for t_1' and t_2'.

For $SB(\cdot)$ it is important to note that t_1 and t_2 can consist of a variable only together because of 5. Even for $f = +$ we have $SB(t_1') - SB(t_1) = SB(t_2') - SB(t_2)$ and hence

- $SB(t_1) < SB(t_2) \Rightarrow SB(t_1') < SB(t_2')$
- $SB(t_1) = SB(t_2) \Rightarrow SB(t_1') = SB(t_2')$

Moreover t_1 and t_2 can match the pattern $(/\ t)$ only together and therefore even for $f = *$ and arity 1 we have $PE(t_1') - PE(t_1) = PE(t_2') - PE(t_2)$ this implies:

- $PE(t_1) < PE(t_2) \Rightarrow PE(t_1') < PE(t_2')$
- $PE(t_1) = PE(t_2) \Rightarrow PE(t_1') = PE(t_2')$

As in lemma 4.12 from the validity of 4. for t_1' and t_2' we can conclude $PN(t_1') < PN(t_2')$ and from that the validity of 6. for t_1' and t_2'.

All conditions are valid for the results t_1' and t_2' of insertion. We have $t_1' <_6 t_2'$ and therefore monotonicity of $<_6$. $\square$

Theorem 4.14 *The term rewriting system TRS_{A_6} is terminating.*

Proof: We prove that $<_6$ is a termination ordering.
$<_6$ is known to be well-founded from lemma 4.11, it is invariant under variable assignments as shown in lemma 4.12 and it is known to be monotonic from Lemma 4.13.
For every rule $l \to r$ we have $r <_6 l$ as shown below:

1. Condition 1. holds for all rules.

2. Condition 2. trivially holds for the rules 2 and 3. It holds for rule 1, too, because the only changed addend is neither a variable before nor after transformation.

3. For the first rule condition 3., because the only manipulation of products consists of introducing a new product not matching the regarded pattern.
The rules 2 and 3 only handle products containing at least one fraction as factor and therefore not being regarded by condition 3..

4. The only products being processed be the rules 1 and 2 contain no or exactly one factor not matching the form $(/\ t)$. They therefore do not violate condition 4.. The product at the right hand side of rule 3 also contains only one factor not of the form $(/\ t')$. This product cannot violate condition 4. and therefore the condition holds for rule 3, too.

5. Condition 5. holds for all rules trivially.

6. To prove the validity of condition 6. we consider the rules one by one:

   - For rule one we have $SB(r_1) = SB(l_1) - 1$. Therefore condition 6. holds.
   - For rule 2 we have $SB(r_2) = SB(l_2)$ and $PE(r_2) = PE(l_2) - 1$. Condition 6. also holds for this rule.
   - For rule 3 we have $SB(r_3) = SB(l_3)$, $PE(r_3) = PE(l_3)$ and $PN(r_3) = PN(l_3) - 1$. The condition 6. therefore holds for rule 3, too.

   Altogether we have validity of condition 6. for all rules.

For all rules we have shown $r <_6 l$ and together with the necessary conditions shown before this implies that $<_6$ is a termination ordering for the term rewriting system TRS_{A_6}. $\square$

Theorem 4.15 *The term rewriting system TRS_{A_6} is confluent.*

Proof: There are no critical pairs between the rules of TRS_{A_6}. This implies local confluence and together with the termination proved in theorem 4.14 we have confluence. $\square$

Theorem 4.16 *If t is a normal form of TRS_{A_6}, every denominator that is a real subterm of t occurs inside a product, and every product having a denominator as factor has exactly two factors.*
The property of every product containing at most one denominator is not violated.

Proof: The proof is straightforward by regarding the single rules and excluding all terms that are no normal forms. $\square$

We need a further normalization stage TRS_{A_7} to eliminate negation that was shifted into the common denominator. This is done by negating the other factor of the product and eliminating double negation. Simultaneously the denominator is moved to the second position inside the product:

$$\begin{aligned} (* \; x \; (/ \; (- \; y))) \;\; << [exclude \; (/ \; t) \; x] >> \; &\rightarrow (* \; (- \; x) \; (/ \; y)) \\ (* \; (/ \; x) \; y) \;\; << [exclude \; (/ \; t) \; y] >> \; &\rightarrow (* \; y \; (/ \; x)) \\ (- \; (- \; x)) &\rightarrow x \end{aligned}$$

Theorem 4.17 *The term rewriting system TRS_{A_7} is terminating.*

Proof: Termination is obvious, because the first rule decreases the number of of negated denominators, the second rule decreases the number of products whose second factor is not a denominator, and the third rule decreases the number of negation operators. All of these values are natural numbers and none of them is increased by another rule. An ordering based on these values in an hierarchical order is monotonic with respect to the valid terms, and the rules remain decreasing under variable assignment not leaving the scope given by the normalization stages above. □

Theorem 4.18 *The term rewriting system TRS_{A_7} is confluent.*

Proof: The theorem directly follows from all critical pairs having a common normal form and the termination proved in theorem 4.17. □

The last task is to factor out the common denominator. This is done be the term rewriting system TRS_{A_8} consisting of the following rule:

$$\begin{aligned} &(< comm \; + > \; < choose \;\; < (* \; a_1 \; (/ \; x)) \ldots (* \; e_1 \; (/ \; x)) >> \; < a_2 \ldots e_2 >) \\ &\qquad << [notarity \; (1 \; 0)] >> \; \rightarrow (* \; (+ \;\; < a_1 \ldots e_1 > \; < (* \; x \; a_2) \ldots (* \; x \; e_2) >) \; (/ \; x)) \end{aligned}$$

Theorem 4.19 *The term rewriting system is terminating and confluent for all valid input terms given by the stages above.*

Proof: The statement is trivial, because after normalizing a term by the stages above there is at most one position where the rule can be applied at most once. (Note that neither $< a_1 \ldots e_1 >$ or $< a_2 \ldots e_2 >$ nor x contains any fractions and therefore no sum in the result can contain fractions.) □

After performing this normalization the polynomials forming numerator and denominator can be separated for further processing.

4.2 Simplification of sums

The simplification of sums is divided into the following parts:

1. Complete multiplying out numerator and denominator.
2. Simplification of the sums by eliminating addition of 0, elimination of inverse addends and calculation of constant sums.

4.2.1 Multiplying out

Multiplying out is performed separately for numerator and denominator. In combination with applying associativity, this is done by the following term rewriting system TRS_{A_9}:

$$\begin{aligned} (-\ (+\ <a_1\dots e_1>)) &\to (+\ <(-\ a_1)\dots(-\ e_1)>) \\ (+\ <1\dots>\ (+\ <2\dots>)\ <3\dots>) &\to (+\ <1\dots><2\dots><3\dots>) \\ (*\ <1\dots>\ (*\ <2\dots>)\ <3\dots>) &\to (*\ <1\dots><2\dots><3\dots>) \end{aligned}$$

$$\begin{aligned} &(*\ <1\dots>\ (+\ <a_2\dots e_2>)\ <3\dots>) \to \\ &\qquad (+\ <(*\ <1\dots>\ a_2\ <3\dots>)\dots(*\ <1\dots>\ e_2\ <3\dots>)>) \end{aligned}$$

Theorem 4.20 *The term rewriting system TRS_{A_9} is terminating.*

Proof: The statement directly follows from a recursive path ordering with the order $- > * > / > +$ on the set of operators. □

Theorem 4.21 *The term rewriting system TRS_{A_9} is confluent.*

Proof: All critical pairs have a common normal form. This implies local confluence and together with the termination proved in theorem 4.20 confluence. □

Theorem 4.22 *For an input term only containing the operators $+$, $-$, $*$, the output is a term fully multiplied out.*

Proof: The main task of multiplying out is done by rule four. This rule applies for every product containing a sum as factor directly. It also applies to negated sums as factors after applying rule 1, and nested sums or products cannot prevent the application, because these are eliminated by the rules 2 and 3. □

4.2.2 Simplifying sums

A sum fully multiplied out can be simplified by the following term rewriting system $TRS_{A_{10}}$.

$$\begin{aligned}
(-\ 0) &\to 0\\
(-\ (-\ x)) &\to x\\
(+\ \ <1\ldots>\ \ 0\ \ <2\ldots>) &\to (+\ \ <1\ldots>\ <2\ldots>)\\
(+\ \ <1\ldots>\ \ x\ \ <2\ldots>\ \ (-\ x)\ \ <3\ldots>) &\to (+\ \ <1\ldots>\ <2\ldots>\ <3\ldots>)\\
(+\ \ <1\ldots>\ \ (-\ x)\ \ <2\ldots>\ \ x\ \ <3\ldots>) &\to (+\ \ <1\ldots>\ <2\ldots>\ <3\ldots>)\\
(*\ \ <1\ldots>\ \ 1\ \ <2\ldots>) &\to (*\ \ <1\ldots>\ <2\ldots>)\\
(*\ \ <1\ldots>\ \ -1\ \ <2\ldots>) &\to (-\ (*\ \ <1\ldots>\ <2\ldots>))\\
(+\ x) &\to x\\
(+) &\to 0\\
(*\ x) &\to x\\
(*) &\to 1\\
(*\ \ <1\ldots>\ \ 0\ \ <2\ldots>) &\to 0\\
(*\ \ <1\ldots>\ \ (-\ x)\ \ <2\ldots>) &\to (-\ (*\ \ <1\ldots>\ \ x\ \ <2\ldots>))\\
(+\ \ <1\ldots>\ \ x\ \ <2\ldots>\ \ x\ \ <3\ldots>) &\to (+\ \ <1\ldots>\ \ (*\ 2\ x)\ \ <2\ldots>\ <3\ldots>)\\
(-\ \ <\mathit{test\ number?}\ x>) &\to <\mathit{eval}\ (-\ x)>
\end{aligned}$$

$$(/\ \ <\mathit{test}\ (\mathit{lambda}\ (v)\ (\mathit{and}\ (\mathit{number}\ v)\ (\mathit{not}\ (=\ \ v\ 0))))\ x>>) \to <\mathit{eval}\ (/\ x)>$$

$$(-\ (*\ \ <1\ldots>\ <\mathit{test\ number?}\ x>\ <2\ldots>) \to (*\ \ <1\ldots>\ <\mathit{eval}\ (-\ x)>\ <2\ldots>)$$

$$\begin{aligned}
&(+\ \ <1\ldots>\ <\mathit{test\ number?}\ x>\ <2\ldots>\ <\mathit{test\ number?}\ y>\ <3\ldots>) \to\\
&\qquad\qquad (+\ \ <1\ldots>\ <\mathit{eval}\ (+\ x\ y)>\ <2\ldots>\ <3\ldots>)
\end{aligned}$$

$$\begin{aligned}
&(*\ \ <1\ldots>\ <\mathit{test\ number?}\ x>\ <2\ldots>\ <\mathit{test\ number?}\ y>\ <3\ldots>) \to\\
&\qquad\qquad (*\ \ <1\ldots>\ <\mathit{eval}\ (*\ x\ y)>\ <2\ldots>\ <3\ldots>)
\end{aligned}$$

$$\begin{aligned}
&(+\ \ <1\ldots>\ \ (<\mathit{comm}\ *>\ <\mathit{choose}\ \ <2\ldots>>)\ \ <3\ldots>\\
&\qquad (<\mathit{comm}\ *>\ <\mathit{choose}\ \ <2\ldots>>)\ \ <4\ldots>) \to\\
&\qquad\qquad (+\ \ <1\ldots>\ \ (*\ 2\ \ <2\ldots>)\ \ <3\ldots>\ <4\ldots>)
\end{aligned}$$

$$
\begin{array}{l}
(+\ {<}1\ldots{>}\ \ ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}\ {<}\mathit{test\ number?}\ x{>})\ \ {<}3\ldots{>} \\
\qquad ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}\ {<}\mathit{test\ number?}\ y{>})\ \ {<}4\ldots{>}) \rightarrow \\
\qquad\qquad (+\ \ {<}1\ldots{>}\ \ (*\ \ {<}\mathit{eval}\ (+\ x\ y){>}\ {<}2\ldots{>})\ \ {<}3\ldots{>}\ {<}4\ldots{>})
\end{array}
$$

$$
\begin{array}{l}
(+\ {<}1\ldots{>}\ \ ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>})\ \ {<}3\ldots{>} \\
\qquad ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}\ {<}\mathit{testnumber?}\ x{>})\ \ {<}4\ldots{>}) \rightarrow \\
\qquad\qquad (+\ \ {<}1\ldots{>}\ \ (*\ \ {<}\mathit{eval}\ (1+\ x)\ \ {<}2\ldots{>})\ \ {<}3\ldots{>}\ {<}4\ldots{>})
\end{array}
$$

$$
\begin{array}{l}
(+\ {<}1\ldots{>}\ \ ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}\ {<}\mathit{testnumber?}\ x{>})\ \ {<}3\ldots{>} \\
\qquad ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>})\ \ {<}4\ldots{>}) \rightarrow \\
\qquad\qquad (+\ \ {<}1\ldots{>}\ \ (*\ \ {<}\mathit{eval}\ (1+\ x)\ \ {<}2\ldots{>})\ \ {<}3\ldots{>}\ {<}4\ldots{>})
\end{array}
$$

$$
\begin{array}{l}
(+\ \ {<}1\ldots{>}\ \ (-\ ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}))\ \ {<}3\ldots{>} \\
\qquad ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}\ {<}\mathit{test\ number?}\ x{>})\ \ {<}4\ldots{>}) \rightarrow \\
\qquad\qquad (+\ \ {<}1\ldots{>}\ \ (*\ \ {<}\mathit{eval}\ (-1+\ x)\ \ {<}2\ldots{>})\ \ {<}3\ldots{>}\ {<}4\ldots{>})
\end{array}
$$

$$
\begin{array}{l}
(+\ \ {<}1\ldots{>}\ \ ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}\ {<}\mathit{test\ number?}\ x{>})\ \ {<}3\ldots{>} \\
\qquad (-\ ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}))\ \ {<}4\ldots{>}) \rightarrow \\
\qquad\qquad (+\ \ {<}1\ldots{>}\ \ (*\ \ {<}\mathit{eval}\ (-1+\ x)\ \ {<}2\ldots{>})\ \ {<}3\ldots{>}\ {<}4\ldots{>})
\end{array}
$$

$$
\begin{array}{l}
(+\ \ {<}1\ldots{>}\ \ (-\ ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}))\ \ {<}3\ldots{>} \\
\qquad ({<}\mathit{comm}\ *{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>})\ \ {<}4\ldots{>}) \rightarrow \\
\qquad\qquad (+\ \ {<}1\ldots{>}\ {<}3\ldots{>}\ {<}4\ldots{>})
\end{array}
$$

$$
\begin{array}{l}
(+\ \ {<}1\ldots{>}\ \ (-\ ({<}\mathit{comm}*{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}))\ \ {<}3\ldots{>} \\
\qquad (-\ ({<}\mathit{comm}*{>}\ {<}\mathit{choose}\ \ {<}2\ldots{>}{>}))\ \ {<}4\ldots{>}) \rightarrow \\
\qquad\qquad (+\ \ {<}1\ldots{>}\ \ (*\ -2\ \ {<}2\ldots{>})\ \ {<}3\ldots{>}\ {<}4\ldots{>})
\end{array}
$$

To prove termination of this term rewriting system we first define an appropriate term ordering:

Definition 4.23 *The term ordering $<_{10}$ is defined as follows: $t_1 <_{10} t_2$ with terms t_1 and t_2 holds, exactly if the following conditions hold:*

1. *Every variable occurs in t_1 at most as often as in t_2.*

2. *For every term t we denote the number of its addends by $S(t)$, the number of factors by $F(t)$, and the number of negation operators by $N(t)$. $<_{r_{10}}$ denotes the recursive path ordering with the partial order $- < *$ on the set of function symbols. With these notations one of the following conditions holds:*

   (a) $S(t_1) < S(t_2)$

   (b) $S(t_1) = S(t_2) \wedge F(t_1) < F(t_2)$

   (c) $S(t_1) = S(t_2) \wedge F(t_1) = F(t_2) \wedge N(t_1) < N(t_2)$

   (d) $S(t_1) = S(t_2) \wedge F(t_1) = F(t_2) \wedge N(t_1) = N(t_2) \wedge t_1 <_{r_{10}} t_2$

Lemma 4.24 *$<_{10}$ is well-founded.*

Proof: $S(t)$, $F(t)$ and $N(t)$ are natural numbers for any term t and the usual order $<$ on $\mathbb{N}$ is well-founded. $<_{r_{10}}$ as a recursive path ordering is well-founded as well.
When descending along the order $<_{10}$ the values $F(t)$ or $N(t)$ can only increase or the term can become larger with respect to $<_{r_{10}}$, if a higher order value decreases. This is possible only a finite number of times. Therefore there are no infinite sequences of terms descending according to $<_{10}$. □

Lemma 4.25 *The order $<_{10}$ is invariant under variable assignment.*

Proof: Let t_1 and t_2 be two terms fulfilling $t_1 <_{10} t_2$ and let t'_1 and t'_2 respectively be the results of assigning a term to the variable x in t_1 and t_2.

1. For t_1 and t_2 the condition 1. holds. especially x occurs in t_1 at most as often as in t_2. Thus any variable is newly introduced into t_1 at most as often as in t_2. The condition therefore remains valid for t'_1 and t'_2.

2. The number of addends already occurring in t_1 and t_2 does not change. New sums are introduced into t_1 at most as often as in t_2. Thus we have $S(t'_1) - S(t_1) \leq S(t'_2) - S(t_2)$. Analogously we can conclude $F(t'_1) - F(t_1) \leq F(t'_2) - F(t_2)$. The number of negation operators also only changes by inserting new operators. Because of condition 1. this happens in t_1 at most as often as in t_2. Thus we have $N(t'_1) - N(t_1) \leq N(t'_2) - N(t_2)$, too.
   If 2. holds for t_1 and t_2 because of 2a, then $S(t_1) < S(t_2)$ and $S(t'_1) - S(t_1) \leq S(t'_2) - S(t_2)$ imply $S(t'_1) < S(t'_2)$. Condition 2. holds because of 2a for t'_1 and t'_2, too.
   Let condition 2. hold for t_1 and t_2 because of 2b. Because of $S(t_1) = S(t_2)$ and $S(t'_1) - S(t_1) \leq S(t'_2) - S(t_2)$ we have $S(t'_1) \leq S(t'_2)$. Furthermore we can conclude

$F(t'_1) < F(t'_2)$ from $F(t_1) < F(t_2)$ and $F(t'_1) - F(t_1) \leq F(t'_2) - F(t_2)$. Thus condition 2. hold for t'_1 and t'_2 because of 2a or 2b.
Analogously we can conclude $S(t'_1) \leq S(t'_2)$, $F(t'_1) \leq F(t'_2)$ and $N(t'_1) < N(t'_2)$, if condition 2. holds because of part 2c. The condition therefore holds for t'_1 and t'_2 because of one of the parts 2a, 2b or 2c.
Analogously in case 2d for t_1 and t_2 we have $S(t'_1) \leq S(t'_2)$, $F(t'_1) \leq F(t'_2)$ and $N(t'_1) \leq N(t'_2)$. Since recursive path orderings are invariant under variable assignment, we further know $t'_1 <_{r10} t'_2$ and 2. holds for t'_1 and t'_2.

Altogether we have $t'_1 <_{10} t'_2$ and hence $<_{10}$ is invariant under variable assignment. □

Lemma 4.26 *$<_{10}$ is monotonic..*

Proof: With a function symbol f and terms $r_1, \dots, r_{i-1}, r_{i+1}, \dots, r_n$ we regard the insertions $t'_1 = (f\ r_1 \dots r_{i-1}\ t_1\ r_{i+1} \dots r_n)$ and $t'_2 = (f\ r_1 \dots r_{i-1}\ t_2\ r_{i+1} \dots r_n)$.

1. Since the variables being contained in t_1 and t_2 fulfill condition 1., this condition also holds for t'_1 and t'_2.

2. $<_{r10}$ is not affected by the insertion. The changes of $S(t)$, $F(t)$ and $N(t)$ depend on the function symbol f and the terms $r_1, \dots, r_{i-1}, r_{i+1}, \dots, r_n$. But since these are identical in t'_1 and t'_2 we have:

   - $S(t'_1) - S(t_1) = S(t'_2) - S(t_2)$
   - $F(t'_1) - F(t_1) = F(t'_2) - F(t_2)$
   - $N(t'_1) - N(t_1) = N(t'_2) - N(t_2)$

   Altogether condition 2. is also valid for t'_1 and t'_2.

From $t'_1 <_{10} t'_2$ as shown before we can conclude monotonicity of $<_{10}$. □

Theorem 4.27 *The term rewriting system $TRS_{A_{10}}$ is terminating.*

Proof: We show termination of $TRS_{A_{10}}$ by proving $<_{10}$ to be a termination ordering.
$<_{10}$ is well-founded as shown in lemma 4.24, it does not change under variable assignment as shown in lemma 4.25 and we proved it to be monotonic in lemma 4.26. Therefore, it fulfills all preconditions for a termination ordering.
For every rule $l \to r$ we have $r <_{10} l$ as is obvious by inspecting the single rules. Thus $<_{10}$ is a termination ordering for $TRS_{A_{10}}$ and the term rewriting system is terminating. □

Theorem 4.28 *The term rewriting system $TRS_{A_{10}}$ is confluent.*

Proof: As usual we have local confluence from the observation of common normal forms for all critical pairs and confluence from that and the termination proved in theorem 4.28. $\square$

Theorem 4.29 *Any normal form of $TRS_{A_{10}}$ does not contain constant addends 0, additions of inverse elements or factors 0 or 1. No factor in a product is negated.*

Proof: The theorem follows from the existence of rules eliminating exactly these situations. These rules are not restricted by any further conditions. $\square$

Furthermore calculations with constant numbers are performed wherever possible. Because of a large number of special cases a proof of this fact would need a quite long case distinction.

For a term fully multiplied out as input the normal form is an equivalent term that is fully multiplied out, too, and does not contain any unnecessary operators or operands.
For mathematically equivalent polynomials, the result of simplification of sums is unique modulo commutativity.

4.3 Product simplification

Product simplification essentially means elimination of factors 1 and common factors of numerator and denominator by reduction. This task seems to be quite difficult to perform by term rewriting systems. [Fro98] describes an alternative approach of reducing fractions. This approach is based on calculating the greatest common divisor (gcd) of the numerator and denominator and dividing both polynomials by their gcd. The gcd algorithm presented in [Fro98] expects an input term of the form

$$(*\ n\ (/\ d))$$

where both the numerator n and the denominator d are polynomials that do not contain any further fractions. n and d are fully multiplied out, do not contain any nested sums or products, and no sum occurs as an argument to $-$. From any arithmetic term an equivalent term of this form is generated by performing the normalizations up to TRS_{A_9}. The result of the gcd algorithm is a term of the form

$$(*\ n'\ (/\ d'))$$

where n' and d' do not contain any further fractions and have the greatest common divisor 1. The polynomials n' and d' are not guaranteed to be fully multiplied out. Thus they are normalized with TRS_{A_9} again and with $TRS_{A_{10}}$.

Reducing fractions may cause the semantics of terms to change, if it is done for subterms that may have the value 0 for certain variable assignments. However, the semantics of a term can only be changed in the sense that a term whose value is undefined due to a runtime error gets a defined value by avoiding the runtime error (e.g. $\frac{x-1}{x^2-1} = \frac{x-1}{(x+1)\cdot(x-1)} \rightarrow \frac{1}{x+1}$ avoids the undefined value for $x = 1$.) Since runtime errors are never the intended output of a program one can argue to tolerate this change of semantics.

5 Boolean terms

Boolean terms consist of two types of operators:

1. Predicates (especially comparison operators)
2. Boolean connectors

Therefore, simplification in two steps seems appropriate:

1. Normalizing the use of predicates.
2. Normalizing the formulas built by boolean connectors by using exactly the predicates of the original term.

These subtasks are described in detail in the following subsections.

5.1 Normalizing predicates

It seems impossible to find general ways of normalizing every set of predicates, because there might by different forms of dependencies between certain predicates. We are going to give a method of partial normalization that seems applicable in many cases, especially for the comparison operators. It is based on pairs of inverse predicates:

Let A and B be predicates fulfilling

$$A(x) \Leftrightarrow \neg B(x) \text{ for all } x.$$

Then it is already sufficient to use only one of them. If we e.g. choose to express B by using A we have to replace every occurrence matching the pattern $B(x)$ with $\neg A(x)$.

The comparison operators form the following pairs from which we represent the left one by the right:

$$a \neq b \to a\neg = b$$
$$a > b \to a\neg \leq b$$
$$a \geq b \to a\neg < b$$

Further relations have to be expressed for the given case. The example of the comparison operators contains a relation that can be used to eliminate the predicate $\leq$:

$$a \leq b \to (a < b) \vee (a = b)$$

One has to decide from case to case which relations between the given predicates exist and should be used. The more complete the relations are represented the more powerful the simplification becomes.

Unfortunately, we may not only need to handle logical equivalence but also implication. There seems to be no easy way of expressing this by term rewriting systems.

Since the concrete term rewriting system for this stage depends on the set of predicates to be handled, termination and confluence cannot be proved independently. There are no obvious problems in achieving them, but a proof of these properties has to be given for every fixed case.

5.2 Logical combinators

After reducing the number of different predicates that can occur in a term as far as possible, the structure of the term should be transformed to a normal form. Possible forms are the conjunctive and the disjunctive normal form as known from propositional logic. These share the disadvantage of having a much higher complexity than a minimal representation.
Furthermore, these normal forms are equivalent with respect to logic, but not with respect to termination of functional programs e.g. in Scheme. This is because of the fact that *and* and *or* are macros in Scheme and do not evaluate any arguments not needed to determine the result. Transformation into normal form may introduce further tests that need not terminate.

To find an appropriate normal form the following properties in contrast to usual conjunction and disjunction are of interest:

1. *and* and *or* are associative but not commutative in Scheme.
2. Extending a conjunction with a tautology or a disjunction with an unsatisfiable formula may change the termination behaviour.

3. Neither applying distributivity nor de Morgan's laws change the termination behaviour.

Disjunctive and conjunctive normal form in the usual sense of putting all variables into every operand are not useful according to the statements above.
We will use variants of these normal forms not containing all variables in every operand. These formulae can be obtained by using distributivity and de Morgan's laws only and they are uniquely determined by their semantics in logic and their termination behaviour if any of the predicates may not terminate.

A disjunctive form of this kind is generated by the following term rewriting system TRS_{B_2}. (For conjunctive form one must only exchange *and* and *or* in the last rule)

$$\begin{aligned}
(not\ (not\ x)) &\rightarrow x \\
(not\ (or\ \ {<}a_1 \ldots e_1{>})) &\rightarrow (and\ \ {<}(not\ a_1) \ldots (not\ e_1){>}) \\
(not\ (and\ \ {<}a_1 \ldots e_1{>})) &\rightarrow (or\ \ {<}(not\ a_1) \ldots (not\ e_1){>}) \\
(and\ \ {<}1 \ldots{>}\ \ (and\ \ {<}2 \ldots{>})\ \ {<}3 \ldots{>}) &\rightarrow (and\ \ {<}1 \ldots{>}\,{<}2 \ldots{>}\,{<}3 \ldots{>}) \\
(or\ \ {<}1 \ldots{>}\ \ (or\ \ {<}2 \ldots{>})\ \ {<}3 \ldots{>}) &\rightarrow (or\ \ {<}1 \ldots{>}\,{<}2 \ldots{>}\,{<}3 \ldots{>})
\end{aligned}$$

$$\begin{aligned}
&(and\ \ {<}1 \ldots{>}\ \ (or\ \ {<}a_2 \ldots e_2{>})\ \ {<}3 \ldots{>}) \rightarrow \\
&\qquad (or\ \ {<}(and\ \ {<}1 \ldots{>}\ \ a_2\ \ {<}3 \ldots{>}) \ldots (and\ \ {<}1 \ldots{>}\ \ e_2\ \ {<}3 \ldots{>}){>}
\end{aligned}$$

Theorem 5.1 *The term rewriting system TRS_{B_2} is terminating.*

Proof: The theorem follows directly from a recursive path ordering with the order *not* > *and* > *or* on the set of function symbols. □

Theorem 5.2 *The term rewriting system TRS_{B_2} is confluent.*

Proof: Local confluence is obvious from the critical pairs having common normal forms. With termination from theorem 5.1 confluence follows. □

Theorem 5.3 *The result of TRS_{B_2} is a boolean term consisting of a single disjunction with conjunctions of literals as elements.*

Proof: Subterms with top level operator *and* as elements to an operator *or* are the only possible way of nesting *and* and *or* that is not decomposed by the rules 3, 4 and 5. By rules 1 and 2 the scope of negation is restricted as much as possible and thus there is no way of nesting *and* and *or* even indirectly other than stated in the theorem. □

6 Processing combined terms

6.1 Problem description

In addition to normalizing terms of a given type it is necessary to normalize complex terms as well. The problem occurring here is the change of the type of some subterm under normalization. When e.g. normalizing the list term

$$(car\ (cdr\ (cons\ 1\ (cons\ 2\ (cons\ 3\ ())))))$$

the result is the arithmetic constant 2. For a complex term containing the expression above as subterm of an arithmetic expression, the result may depend on the order in which subterms of the different types are normalized: If we normalize list terms first, we can possibly use the constant 2 in further normalization. This is impossible if we try arithmetic normalization first.

The following methods yield completely simplified terms:

1. The different term rewriting systems are applied iteratedly in an arbitrary order until the normalized term does not change for the first time.
2. The term is normalized by innermost rewriting, choosing the right term rewriting system or set of term rewriting systems according to the subterm's top level function symbol.
3. By checking the possible type transformations of a term there may be an order on the term rewriting systems that guarantees every term to be free of subterms normalizable to the actual type on normalization time.
4. Those term rewriting systems with different results depending on the order have to be checked for the possibility of uniting the rule sets to a single term rewriting system.

The approaches 1. and 2. cause certain problems: 1. is quite inefficient and thus hardly useful in practical use. On the other hand 2. requires control over the choose of the actual term and the rule set to use. There is no way to realize this using a normal interpreter for term rewriting systems.
3. seems to be the best approach, since it yields a sequence of term rewriting systems each having a small number of rules that must be applied only once and guarantees full normalization. Unfortunately we will show in subsection 6.2 that the given set of term rewriting systems cannot be ordered in that way.
Thus we will unite some of the rule sets and show some properties in subsection 6.3.

6.2 Choosing a partial order

The best way of handling complex terms is by applying the given rule sets for the different types in a certain order that guarantees the *complete* normalization of any term. This partial order is determined by:

$a < b \Longleftrightarrow$ terms of type a can be reduced to terms of type b.

The example given at the beginning of this section shows that list terms can be transformed to terms of any other type. This implies:

- `list` < `arith`
- `list` < `bool`

Arithmetic terms are not critical: An arithmetic expression only contains subexpressions returning a number. Normalizing an arithmetic expression therefore always results in an arithmetic expression again, provided that the expression does not cause any runtime errors.

Boolean expressions can be interpreted in different ways: Scheme does not only distinguish between the two values *true* and *false* as usual, but any value except `'()` representing *false* represents *true*. The output of the boolean functions is adapted to that:

```
1 ]=> (and (> 2 1) (list 1 2 3))
;Value 1: (1 2 3)
```

In this interpretation a type change from a boolean value to any other type is possible. Especially boolean expressions may be transformed to list terms as in the example above. Altogether the restrictions of the partial order contain the following:

- `bool` < `list`
- `list` < `bool`

The wanted partial order cannot exist, if normalizing boolean values may result in values of other types.

If we restrict boolean values to *true* and *false* there is no problem because a type change from boolean to any other type is impossible. We do not consider this restriction any further, because we do not want to restrict the set of valid Scheme programs.

6.3 Unions of rule sets

This subsection describes which of the rule sets presented before must be united to guarantee full normalization of any term in one application of the rule sets. Termination and confluence are proved for the combined rule set.

From subsection 6.2 we know, that arithmetic terms cannot change their type during normalization. We can therefore put arithmetic normalization to the end of the normalization task.

The only conflicts are between the rule set for lists and those for boolean terms, especially for boolean connectors. This implies the following way of handling complex terms:

1. Normalizing boolean predicates (by TRS_{B_1}).
2. Combined normalization of list terms and boolean connectors (by the direct sum TRS_2 of TRS_L and TRS_{B_2}).
3. Normalization of arithmetic terms (by the multi level trs from Sec. 4).

All term rewriting systems except of the direct sum $TRS_2 = TRS_L \oplus TRS_{B_2}$ used in 2 are proved to be terminating and confluent. Not considering the extended constructs, confluence follows for TRS_2 as the direct sum of confluent trs (see [Toy87]). Termination in the given case follows from the fact that TRS_L and TRS_{B_2} are left linear (i.e. no variable occurs repeatedly at the left hand side of a rule) and complete (i.e. confluent and terminating) (see [TKB95]).[6]

The cited properties also hold for term rewriting systems containing the constructs $<i \ldots>$, $< a_j \ldots e_j >$ and $<< [\mathit{notarity}\ (i\ k)] >>$, because the cited proofs do not depend on the finiteness of the rule sets. Since every of the extended rules represents a countable set of non-extended rules, TRS_L and TRS_{B_2} represent countable sets of rules to which the cited results apply. Thus we have:

Corollary 6.1 *The term rewriting system $TRS_2 = TRS_L \oplus TRS_{B_2}$ is confluent and terminating.*

6.4 Putting it all together

This section gives a short description of the local normalization system TRS that results from integrating all the stages described before and whose implementation is described in [Fro98].

[6] The alternative conditions proved sufficient for termination of the direct sum of trs in [Rus87] and [Mid89] do not hold in the actual case.

Normalizing an arbitrary term consists of the following stages (The term rewriting systems can all be found in Appendix A):

1. Applying the term rewriting systems TRS_{B_1}, TRS_2, TRS_{A_1}, TRS_{A_2}, TRS_{A_3}, TRS_{A_4}, TRS_{A_5}, TRS_{A_6}, TRS_{A_7}, TRS_{A_8} and TRS_{A_9} in the given order.
2. Applying the gcd algorithm mentioned in Sec. 4.3 and described in detail in [Fro98].
3. Applying the term rewriting systems TRS_{A_9} and $TRS_{A_{10}}$.

With this definition of TRS we have the following properties.

Theorem 6.2 *The system TRS is terminating.*

Proof: All components of TRS are terminating and there are no additional loops in the compound algorithm that could cause non-termination. □

Theorem 6.3 *The result t' of normalizing a term t with the system TRS fulfills the following properties:*

1. *t' is computationally equivalent to t except of avoiding runtime errors because of division by* 0 *in some cases (see Sec. 4.3).*
2. *t' does not contain any list selector abbreviations (e.g.* caar, cdadr*), but these are transformed to nested occurrences of* car *and* cdr.
3. *In t' there are no applications of* car *or* cdr *to a term whose top level constructor is* cons. *These applications are replaced by their result.*
4. *The only boolean terms in t' are in disjunctive form, i.e.* not *is not applied to a conjunction or disjunction but only to the elements,* and *and* or *do not occur nested and no disjunctive term occurs as an argument of* and.
5. *Arithmetic terms in t' consist of a single fraction with no nested fractions. The greatest common divisor of the polynomials in the numerator and denominator is* 1 *and both polynomials are fully multiplied out and do not contain any sums of inverse subterms, sums with addend* 0 *or products with factor* 0 *or* 1. *Furthermore there ares no sums or products containing more than one constant element.*

Proof: All the properties of the resulting normal form t' follow from the properties of the individual normalization stages. □

Theorem 6.4 *The system TRS is confluent modulo commutativity of $+$ and $*$.*

Proof: The normalization stages for lists and boolean terms are confluent. The representation of every arithmetic term with the properties of Theorem 6.3 is unique modulo commutativity. Since the terms resulting from a normalization with TRS are equivalent to the input term (except of enlarging the definition set) the reaching of a unique representation implies confluence. $\square$

7 Conclusion

In this paper we have presented an approach to normalize parts of functional programs, especially lists, arithmetic terms and boolean terms that often occur in real programs. The normal forms of equivalent terms are equal except of commutativity, variable renaming and not represented relations between predicates. When using the system before program analysis, the analysis procedures can become much smaller since they can rely on special properties of the normalized program, especially that a number of special cases cannot occur. On the other hand in program analysis it may be necessary to check different terms for equivalence. This question is reduced to the question of syntactic equality, where commutativity and the identification of two variable sets is easy to handle.

Furthermore we have introduced a number of extensions to term rewriting. These are useful especially but not only in handling programs in LISP. The practical use of these extensions is shown by a prototype term rewriting interpreter [Fro98] that can handle all these extensions.

References

[Bau85] G. Bauer. n-Level rewriting systems. *Theoretical Computer Science*, 40(2–3):85–99, October 1985.

[DJ90] 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.

[DO90] Nachum Dershowitz and Mitsuhiro Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75(1–2):111–138, 25 September 1990.

[Fro98] Dieter Froning. Lokale Normalisierung von Scheme-Programmen. Diplomarbeit, FernUniversität Hagen, April 1998.

[Jou95] J. P. Jouannaud. Rewrite proofs and computations. International Summer School Marktoberdorf “Logic of computation”, Institut für Informatik, Technische Universität München, 1995.

[KB67] 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.

[Mid89] Aart Middeldorp. A sufficient condition for the termination of the direct sum of term rewriting systems. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 396–401, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[Rus87] Michael Rusinowitch. On termination of the direct sum of term-rewriting systems. *Information Processing Letters*, 26(2):65–70, 19 October 1987.

[TKB95] Y. Toyama, Jan Willem Klop, and H. P. Barendregt. Termination for direct sums of left-linear complete term rewriting systems. *Journal of the ACM*, 42(6):1275–1304, November 1995.

[Toy87] Yoshihito Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1):128–143, January 1987.

A The complete normalizing system

The normalizing system presented in this paper is listed here in the final order for an overview:

1. The term rewriting system TRS_{B_1} has to be implemented depending on the concrete set of predicates.

2. TRS_2:

$$
\begin{aligned}
(caar\ x) &\rightarrow (car\ (car\ x))\\
(cadr\ x) &\rightarrow (car\ (cdr\ x))\\
(cdar\ x) &\rightarrow (cdr\ (car\ x))\\
(cddr\ x) &\rightarrow (cdr\ (cdr\ x))\\
(caaar\ x) &\rightarrow (car\ (car\ (car\ x)))\\
(caadr\ x) &\rightarrow (car\ (car\ (cdr\ x)))\\
(cadar\ x) &\rightarrow (car\ (cdr\ (car\ x)))\\
(caddr\ x) &\rightarrow (car\ (cdr\ (cdr\ x)))
\end{aligned}
$$

$$\begin{aligned}
(\mathit{cdaar}\ x) &\to (\mathit{cdr}\ (\mathit{car}\ (\mathit{car}\ x))) \\
(\mathit{cdadr}\ x) &\to (\mathit{cdr}\ (\mathit{car}\ (\mathit{cdr}\ x))) \\
(\mathit{cddar}\ x) &\to (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{car}\ x))) \\
(\mathit{cdddr}\ x) &\to (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{cdr}\ x))) \\
(\mathit{caaaar}\ x) &\to (\mathit{car}\ (\mathit{car}\ (\mathit{car}\ (\mathit{car}\ x)))) \\
(\mathit{caaadr}\ x) &\to (\mathit{car}\ (\mathit{car}\ (\mathit{car}\ (\mathit{cdr}\ x)))) \\
(\mathit{caadar}\ x) &\to (\mathit{car}\ (\mathit{car}\ (\mathit{cdr}\ (\mathit{car}\ x)))) \\
(\mathit{caaddr}\ x) &\to (\mathit{car}\ (\mathit{car}\ (\mathit{cdr}\ (\mathit{cdr}\ x)))) \\
(\mathit{cadaar}\ x) &\to (\mathit{car}\ (\mathit{cdr}\ (\mathit{car}\ (\mathit{car}\ x)))) \\
(\mathit{cadadr}\ x) &\to (\mathit{car}\ (\mathit{cdr}\ (\mathit{car}\ (\mathit{cdr}\ x)))) \\
(\mathit{caddar}\ x) &\to (\mathit{car}\ (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{car}\ x)))) \\
(\mathit{cadddr}\ x) &\to (\mathit{car}\ (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{cdr}\ x)))) \\
(\mathit{cdaaar}\ x) &\to (\mathit{cdr}\ (\mathit{car}\ (\mathit{car}\ (\mathit{car}\ x)))) \\
(\mathit{cdaadr}\ x) &\to (\mathit{cdr}\ (\mathit{car}\ (\mathit{car}\ (\mathit{cdr}\ x)))) \\
(\mathit{cdadar}\ x) &\to (\mathit{cdr}\ (\mathit{car}\ (\mathit{cdr}\ (\mathit{car}\ x)))) \\
(\mathit{cdaddr}\ x) &\to (\mathit{cdr}\ (\mathit{car}\ (\mathit{cdr}\ (\mathit{cdr}\ x)))) \\
(\mathit{cddaar}\ x) &\to (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{car}\ (\mathit{car}\ x)))) \\
(\mathit{cddadr}\ x) &\to (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{car}\ (\mathit{cdr}\ x)))) \\
(\mathit{cdddar}\ x) &\to (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{car}\ x)))) \\
(\mathit{cddddr}\ x) &\to (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{cdr}\ (\mathit{cdr}\ x)))) \\
(\mathit{car}\ (\mathit{cons}\ x\ y)) &\to x \\
(\mathit{cdr}\ (\mathit{cons}\ x\ y)) &\to y \\
(\mathit{list}) &\to () \\
(\mathit{list}\ x) &\to (\mathit{cons}\ x\ ())
\end{aligned}$$

$$(\mathit{list}\ x\ \ <1\ldots>) \quad <<[\mathit{notarity}\ (1\ 0)]>> \quad \to (\mathit{cons}\ x\ (\mathit{list}\ \ <1\ldots>))$$

$$\begin{aligned}
(\mathit{not}\ (\mathit{not}\ x)) &\to x \\
(\mathit{not}\ (\mathit{or}\ \ <a_1\ldots e_1>)) &\to (\mathit{and}\ \ <(\mathit{not}\ a_1)\ldots(\mathit{not}\ e_1)>) \\
(\mathit{not}\ (\mathit{and}\ \ <a_1\ldots e_1>)) &\to (\mathit{or}\ \ <(\mathit{not}\ a_1)\ldots(\mathit{not}\ e_1)>) \\
(\mathit{and}\ \ <1\ldots>\ \ (\mathit{and}\ \ <2\ldots>)\ \ <3\ldots>) &\to (\mathit{and}\ \ <1\ldots><2\ldots><3\ldots>) \\
(\mathit{or}\ \ <1\ldots>\ \ (\mathit{or}\ \ <2\ldots>)\ \ <3\ldots>) &\to (\mathit{or}\ \ <1\ldots><2\ldots><3\ldots>)
\end{aligned}$$

$$\begin{aligned}
&(\mathit{and}\ \ <1\ldots>\ \ (\mathit{or}\ \ <a_2\ldots e_2>)\ \ <3\ldots>) \to \\
&\qquad (\mathit{or}\ \ <(\mathit{and}\ \ <1\ldots>\ \ a_2\ \ <3\ldots>)\ldots(\mathit{and}\ \ <1\ldots>\ \ e_2\ \ <3\ldots>)>
\end{aligned}$$

3. TRS_{A_1}:

$$\begin{aligned}
(1+\ x) &\rightarrow (+\ 1\ x)\\
(-1+\ x) &\rightarrow (+\ -1\ x)\\
(-\ a\ \ <1\ldots>)\ \ \ll[\mathit{notarity}\ (1\ 0)]\gg\ \ &\rightarrow (+\ a\ (-\ (+\ \ <1\ldots>)))\\
(/\ a\ \ <1\ldots>)\ \ \ll[\mathit{notarity}\ (1\ 0)]\gg\ \ &\rightarrow (*\ a\ (/\ (*\ \ <1\ldots>)))
\end{aligned}$$

4. TRS_{A_2}:

$$\begin{aligned}
(/\ (/\ x)) &\rightarrow x\\
(-\ (-\ x)) &\rightarrow x\\
(*\ \ <1\ldots>\ \ (*\ \ <2\ldots>)\ \ <3\ldots>) &\rightarrow (*\ \ <1\ldots><2\ldots><3\ldots>)\\
(+\ \ <1\ldots>\ \ (+\ \ <2\ldots>)\ \ <3\ldots>) &\rightarrow (+\ \ <1\ldots><2\ldots><3\ldots>)\\
(/\ (*\ \ <a_1\ldots e_1>)) &\rightarrow (*\ \ <(/\ a_1)\ldots(/\ e_1)>)\\
(-\ (+\ \ <a_1\ldots e_1>)) &\rightarrow (+\ \ <(-\ a_1)\ldots(-\ e_1)>)
\end{aligned}$$

$$\begin{aligned}
&(*\ \ <1\ldots>\ \ (+\ \ <a_2\ldots e_2>)\ \ <3\ldots>) \rightarrow\\
&\qquad\qquad (+\ \ <(*\ \ <1\ldots>\ \ a_2\ \ <3\ldots>)\ldots(*\ \ <1\ldots>\ \ e_2\ \ <3\ldots>)>)
\end{aligned}$$

$$\begin{aligned}
&(/\ (+\ \ <a_1\ldots e_1>\ \ (/\ y)\ \ <a_2\ldots e_2>)) \rightarrow\\
&\qquad\qquad (*\ y\ (/\ (+\ \ <(*\ y\ a_1)\ldots(*\ y\ e_1)>\ \ 1\ \ <(*\ y\ a_2)\ldots(*\ y\ e_2)>)))
\end{aligned}$$

$$\begin{aligned}
&(/\ (+\ \ <a_1\ldots e_1>\ \ (*\ \ <2\ldots>\ \ (/\ y)\ \ <3\ldots>)\ \ <a_4\ldots e_4>)) \rightarrow\\
&(*\ y\ (/\ (+\ \ <(*\ y\ a_1)\ldots(*\ y\ e_1)>\ \ (*\ \ <2\ldots><3\ldots>)\ \ <(*\ y\ a_4)\ldots(*\ y\ e_4)>))
\end{aligned}$$

5. TRS_{A_3}:

$$\begin{aligned}
(-\ 0) &\rightarrow 0\\
(-\ (-\ x)) &\rightarrow x\\
(+\ \ <1\ldots>\ \ 0\ \ <2\ldots>) &\rightarrow (+\ \ <1\ldots><2\ldots>)\\
(+\ \ <1\ldots>\ \ x\ \ <2\ldots>\ \ (-\ x)\ \ <3\ldots>) &\rightarrow (+\ \ <1\ldots><2\ldots><3\ldots>)\\
(+\ \ <1\ldots>\ \ (-\ x)\ \ <2\ldots>\ \ x\ \ <3\ldots>) &\rightarrow (+\ \ <1\ldots><2\ldots><3\ldots>)\\
(+) &\rightarrow 0\\
(+\ x) &\rightarrow x\\
(-\ (+\ \ <a_1\ldots e_1>)) &\rightarrow (+\ \ <(-\ a_1)\ldots(-\ e_1)>)\\
(+\ \ <1\ldots>\ \ (+\ \ <2\ldots>)\ \ <3\ldots>) &\rightarrow (+\ \ <1\ldots><2\ldots><3\ldots>)
\end{aligned}$$

$$(* \ {<}1\ldots{>} \ (+ \ {<}a_2\ldots e_2{>}) \ {<}3\ldots{>}) \rightarrow (+ \ {<}(* \ {<}1\ldots{>} \ a_2 \ {<}3\ldots{>})\ldots(* \ {<}1\ldots{>} \ e_2 \ {<}3\ldots{>}){>})$$

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

6. TRS_{A_4}:

$$(+ \ {<}1\ldots{>} \ (/ \ y) \ {<}2\ldots{>}) \rightarrow (+ \ {<}1\ldots{>} \ (* \ (/ \ y)) \ {<}2\ldots{>})$$
$$(/ \ x) \quad {<}{<}[exclude \ (* \ {<}1\ldots{>}) \ x]{>}{>} \quad \rightarrow (/ \ (* \ x))$$

$$(+ \ {<}1\ldots{>} \ (* \ {<}2\ldots{>} \ (/ \ ({<}comm *{>} \ {<}choose \ {<}3\ldots{>}{>} \ {<}4\ldots{>})) \ {<}5\ldots{>}) \ {<}6\ldots{>} \ (* \ {<}7\ldots{>} \ (/ \ ({<}comm *{>} \ {<}choose \ {<}3\ldots{>}{>} \ {<}8\ldots{>})) \ {<}9\ldots{>}) \ {<}10\ldots{>}) \quad {<}{<}[notarity \ (4 \ 0) \ (8 \ 0)]{>}{>} \ \rightarrow (+ \ {<}1\ldots{>} \ (* \ {<}2\ldots{>} \ {<}8\ldots{>} \ (/ \ (* \ {<}3\ldots{>} \ {<}4\ldots{>} \ {<}8\ldots{>})) \ {<}5\ldots{>}) \ {<}6\ldots{>} \ (* \ {<}7\ldots{>} \ {<}4\ldots{>} \ (/ \ (* \ {<}3\ldots{>} \ {<}4\ldots{>} \ {<}8\ldots{>})) \ {<}9\ldots{>}) \ {<}10\ldots{>})$$

7. TRS_{A_5}:

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

$$(* \ {<}1\ldots{>} \ (+ \ {<}a_2\ldots e_2{>}) \ {<}3\ldots{>}) \rightarrow (+ \ {<}(* \ {<}1\ldots{>} \ a_2 \ {<}3\ldots{>})\ldots(* \ {<}1\ldots{>} \ e_2 \ {<}3\ldots{>}){>})$$

8. TRS_{A_6}:

$$(+ \ {<}1\ldots{>} \ (/ \ x) \ {<}2\ldots{>}) \rightarrow (+ \ {<}1\ldots{>} \ (* \ (/ \ x) \ 1) \ {<}2\ldots{>})$$
$$(* \ (/ \ x)) \rightarrow (* \ (/ \ x) \ 1)$$

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

9. TRS_{A_7}:

$$\begin{aligned} (*\ x\ (/\ (-\ y)))\ \ <<[exclude\ (/\ t)\ x]>>\ &\rightarrow (*\ (-\ x)\ (/\ y)) \\ (*\ (/\ x)\ y)\ \ <<[exclude\ (/\ t)\ y]>>\ &\rightarrow (*\ y\ (/\ x)) \\ (-\ (-\ x)) &\rightarrow x \end{aligned}$$

10. TRS_{A_8}:

$$\begin{aligned} &(<comm\ +>\ <choose\ \ <(*\ a_1\ (/\ x))\ldots(*\ e_1\ (/\ x))>>\ <a_2\ldots e_2>) \\ &\qquad <<[notarity\ (1\ 0)]>> \rightarrow (*\ (+\ \ <a_1\ldots e_1>\ <(*\ x\ a_2)\ldots(*\ x\ e_2)>)\ (/\ x)) \end{aligned}$$

11. TRS_{A_9}:

$$\begin{aligned} (-\ (+\ \ <a_1\ldots e_1>)) &\rightarrow (+\ \ <(-\ a_1)\ldots(-\ e_1)>) \\ (+\ \ <1\ldots>\ \ (+\ \ <2\ldots>)\ \ <3\ldots>) &\rightarrow (+\ \ <1\ldots>\ <2\ldots>\ <3\ldots>) \\ (*\ \ <1\ldots>\ \ (*\ \ <2\ldots>)\ \ <3\ldots>) &\rightarrow (*\ \ <1\ldots>\ <2\ldots>\ <3\ldots>) \end{aligned}$$

$$\begin{aligned} &(*\ \ <1\ldots>\ \ (+\ \ <a_2\ldots e_2>)\ \ <3\ldots>) \rightarrow \\ &\qquad\qquad (+\ \ <(*\ \ <1\ldots>\ \ a_2\ \ <3\ldots>)\ldots(*\ \ <1\ldots>\ \ e_2\ \ <3\ldots>)>) \end{aligned}$$

12. $TRS_{A_{10}}$:

$$\begin{aligned} (-\ 0) &\rightarrow 0 \\ (-\ (-\ x)) &\rightarrow x \\ (+\ \ <1\ldots>\ \ 0\ \ <2\ldots>) &\rightarrow (+\ \ <1\ldots>\ <2\ldots>) \\ (+\ \ <1\ldots>\ \ x\ \ <2\ldots>\ \ (-\ x)\ \ <3\ldots>) &\rightarrow (+\ \ <1\ldots>\ <2\ldots>\ <3\ldots>) \\ (+\ \ <1\ldots>\ \ (-\ x)\ \ <2\ldots>\ \ x\ \ <3\ldots>) &\rightarrow (+\ \ <1\ldots>\ <2\ldots>\ <3\ldots>) \\ (*\ \ <1\ldots>\ \ 1\ \ <2\ldots>) &\rightarrow (*\ \ <1\ldots>\ <2\ldots>) \\ (*\ \ <1\ldots>\ \ -1\ \ <2\ldots>) &\rightarrow (-\ (*\ \ <1\ldots>\ <2\ldots>)) \\ (+\ x) &\rightarrow x \\ (+) &\rightarrow 0 \\ (*\ x) &\rightarrow x \\ (*) &\rightarrow 1 \\ (*\ \ <1\ldots>\ \ 0\ \ <2\ldots>) &\rightarrow 0 \\ (*\ \ <1\ldots>\ \ (-\ x)\ \ <2\ldots>) &\rightarrow (-\ (*\ \ <1\ldots>\ \ x\ \ <2\ldots>)) \\ (+\ \ <1\ldots>\ \ x\ \ <2\ldots>\ \ x\ \ <3\ldots>) &\rightarrow (+\ \ <1\ldots>\ \ (*\ 2\ x)\ \ <2\ldots>\ <3\ldots>) \\ (-\ \ <test\ number?\ x>) &\rightarrow <eval\ (-\ x)> \end{aligned}$$

$$(/ \ < test\ (lambda\ (v)\ (and\ (number\ v)\ (not\ (= \ v\ 0))))\ x >) \rightarrow < eval\ (/\ x) >$$

$$(-\ (*\ \ <1\ldots>\ <test\ number?\ x>\ <2\ldots>) \rightarrow \\ (*\ \ <1\ldots>\ <eval\ (-\ x)>\ <2\ldots>)$$

$$(+\ \ <1\ldots>\ <test\ number?\ x>\ <2\ldots>\ <test\ number?\ y>\ <3\ldots>) \rightarrow \\ (+\ \ <1\ldots>\ <eval\ (+\ x\ y)>\ <2\ldots>\ <3\ldots>)$$

$$(*\ \ <1\ldots>\ <test\ number?\ x>\ <2\ldots>\ <test\ number?\ y>\ <3\ldots>) \rightarrow \\ (*\ \ <1\ldots>\ <eval\ (*\ x\ y)>\ <2\ldots>\ <3\ldots>)$$

$$(+\ \ <1\ldots>\ \ (<comm\ *>\ <choose\ \ <2\ldots>>)\ \ <3\ldots> \\ (<comm\ *>\ <choose\ \ <2\ldots>>)\ \ <4\ldots>) \rightarrow \\ (+\ \ <1\ldots>\ \ (*\ 2\ \ <2\ldots>)\ \ <3\ldots>\ <4\ldots>)$$

$$(+\ \ <1\ldots>\ \ (<comm\ *>\ <choose\ \ <2\ldots>>\ <test\ number?\ x>)\ \ <3\ldots> \\ (<comm\ *>\ <choose\ \ <2\ldots>>\ <test\ number?\ y>)\ \ <4\ldots>) \rightarrow \\ (+\ \ <1\ldots>\ \ (*\ \ <eval\ (+\ x\ y)>\ <2\ldots>)\ \ <3\ldots>\ <4\ldots>)$$

$$(+ <1\ldots>\ \ (<comm\ *>\ <choose\ \ <2\ldots>>)\ \ <3\ldots> \\ (<comm\ *>\ <choose\ \ <2\ldots>>\ <testnumber?\ x>)\ \ <4\ldots>) \rightarrow \\ (+\ \ <1\ldots>\ \ (*\ \ <eval\ (1+\ x)\ \ <2\ldots>)\ \ <3\ldots>\ <4\ldots>)$$

$$(+ <1\ldots>\ \ (<comm\ *>\ <choose\ \ <2\ldots>>\ <testnumber?\ x>)\ \ <3\ldots> \\ (<comm\ *>\ <choose\ \ <2\ldots>>)\ \ <4\ldots>) \rightarrow \\ (+\ \ <1\ldots>\ \ (*\ \ <eval\ (1+\ x)\ \ <2\ldots>)\ \ <3\ldots>\ <4\ldots>)$$

$$(+\ \ <1\ldots>\ \ (-\ (<comm\ *>\ <choose\ \ <2\ldots>>))\ \ <3\ldots> \\ (<comm\ *>\ <choose\ \ <2\ldots>>\ <test\ number?\ x>)\ \ <4\ldots>) \rightarrow \\ (+\ \ <1\ldots>\ \ (*\ \ <eval\ (-1+\ x)\ \ <2\ldots>)\ \ <3\ldots>\ <4\ldots>)$$

$$
\begin{aligned}
&(+\ <1\ldots>\ (<comm\ *><choose\ \ <2\ldots>><test\ number?\ x>)\ \ <3\ldots> \\
&\qquad\qquad(-\ (<comm\ *><choose\ \ <2\ldots>>))\ \ <4\ldots>)\rightarrow \\
&\qquad\qquad\qquad(+\ \ <1\ldots>\ \ (*\ \ <eval\ (-1+\ x)\ \ <2\ldots>)\ \ <3\ldots><4\ldots>)
\end{aligned}
$$

$$
\begin{aligned}
&(+\ <1\ldots>\ (-\ (<comm\ *><choose\ \ <2\ldots>>))\ \ <3\ldots> \\
&\qquad\qquad(<comm\ *><choose\ \ <2\ldots>>)\ \ <4\ldots>)\rightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad(+\ \ <1\ldots><3\ldots><4\ldots>)
\end{aligned}
$$

$$
\begin{aligned}
&(+\ <1\ldots>\ (-\ (<comm*><choose\ \ <2\ldots>>))\ \ <3\ldots> \\
&\qquad\qquad(-\ (<comm*><choose\ \ <2\ldots>>))\ \ <4\ldots>)\rightarrow \\
&\qquad\qquad\qquad\qquad(+\ \ <1\ldots>\ \ (*\ -2\ \ <2\ldots>)\ \ <3\ldots><4\ldots>)
\end{aligned}
$$