Theoretical Computer Science 112 (1993) 145–185
Elsevier

Solving divergence in Knuth–Bendix completion by enriching signatures

Muffy Thomas and Phil Watson*

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK

Abstract

Thomas, M. and P. Watson, Solving divergence in Knuth–Bendix completion by enriching signatures, Theoretical Computer Science 112 (1993) 145–185.

The Knuth–Bendix completion algorithm is a procedure which generates confluent and terminating sets of rewrite rules. The algorithm has many applications: the resulting rules can be used as a decision procedure for equality or, in the case of program synthesis, as a program. We present an effective algorithm to solve some cases of divergence in the Knuth–Bendix completion algorithm, starting from a grammar characterising the infinite rule set. We replace an infinite set of rewrite rules by a finite complete set by enriching the original (order-sorted) signature with new sorts and new operator arities, while remaining within a conservative extension of the original system and within the original term set. The complexity of the new rewriting system is no worse than that of the original system. We characterise the class of examples for which this approach is applicable and give some sufficient conditions for the algorithm to succeed.

Contents

Correspondence to: M. Thomas, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK. Email: muffy@dcs.glasgow.ac.uk.

* Funded by SERC grant gr/f35371/4/1/1477, Verification Techniques for LOTOS.

0. Introduction

Algebraic specification of abstract data types is a methodology for specifying the behaviour of software systems. A specification consists of a signature and a set of axioms which generate a logical theory; a model of that specification is an algebra with the same signature which satisfies the theory. In equational specification, the axioms are (universally quantified) equations and the theory generated is an equational theory.

Equational reasoning is the process of deriving the consequences of a given system of equations. The simplest way to produce an equational proof that two terms are equal is to keep rewriting subterms of one, using the equations, until it is transformed into the other. The process is more efficient if the equations are considered as *rewrite rules* (rules which represent directed equality) and are used to rewrite both terms. This paradigm is very similar to functional programming; however, in general, rewriting is nondeterministic in the sense that no restrictions are placed on the selection of rules to be applied or on the selection of the subterm to be rewritten. Moreover, there is no restriction on overlapping rules.

Two important properties of a rewriting system are confluence and termination. The *confluence* property ensures that the order of application of rewrite rules is irrelevant, whereas the *termination* property ensures that all sequences of rewrites are well-founded (there are no infinite sequences).

When a set of rewrite rules is confluent and terminating, then each term rewrites to a unique *normal form*: a term which cannot be rewritten. A set of rewrite rules which is confluent and terminating is called *complete*, or *canonical*, and makes equality between terms decidable since repeated application of the rules reduces any term to a unique normal form; in this case two terms are equal if and

only if they have the same normal form. The Knuth–Bendix completion algorithm [18], given a termination ordering, not only tests for the confluence property but it also generates additional rules, in case the given set is not confluent. It is called a "completion" algorithm because if it converges, it generates a complete set of rules which can then be used as a decision procedure for equality. Another application of the completion algorithm is in the synthesis of programs from specifications: the completion algorithm may be used as an "inference engine" to generate the program [7, 8].

Unfortunately, the Knuth–Bendix completion algorithm is not guaranteed to converge, even when the word problem defined by the given system of equations is decidable. When the completion algorithm diverges and results in an infinite confluent sequence of rewrite rules, then we only have a semi-decision procedure for the word problem and in the case of program synthesis, an infinite program. There are several different approaches to solving the problem of divergence and we will discuss them in Section 8.

We aim to replace such an infinite sequence of rules with a finite sequence, or set, which is equivalent in the following sense. First, the finite set should preserve the equational theory defined by the given equations, i.e. the finite set should at least be a *conservative extension* [29] of the infinite sequence. (This is more formally expressed by Theorem 6.11 in Section 6.) Note, however, that the finite set may be based on a larger signature than the infinite sequence; the rules in the former may use some sorts which do not occur in the latter. Second, the finite set should be canonical, i.e. confluent and terminating.

Our approach is based on finding exact generalisations [29] of the varying parts of the infinite sequence of rules. Often, exact generalisations cannot be found with respect to the given signature, but they may exist if we enrich the signature. In [23, 24] the signature is enriched with new operators; here, we enrich the signature with new sorts, sort inclusions and operator arities: the result is an order-sorted signature. The new sorts allow us to capture exactly the varying parts of the rules; and since we avoid adding new operators, the (ground) term set is unchanged. Moreover, since our approach is more liberal than that of exact generalisations, we are able to solve the divergence problem in more cases.

The paper is organised as follows. In Section 1 we review the basic definitions of term rewriting. In Section 2 we present an example of the kind of problem to be solved, and an informal solution. Sections 3 and 4 contain background material concerning tree languages and grammars. Section 5 contains our algorithm which takes as input a signature and a regular tree grammar G (with start symbol S) describing the language of the varying parts of the infinite canonical sequence of rewrite rules. It produces an order-sorted signature with a distinguished sort $\mathscr{S}$ such that the term set of $\mathscr{S}$ contains exactly all instances of words in the language of varying parts, and a rule generalising the sequence. The correctness of the algorithm is demonstrated in Section 6. In Section 7 we apply the algorithm to several examples. In Section 8 we compare other approaches with ours and in Section 9 we discuss directions for further research.

1. Order-sorted term rewriting

A *signature* Σ is a triple $\langle S, <, F \rangle$, where

- S is a set of *sorts*,
- $<$ is a partial order on S (the *subsort ordering*),
- F is a set of *operator arities* of the form $f: s_1 \dots s_n \rightarrow t$, where $s_1, \dots, s_n, t$ are sorts in S and $n \geqslant 0$.

By abuse of notation, $<$ is identified with its own transitive closure. The reflexive closure of $<$ is denoted $\leqslant$.

Operators of the form $f: s_1 \dots s_n \rightarrow t$, where $n=0$, are written $f:t$ and are called *constants* (of sort t).

Terms are constructed from operators in the following way. A constant of sort t is a term (of sort t). If $f: s_1 \dots s_n \rightarrow t$ is an arity in F and $u_1, \dots, u_n$ are terms of sorts $s_1, \dots, s_n$, respectively, then $f(u_1, \dots, u_n)$ is a term (of sort t). Nothing else is a term.

The set of all terms in Σ is denoted as T_Σ; these are also referred to as *ground* terms. Each sort has an associated set of (infinitely many) *variables*, e.g. $\{x, x_1, x_2, \dots\}$ and these sets are disjoint for different sorts. The set of all terms in $\Sigma \cup X$, where X is a sorted set of variables, is constructed by treating the variables like constants and is denoted as $T_\Sigma(X)$. $\Sigma \cup X$ is also denoted by $\Sigma(X)$.

The set of all ground terms of sort t in the signature Σ is denoted $(T_\Sigma)_t$. If $s > t$ then $(T_\Sigma)_s \supseteq (T_\Sigma)_t$, and similarly for $T_\Sigma(X)$.

The set of *subterms* of a term $v = f(u_1, \dots, u_n)$ is defined to be $\{v\} \cup \{w \mid w \text{ is a subterm of } u_i\}$, for $i = 1, \dots, n$.

A *substitution* σ is a mapping from variables to terms (usually with the restriction that all but finitely many variables map to themselves) such that if x is a variable of sort s, then x maps to a term of sort s.

Two properties of signatures which are usually regarded as desirable are defined below. Σ is *monotonic* iff for every pair of operator arities $f: s_1 \dots s_n \rightarrow s, f: t_1 \dots t_n \rightarrow t$, if $\forall i: 1 \leqslant i \leqslant n$, $s_i \leqslant t_i$, then $s \leqslant t$. Σ is *regular* iff every term has a *least sort*. We use the notation LS(u) to denote the least sort of u, when it exists.

A *term-rewriting system* consists of a set of *rules* (R) over a signature. A (rewrite) rule is an ordered pair of terms of the form $l \rightarrow r$.

A term u may be *reduced* by a rule $l \rightarrow r$ in the following way. Let a subterm of u be $l\sigma$, for some substitution σ (written as $u = u[l\sigma]$; otherwise u cannot be reduced by $l \rightarrow r$). Then u can be reduced to $u' = u[r\sigma]$, which we write $u \rightarrow u'$. The slight ambiguity between rewrite rules and reductions will be resolved by context. Informally, *rewriting* with the rule $l \rightarrow r$ replaces $l\sigma$ with $r\sigma$. If $u \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow v$ (zero or more rewrite steps), we write $u \rightarrow^* v$. A term which cannot be reduced by any rule in R is said to be a *normal form* (with respect to R).

In practice, we use a term-rewriting system to *implement* an *equational theory* E. Our intention is to have a decision procedure for equality: terms u, v are equal in the equational theory *if* there is some w such that $u \rightarrow^* w$ and $v \rightarrow^* w$. Note that, in

general, this is not an equivalence; it is an equivalence only when the set of rewrite rules is confluent.

The *Knuth–Bendix completion algorithm* [18] is central to the theory of term rewriting. The algorithm takes a set of rules R and, within certain restrictions, generates new rules of the form $l \rightarrow r$, where $l =_E r$. These rules are added to R and the process iterates.

There are three possible outcomes of the algorithm:

- The algorithm terminates with success, in which case a (finite) canonical set of rules is output.
- The algorithm terminates with failure, due to the restrictions within which it must operate.
- The algorithm *diverges*, i.e. fails to terminate, in which case the canonical set of rules derived is infinite.

It is this last case which interests us. We would like to transform effectively a divergent case of Knuth–Bendix completion into a case of termination with success. Unfortunately, it is, in general, undecidable whether the algorithm will diverge, although there are some classes of rule set where we can effectively recognise divergence [13]. We will return to this topic in Section 8.

The above is a brief introduction or recap of those basic definitions in the field of term rewriting which will be needed for this paper. We have omitted as much detail as possible in order that this paper will be accessible to nonspecialists. For a more detailed description of order-sorted term rewriting, the reader is referred to [27].

Finally, we note that the notions of single-sorted (unsorted) rewriting (in which $S = \{s\}$) and many-sorted rewriting (in which $< = \{\ \}$) can be naturally regarded as special cases of order-sorted term rewriting.

2. An example [2]

Consider the set of rules generated by application of the Knuth–Bendix completion algorithm to the rule

(R) $\quad f(g(f(x))) \rightarrow g(f(x))$

where we assume a single-sorted signature with no operators apart from $g: T \rightarrow T$, $f: T \rightarrow T$ and a constant symbol $c:T$. Then the complete set of rules generated is the infinite sequence

(R1) $\quad f(g(f(x))) \rightarrow g(f(x))$

(R2) $\quad f(g(g(f(x)))) \rightarrow g(g(f(x)))$

(R3) $\quad f(g(g(g(f(x))))) \rightarrow g(g(g(f(x))))$

etc. We use R^∞ to denote this infinite sequence.

It can easily be seen that the rules in R^∞ fall into a clear pattern:

$$f(g^n(f(x))) \to g^n(f(x)) \quad \text{for any } n>0.$$

In fact, we might observe that all terms of the form

$$t = g^n(f(x)) \quad \text{for any } n>0$$

are qualitatively different from all others; these are exactly the terms for which

$$f(t) \to t.$$

Note that we cannot generalise R^∞ by the rule

$$f(y) \to y,$$

where y is a variable of sort S. Such a rule is too powerful – it equates terms which have different normal forms under R^∞. If we add such a rule, then the new rule set is not a conservative extension of the original.

If we were able to define a variable y which could *only* be instantiated by terms of the form $g^n(f(x))$, $n>0$, then we would be able to replace the infinite sequence by the single rule $f(y) \to y$. We can do this by defining a new sort V which contains exactly those terms of the form $g^n(f(x))$, $n>0$, and modifying the arities of f and g appropriately:

Sorts T, U, V with $U < T$ and $V < T$

$$c : T$$

$$f : T \to U$$

$$g : T \to T$$

$$g : U \to V$$

$$g : V \to V$$

Now the single rule

$$f(y) \to y$$

with variable y of sort V is

(i) a complete rewrite set;

(ii) a conservative extension of R^∞.

Moreover, the order-sorted signature is monotonic and regular.

We note that this new system allows the rewriting of terms of sort U to sort V, which are incomparable with respect to the sort (inclusion) ordering. In general, we are unable to guarantee the property of sort-decreasingness which is often taken to be a necessary condition for the confluence of a given set of rewrite rules (see, for example, [27]). Thus, our work depends on the use of a method of removing the requirement of sort-decreasingness and we rely on the idea of dynamic sorting [32]

(cf. critical pairs lemma in [33]). Alternatively, we could use the idea of completion with membership constraints [6].

We will formalise the procedure above in the algorithm presented in Section 5. In the next two sections we review some background material and preliminaries concerning the input grammar for our algorithm.

3. Languages and grammars: preliminaries

In Section 2 we introduced informally the idea of regarding an infinite set of rewrite rules as a language. The languages we are interested in can be characterised in various ways, for example, as regular tree languages: those languages accepted by regular tree automata [11, 22].

Definition (Kucherov [22]). A *regular tree language* is a set of terms over a signature derived from a *regular tree grammar*

$$G = (V, \Sigma, S, P),$$

where

V is a finite set of variables or non-terminals,

Σ is a finite signature,

S is a special nonterminal called the start symbol,

P is a finite set of productions each of which is of the form

$$A \to s,$$

where $s \in T_\Sigma(V)$ and $A \in V$.

The language of terms generated by G is the set of ground terms derivable from S in one or more steps and is denoted by $L(G)$, or by $L(S)$ if the grammar used is clear.

In tree grammars, the terms constructed from Σ play the role of terminal symbols in (string) grammars. Since we may be concerned with signatures enriched with variables, i.e. $\Sigma(X)$, we will refer to the constant operator symbols as constant-terminals and the variables as variable-terminals.

In order to simplify our treatment of grammars in our algorithm, we assume that the grammars are *weakly simple*, as defined below.

Definition. A tree grammar G is *weakly simple* iff for every nonterminal N, N cannot be derived from N in one or more steps and every production rule in G has one of the forms:

$$N \to f(x_1, \ldots, x_n),$$

or

$$N \to f,$$

or

$$N \to N',$$

where each x_i is either a constant-terminal, variable-terminal, or a nonterminal, f is an operator symbol, N and N' are nonterminals.

Lemma 3.1. *Any regular tree grammar can be effectively transformed into a weakly simple grammar generating the same language.*

Proof. Trivial. □

See Section 8.2 for a further discussion of the relationship between grammars and signatures.

4. Language of varying parts

From a given rewriting system R, let R^∞ be the infinite sequence of rules generated by the Knuth–Bendix completion algorithm. We partition R^∞ into Q and Q^∞, where Q^∞ is the infinite sequence we wish to generalise; i.e. we aim to replace Q^∞ by a finite set of rules. In order to do so, we first identify the language which has to be generalised.

Definition. The *position* of a subterm is that of its top function symbol. The top function symbol of a term has position 1. If a subterm t is the nth argument of a function symbol in position p, the position of t is $p.n$.

Note. The _._ operator is intended to be associative, so no brackets are required.

Definition. When p is a position in a term, then $t[p]$ is the subterm of t at position p, and $t[p/x]$ denotes the term arrived at by replacing the subterm at position p in t by x.

Definition. Let Q^∞ be the sequence

$$l_1 \to r_1$$
$$l_2 \to r_2$$
$$l_3 \to r_3$$
$$\vdots$$

A description of the *varying positions* of Q^∞ is a pair of sequences of positions $[\langle p_1, \ldots, p_m \rangle, \langle q_1, \ldots q_n \rangle]$, $m > 0$, $n \geqslant 0$, such that

(i) $\forall i, j, k,\ j \neq k$: $l_j[p_i] \neq l_k[p_i]$ (terms appearing in varying positions differ between rules),

(ii) $\forall i$: $l_i[p_1] = l_i[p_2] = \cdots = l_i[p_m] = r_i[q_1] = \cdots = r_i[q_n]$ (all varying positions in a rule contain the same term),

(iii) $\forall i: l_i[p_1/x, \ldots, p_m/x] = l_{i+1}[p_1/x, \ldots, p_m/x]$ and $r_i[q_1/x, \ldots, q_n/x] = r_{i+1}[q_1/x, \ldots, q_n/x]$, (all left (right)-hand sides are identical except at or below a varying position),

(iv) no variable which occurs at or below a varying position can occur at a position disjoint from the varying positions and each variable may occur at most once at or below each varying position.

As an example, the pair $[\langle 1.1.1\rangle, \langle 1.1\rangle]$ describes the varying positions of the sequence (R1),(R2),... given in Section 2. Note that the varying positions are not necessarily unique. For example, the pair $[\langle 1.1\rangle, \langle 1\rangle]$ is also a description. Conventionally, we will choose the varying positions furthest from the root, unless otherwise stated.

We are interested in the language consisting of the terms at the varying positions.

Definition. Let $[\langle p_1, \ldots, p_m\rangle, \langle q_1, \ldots, q_n\rangle]$, $m>0$, $n\geqslant 0$, be the varying positions of an infinite sequence Q^∞ of rewrite rules. The language of the *varying parts* of Q^∞ is defined by

$$\mathrm{LVP}(Q^\infty) = \{l_1[p_1], l_2[p_1], \ldots\}.$$

Note. We need only consider the subterms at one of the varying positions in each rule.

With reference to the example above, the pair $[\langle 1.1.1\rangle, \langle 1.1\rangle]$ generates the language $\{f(x), g(f(x)), g(g(f(x))), \ldots\}$ and the pair $[\langle 1.1\rangle, \langle 1\rangle]$ generates the language $\{g(f(x)), g(g(f(x))), \ldots\}$.

4.1. Language of varying parts with leading variables

We intend to characterise the language of varying parts by a grammar. However, we do not require a grammar which *exactly* generates the language of varying parts because that language may contain variables. The importance of these variables is not that they mark the position of any particular subterm relative to other subterms, but that they indicate the *sort* of the subterm which must occur in that position in order for the rule to be applicable. Thus, the language we will work with does not distinguish between variables of the same sort.

Definition. Let $\mathrm{LVP}(Q^\infty)$ be the language of varying parts for an infinite sequence Q^∞, over signature $\Sigma(X)$, and let each sort T in Σ have in its variable set a distinguished variable called the *leading variable* of that sort. For a given term $t \in T_\Sigma(X)$, we define $\mathrm{lv}(t)$ to be the term derived from t be replacing every variable in t with the leading variable of the appropriate sort.

Now, two further languages are defined: $\mathrm{LVP}^*(Q^\infty)$ and $\mathrm{LVP}^+(Q^\infty)$.

Definition. Let $\mathrm{LVP}(Q^\infty)$ be the language of varying parts for an infinite sequence Q^∞, over signature $\Sigma(X)$. The languages $\mathrm{LVP}^*(Q^\infty)$ and $\mathrm{LVP}^+(Q^\infty)$ are defined by

$$\mathrm{LVP}^*(Q^\infty)=\{\sigma t \mid t\in\mathrm{LVP}(Q^\infty) \text{ and } \sigma: X\to T_\Sigma(X)\},$$

$$\mathrm{LVP}^+(Q^\infty)=\{\mathrm{lv}(t)\mid t\in\mathrm{LVP}(Q^\infty)\}.$$

We note that $\mathrm{LVP}^+(Q^\infty)$ and $\mathrm{LVP}(Q^\infty)$ are contained in $\mathrm{LVP}^*(Q^\infty)$, and if there are no variables in the signature, then the languages $\mathrm{LVP}(Q^\infty)$, $\mathrm{LVP}^+(Q^\infty)$ and $\mathrm{LVP}^*(Q^\infty)$ are identical.

5. Two presentations of the algorithm

Provided the language $\mathrm{LVP}^+(Q^\infty)$ can be described by a (weakly simple) regular tree grammar, our approach enriches the signature in the appropriate way, so that Q^∞ can be generalised. We do not concern ourselves here with the generation of the grammar from the finite subset of the infinite sequence Q^∞ which we can see up to any one time, but we proceed by inspection. We will return to the question of how the grammar is obtained in Section 9.1.

We define new sorts and function arities to form an enriched, regular and monotonic signature based on the grammar defining the language $\mathrm{LVP}^+(Q^\infty)$. (We have assumed that the properties of regularity and monotonicity are desirable, thus the resulting signature has these properties, regardless of whether or not they hold for the original signature.)

Our aim is to produce an enriched signature with a distinguished sort $\mathscr{S}$ such that a term $t\in T_\Sigma(X)$ has sort $\mathscr{S}$ iff t is a term in the language $\mathrm{LVP}^*(Q^\infty)$. Then, the single rule $l_1[p_1/x,\ldots,p_m/x]\to r_1[q_1/x,\ldots,q_n/x]$, where x is a variable of sort $\mathscr{S}$, under dynamically sorted rewriting, generalises Q^∞.

We present our approach both by a set of inference rules, and by an explicit algorithm. In both cases, we begin with the following assumptions.

Let R^∞ be a rewriting system over signature $\Sigma(X)$ and let Q^∞ be the infinite sequence of rewrite rules which we wish to generalise. Let $\mathrm{LVP}(Q^\infty)$ be the chosen language of varying parts. Note, in general, R^∞ may contain more than one sequence to be generalised, in which case the algorithm/rules may be applied in turn to each sequence (cf. Example 7.6 in Section 7).

Recall that by an abuse of notation, we identify the sort ordering $<$ with its transitive closure. Thus, when we add new pairs to the relation, we do not explicitly add the transitive consequences.

Let G be a weakly simple tree grammar over signature $\Sigma(X)$, with nonterminals V and start symbol S such that

- G generates the language $\mathrm{LVP}^+(Q^\infty)$,
- there is a sort Y in Σ such that every term in $L(G)$ has sort Y and Y is minimal among such sorts.

5.1. Inference rules

The inference rules are grouped into three sets: the first set converts productions into signature enrichments; the second set performs further enrichments, or tidies up the signature, assuming that all productions have been converted; the final set replaces the infinite set of rewrite rules by a finite set. The starting position for the inference is $\langle S, <, F\rangle; P; R^\infty$, where $\Sigma = \langle S, <, F\rangle$, P is the set of productions in G, and R^∞ is the set of rewrite rules including Q^∞. We note that symbols S, $<$ and F are also used as metavariables in the inference rules. The signature Σ' is defined when no more inference rules from the first two sets can be applied.

Rules for signature enrichment

Sort 1 (nonterminals become sorts):

$$\frac{\langle S, <, F\rangle; P; R^\infty}{\langle S \cup \mathcal{N}, <, F\rangle; P; R^\infty}$$

if N is a nonterminal in G.

Sort 2 (orders start symbol sort):

$$\frac{\langle S, <, F\rangle; P; R^\infty}{\langle S, < \cup \{(\mathcal{S}, Y)\}, F\rangle; P; R^\infty}.$$

Sort 3 (constant operands get their own sorts):

$$\frac{\langle S, <, F\rangle; P; R^\infty}{\langle S \cup \{\ell\}, < \cup \{(\ell, U)\}, F \cup \{t{:}\ell\}\rangle; P; R^\infty}$$

if $t{:}U \in F$ and $N \to f(\ldots, t, \ldots) \in P$.

Order (converts productions between nonterminals):

$$\frac{\langle S, <, F\rangle; P \cup \{N \to N'\}; R^\infty}{\langle S, < \cup \{(\mathcal{N}', \mathcal{N})\}, F\rangle; P; R^\infty}.$$

Convert 1 (converts constant-terminals to sorts):

$$\frac{\langle S, <, F\rangle; P \cup \{N \to f(\ldots, t, \ldots)\}; R^\infty}{\langle S, <, F\rangle; P \cup \{N \to f(\ldots, \ell, \ldots)\}; R^\infty}$$

if $t{:}U \in F$ and $\ell \in S$.

Convert 2 (converts variable-terminals to sorts):

$$\frac{\langle S, <, F\rangle; P \cup \{N \to f(\ldots, x, \ldots)\}; R^\infty}{\langle S, <, F\rangle; P \cup \{N \to f(\ldots, T, \ldots)\}; R^\infty}$$

if $T \in S$ and $x{:}T \in X$.

Convert 3 (converts non terminals to sorts):

$$\frac{\langle S, <, F\rangle; P \cup \{N \to f(\ldots, M, \ldots)\}; R^{\infty}}{\langle S, <, F\rangle; P \cup \{N \to f(\ldots, \mathscr{M}, \ldots)\}; R^{\infty}}$$

if $M \in V$ and $\mathscr{M} \in S$.

Prod 1 (converts productions to operator arities):

$$\frac{\langle S, <, F\rangle; P \cup \{N \to f(M_1, \ldots, M_n)\}; R^{\infty}}{\langle S, <, F \cup \{f: M_1 \ldots M_n \to \mathscr{N}\}\rangle; P; R^{\infty}}$$

if $M_1, \ldots, M_n, \mathscr{N} \in S$ and $n \geqslant 0$.

Prod 2 (converts productions to subsort ordering):

$$\frac{\langle S, <, F\rangle; P \cup \{N \to x\}; R^{\infty}}{\langle S, < \cup \{(T, \mathscr{N})\}, F\rangle; P; R^{\infty}}$$

if $\mathscr{N} \in S$ and $x: T \in X$.

Rules for signature properties

Reg (ensures regularity):

$$\frac{\langle S, <, F\rangle; \emptyset; R^{\infty}}{\begin{array}{l}\langle S \cup \{\mathrm{GLB}(t,t')\}, < \cup \{(\mathrm{GLB}(t,t'), t),\ (\mathrm{GLB}(t,t'), t')\}, \\ \quad F \cup \{f: u_1 \ldots u_n \to \mathrm{GLB}(t,t')\}\rangle; \emptyset; R^{\infty}\end{array}}$$

if $f: s_1 \ldots s_n \to t, f: s_1' \ldots s_n' \to t' \in F$, $n \geqslant 0$,
for all $i \leqslant n$, $u_i \leqslant s_i$, $u_i \leqslant s_i'$, and
the u_i are maximal among such sorts and $\sim(t \leqslant t')$ and $\sim(t' \leqslant t)$.

Note. The new sorts are intended to be greatest lower bounds, thus $\mathrm{GLB}(x, y) = \mathrm{GLB}(y, x)$, $\mathrm{GLB}(x, \mathrm{GLB}(y, z)) = \mathrm{GLB}(x, y, z)$, etc.

Mono (ensures monotonicity):

$$\frac{\langle S, <, F \cup \{f: s_1 \ldots s_n \to t\}\rangle; \emptyset; R^{\infty}}{\langle S, <, F\rangle; \emptyset; R^{\infty}}$$

if for some $s_1' \ldots s_n', t', f: s_1' \ldots s_n' \to t' \in F$, $n \geqslant 0$, $t' < t$,
and for all $i \leqslant n$, $s_i' \geqslant s_i$.

Remove (removes redundant sorts):

$$\frac{\langle S \cup \{s\}, <,\ F\rangle; \emptyset; R^{\infty}}{\langle S, <, F\rangle; \emptyset; R^{\infty}}$$

if $\sim(s = \mathscr{S}$ or $\exists f \in F$ s.t. s occurs in an arity of $f)$.

Rule for synthesising rewrite rule

Generalisation (replaces infinite sequence of rules by one rule):

$$\frac{\Sigma';\emptyset;Q\cup Q^{\infty}}{\Sigma';\emptyset;Q\cup\{l[p_1/x,\ldots,p_m/x]\to r[q_1/x,\ldots,q_n/x]\}}$$

if $[\langle p_1,\ldots,p_m\rangle,\langle q_1,\ldots,q_n\rangle]$ are the varying positions and x is a variable of sort $\mathscr{S}$.

5.2. *Algorithm*

Now we present an explicit algorithm which synthesises the enriched signature and generalising rewrite rule for the infinite set of rewrite rules. It is this presentation of the algorithm which we will use in the proof of correctness.

The algorithm consists of 7 steps. The first 3 steps correspond to the first set of inference rules, Steps 4–6 correspond to the second set of inference rules, and Step 7 corresponds to the last inference rule. We note that steps 1–6 of our algorithm effectively construct a tree automaton for the language $\mathrm{LVP}^{+}(Q^{\infty})$.

In addition to the assumptions given above, let $Z=(\{Y,\mathscr{S}\},\{(\mathscr{S},Y)\},\{\ \})$ be a triple consisting of sorts, a relation $<$ on sorts and operator arities. Note, Z may only be a fragment of a signature. We now proceed to enrich Z and combine it with Σ as follows:

Step 1 (add sorts):

For every nonterminal N in V, add the sort $\mathscr{N}$ to Z (nonterminals are sorts).

For every constant-terminal t in T, if t occurs as an operand in the right-hand side of a rule then define a new sort, ℓ, say. Add sort ℓ and operator $t:\ell$ to Z. If t is a term of sort U in Σ, then add the pair (ℓ,U) to $<$ in Z, i.e. order $\ell<U$.

Define the partial function sort: $V\cup T\to$ Sorts of $Z\cup$ VarSorts, where VarSorts is the set of sorts of the variable-terminals in G, by

$\mathrm{sort}(t)=T$ if t is a variable-terminal of sort T,

$\mathrm{sort}(t)=\ell$ if $t:\ell$ was defined in the previous substep,

$\mathrm{sort}(N)=\mathscr{N}$ if N is a nonterminal.

Step 2 (add operator arities and sort orderings):

For every production of the form $N\to f$, where f is a constant-terminal, add the operator arity

$$f:\mathscr{N}$$

to Z.

For every production of the form $N\to f$, where f is a variable-terminal of sort T, add the pair $(T,\mathscr{N})$ to $<$ in Z, i.e. order $T<\mathscr{N}$.

For every production of the form $N\to f(x_1,\ldots,x_n)$, $n>0$, add the operator arity

$$f:\mathrm{sort}(x_1)\ldots\mathrm{sort}(x_n)\to\mathscr{N}$$

to Z.

For every production of the form $N \to N'$, where N' is a nonterminal, add the pair

$$(\mathscr{N}', \mathscr{N})$$

to the relation $<$ in Z, i.e. order $\mathscr{N}' < \mathscr{N}$.

Step 3 (combine Z and Σ):
Let Σ' be the union of Z and Σ.

Step 4 (ensure regularity):
For each n-ary operator f, $n \geqslant 0$, in Σ', for each pair of arities

$$f: s_1 \ldots s_n \to t,$$

$$f: s_1' \ldots s_n' \ldots t',$$

with $\sim(t' \leqslant t \vee t \leqslant t')$, for each sequence of sorts $\langle u_1, \ldots, u_n \rangle$ such that for all $i = 1, \ldots, n$, $u_i \leqslant s_i$ and $u_i \leqslant s_i'$, and u_i is maximal among such sorts,
do:

add the new sort $\mathrm{GLB}(t, t')$ to Σ',
add the pairs $(\mathrm{GLB}(t, t'), t)$ and $(\mathrm{GLB}(t, t'), t')$ to the relation $<$ in Σ',
if for any r we have $r < t$ and $r < t'$,
then add $(r, \mathrm{GLB}(t, t'))$ to the relation $<$ in Σ',
add a new arity $f: u_1 \ldots u_n \to \mathrm{GLB}(t, t')$ to Σ'.

(*Note.* Any of these substeps must be omitted if done already. As before, we have $\mathrm{GLB}(x, y) = \mathrm{GLB}(y, x)$, $\mathrm{GLB}(x, \mathrm{GLB}(y, z)) = \mathrm{GLB}(x, y, z)$, etc.)

Step 5 (ensure monotonicity):
For each n-ary operator f, $n \geqslant 0$, in Σ', for each ordered pair of arities

$$f: s_1 \ldots s_n \to t,$$

$$f: s_1' \ldots s_n' \to t'$$

if for all $i = 1, \ldots, n$ $s_i \geqslant s_i'$, then:

if $t' > t$ then delete the arity $f: s_1' \ldots s_n' \to t'$ from Σ'.

Step 6 (remove redundant sorts):
For every sort s in Z, excepting sort $\mathscr{S}$, if s does not occur in an operator arity, then delete s from Σ'.
(*Note.* Weaker redundancy conditions are possible.)

Step 7 (deduce generalising rule):
Let $l \to r$ be a rule in Q^∞. Replace Q^∞ by the single rule

$$l[p_1/x, \ldots, p_m/x] \to r[q_1/x, \ldots, q_n/x],$$

where $[\langle p_1, \ldots, p_m \rangle, \langle q_1, \ldots, q_n \rangle]$ are the varying positions and x is a variable of sort $\mathscr{S}$.
End of Algorithm.

6. Correctness of algorithm

In [29], it is shown that if a rewrite rule ρ is an exact (normal) generalisation of Q^∞, then $Q \cup \rho$ is a conservative extension of R^∞, where $R^\infty = Q \cup Q^\infty$. We will show in Theorem 6.7 that we, in fact, achieve a slightly stronger property than conservative extension. First, however, we show that we have captured exactly the right terms in each sort, which is proved by Theorem 6.1 below.

Let $G = \langle V, \Sigma, S, P \rangle$ be a weakly simple tree grammar generating $\mathrm{LVP}^+(Q^\infty)$. Let Σ and Σ' be the signature before and after application of the algorithm, respectively, and let X be the sorted set of variables occurring in R^∞. Let N be any nonterminal in the grammar G such that $\mathcal{N}$ is a sort in Σ'. $L(N)$ is the (regular tree) language defined by the grammar $\langle V, \Sigma, N, P \rangle$; $L(S) = \mathrm{LVP}^+(Q^\infty)$. $(T_{\Sigma'})_{\leqslant \mathcal{N}}$ is the set of all terms t in Σ' with least sort $\mathrm{LS}(t) \leqslant \mathcal{N}$. We may call any such t a term of sort $\mathcal{N}$. Recall that $\mathrm{lv}(t)$ is the term arrived at by replacing every variable in t by the leading variable of the same sort.

Theorem 6.1 (Term set theorem). *Let t be a term in $T_\Sigma(X)$.*
Then

$$(\exists v, \sigma\colon \sigma v = t \ \wedge \ \mathrm{lv}(v) \in L(N)) \ \Leftrightarrow \ t \in (T_{\Sigma'}(X))_{\leqslant \mathcal{N}}.$$

Corollary 6.2 (All instances of varying parts captured by the term set of a sort).

$$\mathrm{LVP}^*(Q^\infty) = (T_{\Sigma'}(X))_{\leqslant \mathcal{S}}.$$

Proof of Corollary 6.2.

$$t \in \mathrm{LVP}^*(Q^\infty) \ \Leftrightarrow \ (\exists v, \sigma\colon \sigma v = t \ \wedge \ \mathrm{lv}(v) \in L(S)) \quad \text{(by Definition)}$$

$$\Leftrightarrow \ t \in (T_{\Sigma'}(X))_{\leqslant \mathcal{S}} \quad \text{(by Theorem 6.1).} \ \square$$

Lemma 6.3. *If at some point in the algorithm we introduce an arity*

$$f\colon S_1 \ldots S_n \to S$$

then in Σ' there is an arity

$$f\colon S'_1 \ldots S'_n \to S',$$

where for $i = 1, \ldots, n$ $S'_i \geqslant S_i$ and $S' \leqslant S$.

Proof of Lemma 6.3. We only delete arities at Step 5. Proof is immediate from Step 5. $\square$

Lemma 6.4. *If $N \to N'$, where N and N' are nonterminals, is a production then $\mathcal{N}' < \mathcal{N}$ in the sort ordering. If $\mathcal{N}' < \mathcal{N}$ in the sort ordering and $\mathcal{N}, \mathcal{N}'$ are introduced at Step 1, then there is a derivation $N \to \cdots \to N'$.*

Proof of Lemma 6.4. Immediate from Step 2. □

Lemma 6.5.

$$\mathrm{lv}(v)\in(T_{\Sigma'}(X))_{\iota'} \Leftrightarrow v\in(T_{\Sigma'}(X))_{\iota'}$$

Proof of Lemma 6.5. $\Rightarrow$: Trivial because substitutions are sort-preserving.

$\Leftarrow$: Obvious because the leading variable substitution replaces variables with variables of the same sort; so, the same operator arities apply to v as those which apply to $\mathrm{lv}(v)$. □

Note. Lemma 6.5 simplifies the proof of Theorem 6.1 ($\Rightarrow$) by allowing us to consider only terms of the form $\mathrm{lv}(v)$.

Proof of Theorem 6.1. $\Rightarrow$: Let $t=\sigma v$ and $\mathrm{lv}(v)\in L(N)$. We show that $\mathrm{lv}(v)\in(T_{\Sigma'}(X))_{\iota'}$. Then by Lemma 6.5 and well-sortedness of substitution, $t\in(T_{\Sigma'}(X))_{\iota'}$. Proof is by induction on the height of the tree which represents $\mathrm{lv}(v)$.

Base step: $\mathrm{lv}(v)$ is a constant in Σ' or a variable in X.

The shortest derivation of $\mathrm{lv}(v)$ from N has the form

$$N\to M_1\to\cdots\to M_p\to\mathrm{lv}(v)$$

for some $p\geqslant 0$.

Case (*a*): $\mathrm{lv}(v)$ is a constant terminal.

Then at Step 2 the production gives us $\mathrm{lv}(v):\mathscr{M}_p$ so by Lemma 6.3 $\mathrm{lv}(v)\in(T_{\Sigma'}(X))_{\mathscr{M}_p}$ and by Lemma 6.4 $\mathscr{M}_p\leqslant\mathscr{M}_{p-1}\leqslant\cdots\leqslant\mathscr{M}_1\leqslant\mathscr{N}$. So, $\mathrm{lv}(v)\in(T_{\Sigma'}(X))_{\iota'}$.

Case (*b*): $\mathrm{lv}(v)=x$, a variable terminal of sort T in $T_{\Sigma}(X)$.

Then the second substep of Step 2 and Lemma 6.4 give us $T\leqslant\mathscr{M}_p\leqslant\mathscr{M}_{p-1}\leqslant\cdots\leqslant\mathscr{M}_1\leqslant\mathscr{N}$. So, $x\in(T_{\Sigma'}(X))_{\iota'}$.

Induction step: Let $\mathrm{lv}(v)=f(u_1,\ldots,u_n)$, $n>0$, and $\mathrm{lv}(v)$ has tree height $k+1$, i.e. some u_i, $1\leqslant i\leqslant n$, has height k and all u_j, $i\leqslant j\leqslant n$, have height $\leqslant k$.
We derive

$$N\to\cdots\to N'\to f(U_1,\ldots,U_n)\to\cdots\to f(u_1,\ldots,u_n),$$

where for $i=1,\ldots,n$ either $U_i=u_i$ or $U_i\to\cdots\to u_i$, in which case, by induction, u_i is a term of $(T_{\Sigma'}(X))_{\mathscr{U}_i}$ since tree height of u_i is $\leqslant k$.

Then by Step 2 and Lemma 6.3, $\mathrm{lv}(v)$ is of some sort $V\leqslant\mathscr{N}'$ and by repeated application of Lemma 6.4, $\mathrm{lv}(v)\in(T_{\Sigma'}(X))_{\iota'}$. So by Lemma 6.5 and well-sortedness of substitution, $t\in(T_{\Sigma'}(X))_{\iota'}$.

Lemma 6.6. *If $\mathscr{T}$ is introduced as a new sort at Step* 1 *to allow the declaration $t:\mathscr{T}$ then no term other than t is ever of sort $\mathscr{T}$.*

Proof of Lemma 6.6. T occurs in no productions; so, it has no subsorts. Thus, we need only be concerned with terms of the forms:

(i) $t':\mathscr{T}$,

(ii) $f(t_1,\ldots,t_n)$, where $f:\mathscr{T}_1\ldots\mathscr{T}_n\to\mathscr{T}$ is an arity and t_i is of sort $\mathscr{T}_i$ for $i=1,\ldots,n$.

But note that during the algorithm we only introduce an arity like (i) or (ii) if T is a nonterminal, and, by assumption, T is not a nonterminal. So, we are done. □

Lemma 6.7. *Let s be any sort in Σ'.* $\mathrm{GLB}(\mathscr{T}_1,\ldots,\mathscr{T}_n)<s$ *iff for some* i: $1\leqslant i\leqslant n$ $\mathscr{T}_i\leqslant s$.

Proof of Lemma 6.7. $\Leftarrow$: Suppose $\sim(\mathrm{GLB}(\mathscr{T}_1,\ldots,\mathscr{T}_n)<s)$. We have

$$\mathrm{GLB}(\mathscr{T}_1,\ldots,\mathscr{T}_n)<\mathscr{T}_i \quad \text{for all } i=1,\ldots,n,$$

so we cannot have $\mathscr{T}_i\leqslant s$ for any i or we contradict our assumption.

$\Rightarrow$: The sort $\mathrm{GLB}(\mathscr{T}_1,\ldots,\mathscr{T}_n)$ is introduced at Step 4; so, the only sort inclusions we have involving $\mathrm{GLB}(\mathscr{T}_1,\ldots,\mathscr{T}_n)$ are of the form

$$\mathrm{GLB}(\mathscr{T}_1,\ldots,\mathscr{T}_n)<\mathscr{T}_i \quad \text{for all } i=1,\ldots,n.$$

Thus, $\mathrm{GLB}(\mathscr{T}_1,\ldots,\mathscr{T}_n)<s$ implies either:

(i) $s=\mathscr{T}_i$ for some i, or

(ii) $s>\mathscr{T}_i$ for some i.

In each case we are done. □

Proof of Theorem 6.1 (*continued*). $\Leftarrow$: Let t be a term of $(T_{\Sigma'}(X))_{\mathscr{N}}$.

Case (*a*): Suppose $t\in T_{\Sigma'}(X)_M$ for some $M<\mathscr{N}$, where M is a sort in Σ. We can only have $M<\mathscr{N}$ if there is a production

$$N'\to y,$$

where $\mathscr{N}'\leqslant\mathscr{N}$ (so by Lemma 6.4 $N\to\cdots\to N'$) and y is a variable of sort M' in Σ, $M'\geqslant M$.

Then $y\in L(N)$ and we can find σ s.t. $t=\sigma y$ as required.

Case (*b*): Otherwise we can prove the result by induction on the height of the tree representing t.

Base step: t is a constant-terminal. There are two cases:

(i) In Step 2 there is a production $A\to t$, where $\mathscr{A}\leqslant\mathscr{N}$. But then by Lemma 6.4 $N\to\cdots\to A$; so, $t\in L(N)$.

(ii) t occurs as an operand in a production $N'\to f(\ldots,t,\ldots)$, for some N', f.

Then ℓ is created as a new sort at Step 1 and $t:\ell$. Then t cannot be a term of sort $\mathscr{N}$ because $\sim(\ell\leqslant\mathscr{N})$ by Lemma 6.4. So $\sim(t\in(T_{\Sigma}(X))_{\mathscr{N}})$ unless case (i) holds.

Induction step: Let $t=f(t_1,\ldots,t_n)\in(T_{\Sigma'}(X))_{\mathscr{N}}$ be of tree height $k+1$. Then by definition of $(T_{\Sigma'}(X))_{\mathscr{N}}$ there is in Σ' an arity $f:\mathscr{T}_1\ldots\mathscr{T}_n\to\mathscr{S}'$, where $\mathscr{S}'\leqslant\mathscr{N}$ and for $i=1,\ldots,n$, $t_i\in(T_{\Sigma'}(X))_{\mathscr{T}_i}$. The arity $f:\mathscr{T}_1\ldots\mathscr{T}_n\to\mathscr{S}'$ can have been introduced either at Step 4 or Step 2.

Suppose it was introduced at Step 4. Then $\mathscr{S}' = \mathrm{GLB}(Y, Z)$, for some Y, Z. But by Lemma 6.7 $\mathscr{S}' \leqslant \mathscr{N}$ iff $Y \leqslant \mathscr{N}$ or $Z \leqslant \mathscr{N}$. Suppose w.l.o.g. $Y \leqslant \mathscr{N}$. Also at step 4 there was already in existence an arity $f: V_1 \ldots V_n \to Y$, where for $i = 1, \ldots, n$ $V_i \geqslant \mathscr{T}_i$. By assumption, Y is not a sort in Σ (or $t \in (T_\Sigma(X))_Y$ and $Y \leqslant \mathscr{N}$, which is case (a)). So, an arity of form $f: V_1 \ldots V_n \to Y$ must have been introduced at Step 2. So, we may as well assume that the arity $f: \mathscr{T}_1 \ldots \mathscr{T}_n \to \mathscr{S}'$ was introduced at Step 2.

At Step 2 we introduced the arity

$$f: \mathscr{T}_1 \ldots \mathscr{T}_n \to \mathscr{S}',$$

because there was a production of the form $S' \to f(u_1, \ldots, u_n)$
and for all $i = 1, \ldots, n$ either

(i) $u_i = t_i$, a constant, or

(ii) $u_i = T_i$, or

(iii) $u_i : \mathscr{T}_i$, where $\mathscr{T}_i$ is introduced specifically as the sort of u_i at step 1.

If (iii) then $t_i = u_i$ or else t_i is not a term of sort $\mathscr{T}_i$ as no term other than u_i is ever in $\mathscr{T}_i$ by Lemma 6.6. So, we are reduced to cases (i) and (ii).

If (i) then

$$S' \to f(\ldots, t_i, \ldots).$$

If $i < n$ then we go to $i+1$, and if $i = n$ then $\mathscr{S}' \leqslant \mathscr{N}$; so, by Lemma 6.4, there must be a derivation

$$N \to \cdots \to S' \to f(\ldots, u_i, \ldots, t_n) \to \cdots \to f(\ldots, t_n)$$

and we are done.

If (ii) then $S' \to f(\ldots, T_i, \ldots)$ and by the induction hypothesis $\exists s_i \exists \sigma_i$: $\mathrm{lv}(s_i) \in L(T_i)$ and $t_i = \sigma_i s_i$ because $t_i \in (T_{\Sigma \cdot}(X))_{T_i}$ and t has tree height $\leqslant k$. Go to $i+1$ unless $i = n$, in which case we have

$$N \to \cdots \to S' \to f(\ldots, u_i, \ldots, T_n) \to \cdots \to f(\ldots, \mathrm{lv}(s_n)).$$

Now by condition (iv), Section 4, no variable terminal occurs more than once in each word in $L(N)$; so, the σ_i have disjoint domains and there is no difficulty in composing the σ_i into σ such that $t = \sigma f(\ldots, s_n)$ and $\mathrm{lv}(f(s_1, \ldots, s_n)) \in L(N)$, as required. □

Theorem 6.8 (Property of signature). *Σ' is regular.*

Proof (by induction on the length of terms).

Base case (terms of length 1 – constants):

Suppose $t = a$ and $a: A_i$, $i = 1, \ldots, n$, are all the minimal arities of a in Σ'. By Step 4, there exists a sort $\mathrm{GLB}(A_1, \ldots, A_n)$ and arity $a: \mathrm{GLB}(A_1, \ldots, A_n)$ such that none of the other arities of a is minimal (contradiction), unless $n = 1$ and A_1 is the least sort of a.

Induction step: (terms of length $k+1$)

Suppose $t = f(s_1, \ldots, s_n)$, where for all $i = 1, \ldots, n$, s_i has (unique) least sort B_i (each s_i has length $\leqslant k$). Let

$$f: C_{1j} \ldots C_{nj} \to A_j, \quad j = 1, \ldots, m,$$

be the arities of f such that for all $j = 1, \ldots, m$, for all $i = 1, \ldots, n$,

$$C_{ij} \geqslant B_i$$

and each A_j is minimal among the A_j, $j = 1, \ldots, m$.

By Step 4, there exists a sort $\mathrm{GLB}(A_1, \ldots, A_m)$ and a new arity

$$f: D_1 \ldots D_n \to \mathrm{GLB}(A_1, \ldots, A_m),$$

with $C_{ij} \geqslant D_i \geqslant B_i$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. $D_i \geqslant B_i$ because of Lemma 6.7 and the maximality of the u_i in Step 4. So, none of the A_i are minimal (contradiction) unless $m = 1$ and A_1 is the least sort of t. □

Theorem 6.9 (Property of signature). *Σ' is monotonic.*

Proof. Let f be any n-ary operator and

$$f: s_1 \ldots s_n \to t,$$

$$f: s'_1 \ldots s'_n \to t'$$

be arities. Then we cannot have $s_i \geqslant s'_i$ for $i = 1, \ldots, n$, with $t' > t$, or, by Step 5, we would delete the arity $f: s'_1 \ldots s'_n \to t'$ (w.l.o.g.). This suffices to prove monotonicity because the case where $\sim(t \leqslant t' \vee t' < t)$ is prevented by regularity: for all such arities, there exists an arity

$$f: s'_1 \ldots s'_n \to t'',$$

where $t'' < t$ and $t'' < t'$. □

Theorem 6.10 (Term set is unchanged). *$T_\Sigma = T_{\Sigma'}$ and $T_\Sigma(X) = T_{\Sigma'}(X)$ for any sorted variable set X.*

Proof. The proof is trivial. □

Theorem 6.11 (Equational theory is preserved). *If s, t are terms in $T_\Sigma(X)$, then*

$$s \leftrightarrow^*_{R'} t \text{ iff } s \leftrightarrow^*_{Q \cup R'} t,$$

where R' is the rule set containing the generalised rule synthesised in Step 7, i.e. the equational theory of R^∞ is preserved in $Q \cup R'$.

Proof. $\Rightarrow$: Suppose $s \rightarrow_{Q^*} t$ (case $s \rightarrow_Q t$ is trivial) and the varying positions of Q^∞ are $[\langle p_1, \ldots, p_m \rangle, \langle q_1, \ldots, q_n \rangle]$. Let $l \rightarrow r$ be a rule in Q^∞ such that $s \rightarrow_{Q^*} t$ by $l \rightarrow r$. R' contains a rule of form $l[p_1/x, \ldots, p_m/x] \rightarrow r[q_1/x, \ldots, q_n/x]$, where x is a variable of sort $\mathscr{S}$. Then, by Corollary 6.2 ($\Rightarrow$), the more general rule in R' applies whenever $l \rightarrow r$ applies. Thus, $s \rightarrow_{R^*} t$.

$\Leftarrow$: Suppose $s \rightarrow_{R^*} t$ (case $s \rightarrow_Q t$ is trivial) by the rule $l[p_1/x, \ldots, p_m/x] \rightarrow r[q_1/x, \ldots, q_m/x]$, where x is a variable of sort $\mathscr{S}$. Let s' and t' be the subterms of s and t, respectively, at the position where rewriting occurs. Then s' is of the form $l\sigma[p_1/u, \ldots, p_m/u]$ and t' is of the form $r\sigma[q_1/u, \ldots, q_m/u]$, where u is a Σ'-term of sort $\mathscr{S}$.

(*Note.* It is here that we use the requirement (iv, Section 4) that no variable which occurs in the language of varying parts can occur at a position disjoint from the varying positions. This ensures that the substitution σ does not depend on u.) By Corollary 6.2 ($\Leftarrow$), if u is a term of sort $\mathscr{S}$ then $u \in \mathrm{LVP}^*(Q^\infty)$, i.e. $\exists v, \sigma$: $\sigma v = u \wedge v \in \mathrm{LVP}(Q^\infty)$. Thus, the rule $l[p_1/v, \ldots, p_m/v] \rightarrow r[q_1/v, \ldots q_m/v]$ in R^∞ can be applied and, so, $s \rightarrow_{R^*} t$. □

Theorem 6.12 (Efficiency of rewriting). *If s, t are terms in $T_\Sigma(X)$, and s rewrites to t by R^∞ in n rewriting steps, then s rewrites to t by $Q \cup R'$ in n rewriting steps.*

Proof. Recall $R^\infty = Q \cup Q^\infty$. Since each rewrite rule in Q^∞ is generalised by the rule in R' (proof of Theorem 6.11 ($\Rightarrow$)), the number of rewriting steps is preserved. □

7. Examples

In this section we give several examples. In the first example, we simply generalise a set of terms generated by a given regular tree grammar. In the following examples, we address the more complex case where the set of terms must be deduced from a set of rewrite rules and the rules replaced by a single generalising rule.

We will give our working in each case; function arities and sorts which are introduced and later deleted are denoted by *. We shall also show at which stage (in the algorithm) each piece of information is added to Z and Σ'. In each case, unless the signature Σ is given, we assume a one-sorted signature with sort X and appropriate operator arities.

Example 7.1. Consider the set of terms which form the language $L(S)$, generated by the following (weakly simple) tree grammar:

$$L(S) = \{a(0), b(0), a(a(0)), b(b(0)), a(a(a(0))), \ldots\}$$

$$S \rightarrow a(A) | b(B)$$

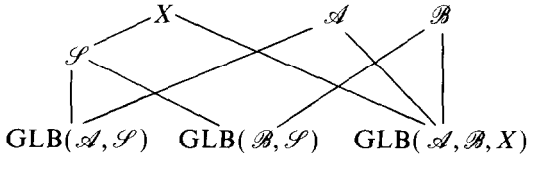
$$A \rightarrow 0 | a(A)$$

$$B \rightarrow 0 | b(B)$$

Applying the algorithm to G gives:

	$<$	*Sorts*	*Operators*	
Step 1				
	$\mathscr{S} < X$	$\mathscr{S}, \mathscr{A}, \mathscr{B}, X$		
Step 2				
			$0 : \mathscr{A}$	(*1)
			$0 : \mathscr{B}$	(*2)
			$a : \mathscr{A} \to \mathscr{S}$	(*3)
			$b : \mathscr{B} \to \mathscr{S}$	(*4)
			$a : \mathscr{A} \to \mathscr{A}$	(*5)
			$b : \mathscr{B} \to \mathscr{B}$	(*6)
Step 3				
			$0 : X$	(*7)
			$a : X \to X$	
			$b : X \to X$	
Step 4				
	$\mathrm{GLB}(\mathscr{A}, \mathscr{B}) < \mathscr{A}$	$\mathrm{GLB}(\mathscr{A}, \mathscr{B})$ (*11)	$a : \mathscr{A} \to \mathrm{GLB}(\mathscr{A}, \mathscr{S})$	
	$\mathrm{GLB}(\mathscr{A}, \mathscr{B}) < \mathscr{B}$	$\mathrm{GLB}(\mathscr{A}, \mathscr{S})$	$b : \mathscr{B} \to \mathrm{GLB}(\mathscr{B}, \mathscr{S})$	
	$\mathrm{GLB}(\mathscr{A}, \mathscr{S}) < \mathscr{A}$	$\mathrm{GLB}(\mathscr{B}, \mathscr{S})$	$0 : \mathrm{GLB}(\mathscr{A}, \mathscr{B})$	(*8)
	$\mathrm{GLB}(\mathscr{A}, \mathscr{S}) < \mathscr{S}$	$\mathrm{GLB}(\mathscr{A}, X)$ (*12)	$0 : \mathrm{GLB}(\mathscr{A}, X)$	(*9)
	$\mathrm{GLB}(\mathscr{B}, \mathscr{S}) < \mathscr{B}$	$\mathrm{GLB}(\mathscr{B}, X)$ (*13)	$0 : \mathrm{GLB}(\mathscr{B}, X)$	(*10)
	$\mathrm{GLB}(\mathscr{B}, \mathscr{S}) < \mathscr{S}$	$\mathrm{GLB}(\mathscr{A}, \mathscr{B}, X)$	$0 : \mathrm{GLB}(\mathscr{A}, \mathscr{B}, X)$	
	$\mathrm{GLB}(\mathscr{A}, X) < \mathscr{A}$			
	$\mathrm{GLB}(\mathscr{A}, X) < X$			
	$\mathrm{GLB}(\mathscr{B}, X) < X$			
	$\mathrm{GLB}(\mathscr{B}, X) < \mathscr{B}$			
	$\mathrm{GLB}(\mathscr{A}, \mathscr{B}, X) < \mathscr{B}$			
	$\mathrm{GLB}(\mathscr{A}, \mathscr{B}, X) < \mathscr{A}$			
	$\mathrm{GLB}(\mathscr{A}, \mathscr{B}, X) < X$			
Step 5				
	(delete (*1), ..., (*10))			
Step 6				
	(delete (*11), ..., (*13))			

Result:



$$(T_{\Sigma'})_{\mathscr{S}} = (T_{\Sigma'})_{\mathrm{GLB}(\mathscr{A},\mathscr{S})} \cup (T_{\Sigma'})_{\mathrm{GLB}(\mathscr{B},\mathscr{S})}$$
$$= \{a(0), a(a(0)), \dots\} \cup \{b(0), b(b(0)), \dots\}$$
$$= L(S).$$

We will now consider examples which arise from sets of rewrite rules. Note that we do not attempt to formalise the procedure of defining the grammar but we proceed "by inspection". This part of the process will be considered in Section 9.1. We begin with the example discussed in Section 2.

Example 7.2. (Ardis [2]).

Sorts: T

Operators:

$c: T$

$f: T \to T$

$g: T \to T$

Rules (x is a variable of sort T):

(R) $f(g(f(x))) \to g(f(x))$

Applying Knuth–Bendix to this rule gives

(Q1) $f(g(f(x))) \to g(f(x))$

(Q2) $f(g(g(f(x)))) \to g(g(f(x)))$

(Q3) $f(g(g(g(f(x))))) \to g(g(g(f(x))))$

etc.

Consider (Q1), (Q2), ... as Q^∞, and let the varying positions be $[\langle 1.1.1\rangle, \langle 1.1\rangle]$. The language of the varying parts, $\mathrm{LVP}(Q^\infty)$, is $\{f(x), g(f(x)), g(g(f(x))), \dots\}$. Since there is only one variable of sort T, if we choose the leading variable to be that variable then we have $\mathrm{LVP}(Q^\infty) = \mathrm{LVP}^+(Q^\infty)$ and a grammar is

$$S \to g(S) \mid f(x)$$

The result of the algorithm is

	$<$	*Sorts*	F
Step 1			
	$\mathscr{S} < T$	$T, \mathscr{S}$	
Step 2			$g: \mathscr{S} \to \mathscr{S}$
			$f: T \to \mathscr{S}$

Step 3

$$c:T$$
$$g:T\to T$$
$$f:T\to T(*1)$$

Step 5

(delete (*1)).

Result:

$$\begin{array}{c} T \\ | \\ \mathscr{S} \end{array}$$

$$(T_{\Sigma'})_{\mathscr{S}} = \{\sigma(f(x)), \sigma(g(f(x))), \sigma(g(g(f(x)))), \dots\},$$

where σ is any substitution.

$$L(S) = \{f(x), g(f(x)), g(g(f(x))), \dots\}.$$

The generalising rule given by Step 7 is

$$f(g(y))\to g(y),$$

where y is a variable of sort $\mathscr{S}$.

Example 7.3. Consider again the rewriting system from the previous example, but now let the description of the varying positions be $[\langle 1.1\rangle, \langle 1\rangle]$. The language of varying parts is $\{g(f(x)), g(g(f(x))), \dots\}$ and a grammar is

$$S\to g(S)|g(F)$$

$$F\to f(x)$$

The result of the algorithm is

	$<$	*Sorts*	F
Step 1			
	$\mathscr{S} < T$	$T, \mathscr{S}, \mathscr{F}$	
Step 2			
			$g:\mathscr{S}\to\mathscr{S}$
			$g:\mathscr{F}\to\mathscr{S}$
			$f:T\to\mathscr{F}$ (*1)
Step 3			
			$g:T\to T$
			$f:T\to T$ (*2)
			$c:T$

Step 4

$\mathrm{GLB}(T, \mathscr{F}) < T$ $\quad$ $\mathrm{GLB}(T, \mathscr{F})$ $\quad$ $f: T \to \mathrm{GLB}(T, \mathscr{F})$

$\mathrm{GLB}(T, \mathscr{F}) < \mathscr{F}$

Step 5

(delete (*1) and (*2))

Result:

$(T_{\Sigma'}(X))_{y} = \{\sigma(g(f(x))), \sigma(g(g(f(x)))), \ldots\},$

where σ is any substitution.

$L(S) = \{g(f(x)), g(g(f(x))), \ldots\}.$

The generalising rule is

$f(y) \to y,$

where y is a variable of sort $\mathscr{S}$.

Note that the sort $\mathscr{F}$ does not contain any terms other than those in T, i.e. we can deduce that $\mathrm{GLB}(T, \mathscr{F}) = \mathscr{F}$ in the initial model. Indeed, a more efficient signature (i.e. no redundant sorts) is possible if at Step 2 of the algorithm we order sorts S and S' as $S < S'$ whenever we can prove inductively that S is below S'. The resulting signature and generalising rule for this example are very similar to the example given in Section 2: the sort V in Section 2 corresponds to the sort $\mathscr{S}$ here, and the sort U in Section 2 corresponds to the union of the sorts $\mathrm{GLB}(T, \mathscr{F})$ and $\mathscr{F}$. See also the note to Step 6 in the Algorithm.

Example 7.4. This example concerns lists and it comes from [13]. It demonstrates an important feature of our approach: the sequence to be generalised may contain an infinite number of variables. For example, each rule in the sequence contains one more variable than its predecessor.

Sorts: list

Operators:

nil: list,

[_]: list → list,

_@_: list list → list,

flatten: list → list.

Rules (x, y, z are variables of sort list):

(R1) $\text{nil} \mathbin{@} x \to x$

(R2) $x \mathbin{@} \text{nil} \to x$

(R3) $(x \mathbin{@} y) \mathbin{@} z \to x \mathbin{@} (y \mathbin{@} z)$

(R4) $\text{flatten}(\text{nil}) \to \text{nil}$

(R5) $\text{flatten}([x]) \to \text{flatten}(x)$

(R6) $\text{flatten}([x] \mathbin{@} y) \to \text{flatten}(x) \mathbin{@} \text{flatten}(y)$

(R7) $\text{flatten}(\text{flatten}(x)) \to \text{flatten}(x)$

The first 6 rules are complete, but applying Knuth–Bendix to rules (R1)–(R7) gives:

(Q1) $\text{flatten}(\text{flatten}(x)) \to \text{flatten}(x)$

(Q2) $\text{flatten}(\text{flatten}(x) \mathbin{@} \text{flatten}(x1)) \to \text{flatten}(x) \mathbin{@} \text{flatten}(x1)$

(Q3) $\text{flatten}(\text{flatten}(x) \mathbin{@} (\text{flatten}(x1) \mathbin{@} \text{flatten}(x2)))$

$\to \text{flatten}(x) \mathbin{@} (\text{flatten}(x1) \mathbin{@} \text{flatten}(x2))$

(Q4) $\text{flatten}(\text{flatten}(x) \mathbin{@} (\text{flatten}(x1) \mathbin{@} (\text{flatten}(x2) \mathbin{@} \text{flatten}(x3))))$

$\to \text{flatten}(x) \mathbin{@} (\text{flatten}(x1) \mathbin{@} (\text{flatten}(x2) \mathbin{@} \text{flatten}(x3)))$

etc.

Consider (Q1), (Q2), ... as Q^∞, and let the varying positions be $[\langle 1.1 \rangle, \langle 1 \rangle]$. The language of the varying parts, $\text{LVP}(Q^\infty)$ is

$\{\text{flatten}(x), \text{flatten}(x) \mathbin{@} \text{flatten}(x1),$
$\text{flatten}(x) \mathbin{@} (\text{flatten}(x1) \mathbin{@} \text{flatten}(x2)), \ldots\}$

and a grammar for $\text{LVP}^+(Q^\infty)$ (with x as leading variable) is

$S \to T \mid T \mathbin{@} S$

$T \to \text{flatten}(x)$

The result of the algorithm is:

	$<$	*Sorts*	F
Step 1:			
	$\mathscr{S} < \text{list}$	list, $\mathscr{T}, \mathscr{S}$	
Step 2:			
	$\mathscr{T} < \mathscr{S}$		$\_ @ \_ : \mathscr{T}\,\mathscr{S} \to \mathscr{S}$ $\text{flatten} : \text{list} \to \mathscr{T}$

Step 3:

$$\begin{aligned}&\text{nil}:\text{list}\\&\_@\_:\text{list list}\to\text{list}\\&[\_]:\text{list}\to\text{list}\\&\text{flatten}:\text{list}\to\text{list}\quad(*1)\end{aligned}$$

Step 5: (delete (*1))

Result:

$$\begin{array}{l}\text{list}\\ \Big|\\ \mathscr{S}\\ \Big|\\ \mathscr{T}\end{array}$$

$$(T_{\Sigma'})_{\mathscr{S}}=\{\sigma(\text{flatten}(x)),\sigma(\text{flatten}(x)\,@\,\text{flatten}(x1)),$$

$$\sigma(\text{flatten}(x)\,@\,(\text{flatten}(x1)\,@\,\text{flatten}(x2))),\ldots\},$$

$$\text{LVP}^{+}(Q^{\infty})=\{\text{flatten}(x),\ \text{flatten}(x)\,@\,\text{flatten}(x),$$

$$\text{flatten}(x)\,@\,(\text{flatten}(x)\,@\,\text{flatten}(x)),\ldots\},$$

$$\text{LVP}^{*}(Q^{\infty})=\{\sigma(\text{flatten}(x)),\ \sigma(\text{flatten}(x)\,@\,\text{flatten}(x1)),$$

$$\sigma(\text{flatten}(x)\,@\,(\text{flatten}(x1)\,@\,\text{flatten}(x2))),\ldots\},$$

where σ is any substitution.
The generalising rule given by Step 7 is

$$\text{flatten}(y)\to y$$

where y is a variable of sort $\mathscr{S}$.

Example 7.5. This example comes from the inductive synthesis of programs, as described in [7, 8]. Completion and generalisation are two important techniques used in synthesis, where often the problem is one of searching for an appropriate inductive theorem. This involves both the synthesis of a hypothesis *and* an inductive proof of that hypothesis. Alternatively, our approach can be used, in some cases, to effectively synthesise the theorem. For example, consider the following problem from [8] concerning the synthesis of the double function (the operation named d) from the following "specification".

Sorts: nat

Operators:

0:nat

s: nat → nat

$\_+\_$: nat nat → nat

d: nat → nat

Rules (x, y are variables of sort nat):

(R1) $0+x \to x$

(R2) $x+s(y) \to s(x+y)$

(R3) $s(x)+y \to s(x+y)$

(R4) $x+x \to d(x)$

In the process of synthesis, which involves deleting rules, inverting rules, and applying the completion algorithm, several infinite sequences are generated. Consider one such sequence:

(Q1) $d(s(0)) \to s(s(d(0)))$

(Q2) $d(s(s(0))) \to s(s(d(s(0))))$

(Q3) $d(s(s(s(0)))) \to s(s(d(s(s(0)))))$

etc.

Considering (Q1), (Q2), ... as Q^∞, let the varying positions be $[\langle 1.1.1\rangle, \langle 1.1.1.1\rangle]$. The language of the varying parts, $\mathrm{LVP}(Q^\infty)$ is $\{0, s(0), s(s(0)), \dots\}$ and a grammar for $\mathrm{LVP}^+(Q^\infty)$ (there are no variables) is

$$S \to 0 \mid s(S).$$

The result of our approach is to introduce a sort $\mathscr{S} <$ nat, to change the arity of 0 to $0{:}\mathscr{S}$ and to add the arity $s{:}\mathscr{S} \to \mathscr{S}$. Then the rules Q^∞ can be generalised by the equational theorem

$$d(s(y)) \to s(s(d(y))),$$

where y is a variable of sort $\mathscr{S}$.

Example 7.6. This example is taken from string rewriting and it comes from [25]. To treat this as a term rewriting system we must regard string concatenation as an "invisible" operator, with arities like any other. We take Σ to contain a universal sort X, and string concatenation is an infix operator with the arity:

$\_\_: X\ X \to X$.

Rewriting is then modulo the associativity of this operator.

This example is of particular interest because it shows that our algorithm may be applied more than once in the case where Q^{∞} contains two or more infinite sequences of rules to be generalised. In this case, a different choice of where to begin the infinite rule set and careful choice of varying parts would have enabled us to solve the problem with just one application of the algorithm, but this is not always the case, and the example is instructive for this reason.

When the algorithm is applied more than once, combinatorial explosion of the number of sorts becomes a danger. To avoid this, we will modify our algorithm by adding some tricks designed to keep the number of introduced sorts to a minimum, and to add as much structure to the signature as possible by using sort inclusion.

Consider a grammar which includes productions of the form

$$M \to g(\ldots f \ldots)$$

$$N \to f$$

where f is a constant-terminal (i.e. a constant in Σ). Then, our algorithm will introduce a sort $\mathscr{f}$ at Step 1, with $f:\mathscr{f}$ and the arity $f:\mathscr{N}$ will then be added at Step 2. The regularity requirement will add a sort $\mathrm{GLB}(\mathscr{f}, \mathscr{N})$ at Step 4 and by monotonicity the arity $f:\mathscr{f}$ will be deleted at Step 5. The sort $\mathscr{f}$ is now redundant and will be deleted at Step 6. This is a great deal of extra work when we can recognise from the grammar that *every term of sort* $\mathscr{f}$ *is a term of sort* $\mathscr{N}$ and so we may order $\mathscr{f} < \mathscr{N}$ at Step 1. This is a key idea, and a variation on this theme concerns X (the greatest sort in Σ); every sort we introduce contains only terms of sort X, and as we introduce no new operators, X will be the greatest sort in Σ' also.

Another idea we will use is to reuse sorts from one application of the algorithm to the next. This applies particularly to sorts of the form $\mathscr{f}$ where f is a constant-terminal in the language of varying parts for both rule sequences and $\mathscr{f}$ is introduced at Step 1. No term other than f is of sort $\mathscr{f}$ after the first application of the algorithm, so if the second application of the algorithm requires the introduction of a sort specifically to hold f, then one is already in existence.

Note that these ideas are effective, and so do not affect the effectiveness of our algorithm, one of its major strengths.

Sorts: X

Operators:

$$K:X$$

$$D:X$$

$$b:X$$

$$d:X$$

$$A:X$$

$$Z:X$$

$$\_\_:XX \to X$$

Rules:

(A1) $Kb \to dK$

(A2) $Ad \to bA$

(A3) $bAZ \to AZD$

Applying Knuth–Bendix completion to this rule set gives

(B1) $bAKAZ \to AKAZD$

(C1) $dKAZ \to KAZD$

(B2) $bAKAKAZ \to AKAKAZD$

(C2) $dKAKAZ \to KAKAZD$

(B3) $bAKAKAKAZ \to AKAKAKAZD$

(C3) $dKAKAKAZ \to KAKAKAZD$
etc.

For generalisation, the sequence is partitioned into two subsequences. Let Q^∞ be the sequence

(B1) $bAKAZ \to AKAZD$

(B2) $bAKAKAZ \to AKAKAZD$

(B3) $bAKAKAKAZ \to AKAKAKAZD$
etc.

Varying positions are difficult to describe in string rewriting; so, we describe the nonvarying and varying parts instead.

Nonvarying parts:

$$b \ldots AZ \to \ldots AZD.$$

Language of varying parts:

$$L(S) = \{(AK)^n,\ n > 0\}.$$

Note. There are no variables in this example and, so, $\mathrm{LVP}^+(Q^\infty) = \mathrm{LVP}(Q^\infty)$.

Grammar:

$$S \to AT$$

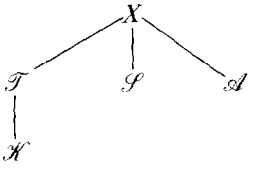
$$T \to K \mid KS$$

Applying the algorithm gives:

	$<$	*Sorts*	F	
Step 1				
	$\mathscr{K}, \mathscr{S}, \mathscr{T}, \mathscr{A} < X$	$\mathscr{S}, \mathscr{T}, \mathscr{A}, \mathscr{K}, X$	$A:\mathscr{A}$	
			$K:\mathscr{K}$	
Step 2				
	$\mathscr{K} < \mathscr{T}$		$\_\_:\mathscr{A}\mathscr{T} \to \mathscr{S}$	
			$\_\_:\mathscr{K}\mathscr{S} \to \mathscr{T}$	
Step 3				
			$\_\_: X\,X \to X$	
			$A:X$	(*1)
			$K:X$	(*2)

Step 5 (delete (*1 and (*2))

Result:



Now if y is a variable of sort $\mathscr{S}$, the rule sequence (B1), (B2), ... is generalised by the rule

(D1) $byAZ \to yAZD$

Now let $Q = \{\text{A1}, \text{A2}, \text{A3}, \text{D1}\}$. Q^{∞} is the sequence:

(C1) $dKAZ \to KAZD$

(C2) $dKAKAZ \to KAKAZD$

(C1) $dKAKAKAZ \to KAKAKAZD$

etc.

Nonvarying parts:

$d \ldots Z \to \ldots ZD.$

Language of varying parts:

$L(P) = \{(KA)^n,\ n > 0\}.$

Grammar:

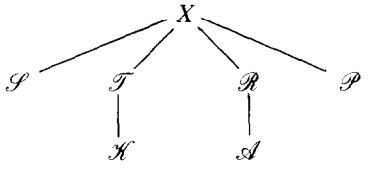
$P \to KR$

$R \to A \mid AP$

Note that we cannot use S as our sort symbol. We use P instead. However, we can reuse the sorts $\mathscr{A}, \mathscr{K}$ from before. Applying the algorithm gives the following:

	$<$	*Sorts*	F
Step 1:			
	$\mathscr{A}, \mathscr{K}, \mathscr{P}, \mathscr{R} < X$	$\mathscr{P}, \mathscr{R}, \mathscr{A}, \mathscr{K}, X$	$A:\mathscr{A}$
			$K:\mathscr{K}$
Step 2:			
	$\mathscr{A} < \mathscr{R}$		$\_\_:\mathscr{K}\mathscr{R} \to \mathscr{P}$
			$\_\_:\mathscr{A}\mathscr{P} \to \mathscr{R}$
Step 3:			
	$\mathscr{S}, \mathscr{T} < X$	$\mathscr{S}, \mathscr{T}$	$\_\_:\mathscr{A}\mathscr{T} \to \mathscr{S}$
	$\mathscr{K} < \mathscr{T}$		$\_\_:\mathscr{K}\mathscr{S} \to \mathscr{T}$
			$\_\_:XX \to X$

Result:



Now if x is a variable of sort $\mathscr{P}$, the sequence (C1), (C2), ... is generalised by the rule

(D2) $dxZ \to xZD$

The final rule set is:

(A1) $Kb \to dK$,

(A2) $Ad \to bA$,

(A3) $bAZ \to AZd$,

(D1) $byAZ - yAZD$,

(D2) $dxZ \to xZD$.

The initial rule set is terminating; so, our final rule set is complete.

Example 7.7. The final example comes from the area of reasoning about concurrent systems and is similar to examples in [21], where rewriting systems for weak bisimulation equivalence are considered. The following infinite set of rules is derived using the Knuth–Bendix completion algorithm; the operators have been renamed for simplicity of presentation.

Sorts: T

Operators:

$c:T$

$f:TT \to T$

$g:TT \to T$

$h:TT \to T$

Rules ($x_1, x_2, x_3, \ldots, y$ are variables of sort T):

(Q1) $g(c, y) \to c$

(Q2) $g(f(x_1, c), y) \to f(x_1, c)$

(Q3) $g(h(x_1, c), y) \to h(x_1, c)$

(Q4) $g(f(x_1, f(x_2, c)), y) \to f(x_1, f(x_2, c))$

(Q5) $g(f(x_1, h(x_2, c)), y) \to f(x_1, h(x_2, c))$

(Q6) $g(h(x_1, f(x_2, c)), y) \to h(x_1, f(x_2, c))$

(Q7) $g(h(x_1, h(x_2, c)), y) \to h(x_1, h(x_2, c))$

(Q8) $g(f(x_1, f(x_2, f(x_3, c))), y) \to f(x_1, f(x_2, f(x_3, c)))$

(Q9) $g(f(x_1, f(x_2, h(x_3, c))), y) \to f(x_1, f(x_2, h(x_3, c)))$
etc.

Considering (Q1), (Q2), ... as Q^∞, let the varying positions be $[\langle 1.1\rangle, \langle 1\rangle]$. A grammar for $\mathrm{LVP}^+(Q^\infty)$ is

$$S \to c \mid f(x, S) \mid h(x, S).$$

The result of applying our algorithm is the enriched signature:

Sorts: $T, \mathscr{S}$.

Ordering: $\mathscr{S} < T$

Operators:

$c: \mathscr{S}$

$f: T\mathscr{S} \to \mathscr{S}$

$f: TT \to T$

$h: T\mathscr{S} \to \mathscr{S}$

$h: TT \to T$

$g: TT \to T$

with the generalising rule

$$g(z, y) \to z$$

where z is a variable of sort $\mathscr{S}$ and y is a variable of sort T.

8. Related work

Several authors have worked or are working on the problems of collecting terms as a grammar or language, characterising critical pairs, and solving divergence in Knuth–Bendix completion. We will survey these, but first we mention some of the known methods of removing the sort-decreasingness (compatibility) requirement from order-sorted term rewriting, which we require for general application of our algorithm. At the end of this section we will prove that our approach complements three of those approaches listed in Section 8.4.

8.1. Order-sorted term rewriting without sort-decreasingness

Our preferred approach here is the idea of dynamically sorted term rewriting ([32, 9, 33]) in which equations also carry sort information and the (syntactic) least sort of a term is only an upper bound on its semantic sort. The idea of this is to allow equational reasoning in which replacement of "equals for equals" is always allowed, without sort constraints. Knuth–Bendix completion in this framework has been shown [32] to differ little from the ordinary version. Full technical details of the method appear in [33] including the critical pairs lemma mentioned in Section 2. Chen and Hsiang [3] have discovered the same idea independently.

Gallier and Isakowitz ([10, 17]) have worked on the semantics of equational reasoning without sort constraints.

Recent work of Comon [6] gives an alternative approach. He investigates Knuth–Bendix completion in the framework of rewriting with membership constraints [30] and gives deduction rules for completion and unification over a fragment of second-order logic in which a critical pairs lemma can be proved. This fragment of second-order logic is large enough to contain order-sorted equational logic. Powerful though this idea is, it lacks the conceptual clarity of dynamic sorting.

8.2. Term sets regarded as a language

Comon [5] uses the idea of describing a collection of terms as a grammar and then transforming the grammar into a signature, but in a more restricted way than our algorithm (for example, Comon is concerned only with ground terms), and for a very different purpose. Namely, his aim is the proof of inductive theorems from the original signature, whereas we specifically avoid adding such theorems to our equational theory (cf. Theorem 6.11).

8.3. Characterising critical pairs

Hermann [13, 14, 15] characterises two sources of divergence, forward and backward crossed rewrite systems, by considering the structure of rewrite systems and critical pairs. He also suggests several "empirical" methods for avoiding divergence.

These include changing the orientation of rules or the termination ordering (backtracking), dividing chains in crossed systems (splitting), or enriching systems with new rules which are inductive theorems. However, there is no method for deriving the appropriate theorems.

Sattler-Klein [25] also investigates the part played by critical pairs in divergence, but without specific reference to the problem of solving divergence. By coding primitive recursive functions into string rewriting systems, she produces some quite sophisticated divergence patterns.

8.4. Methods of solving divergence of Knuth–Bendix completion

The heuristics of backtracking and splitting were mentioned in the preceding section. These solve a very restricted class of problems, and have in any case not been thoroughly formalised.

Schmidt-Schauss [26] solves our Example 7.3 using term declarations, which explicitly collect terms with some common property into a named sort. This is done by inspection. The method used could be applied to solve divergence in more general cases. This method allows generalisation of a wider class of languages than regular tree languages, and so solves a wider class of examples than we can handle. The drawback is that term declarations make a signature untidy and rewriting harder. No implementation of term rewriting with term declarations is mentioned by [16] or is known to the present authors.

Kirchner [19] uses meta-rewriting with meta-variables and meta-rules to solve divergence. Kirchner and Hermann [20] provide a link between this approach and the work of Hermann on critical pairs, and proposes an automatic method for moving from a characterisation of the critical pairs in the divergent sequence to (in some cases) a solution using meta-rewriting. This is the only approach to the problem so far which covers the entire procedure from detection to solution of divergence. However, this approach suffers from the difficulties of meta-rewriting, not the least that implementations are not readily available, and is restricted to the class of sequences formed by forward and backward crossed rewrite systems – [25] has shown that there are many other types of divergence. In addition, this method has difficulty handling examples of divergence arising from one rule systems. Chen et al. ([4], see later) state that use of meta-rewriting in this way merely transfers the problem of finitely representing an infinite set of terms from one level to another, without solving it.

Thomas and Jantke [28, 29] and Lange [23] and Lange and Jantke [24] also attempt to replace an infinite set of rules with a finite set that is a conservative extension of the infinite set. The key idea of their approach is to use inductive inference techniques [1] for synthesising generalisations of infinite sets of rules from a presentation of only finite portions of the sets. Thomas and Jantke [29] present several different definitions of generalisation and some algorithms for synthesising these generalisations of sets of rules. Often, however, generalisation is not possible and the underlying signature and theory must be enriched. An approach to introducing

auxiliary operators and sorts in order to allow generalisation is described in [23, 28]. This is perhaps the most naturalistic approach, but is severely limited by being unable to solve examples where the number of different variables present in a rule is infinite in the limit. Also the method gives no guarantee of a complete system.

A powerful improvement using the same basic idea (counting occurrences of operators using a separate copy of the natural numbers) is proposed by Chen et al. [4]. Their method, which they call recurrence-terms, explicitly codes up infinite sets of terms or rules inside a special operator. Unfolding of this recurrence-term (to recover the original set) is deterministic up to the naming of variables. (This idea is an improvement on [28] and [23] as it handles some sequences of rules in which (in the limit) rules contain infinitely many different variables.) Use of this method involves going beyond ordinary term rewriting into recurrence-rewriting. A matching algorithm is given so that rules including recurrence-terms can rewrite ordinary terms. The procedure given for deducing recurrence-terms is inadequate, although they claim that more elaborate methods have been designed. One problem with this method is that unification involving recurrence-terms may prove to be very difficult, so while a solution to an instance of divergence may be economically expressed its use in an ongoing Knuth–Bendix completion procedure may be limited.

A further improvement of the recurrence-term approach is the addition of a choice operator [31]. This extends the solving power of recurrence-terms to include the class of problems solved by our algorithm.

In none of the papers mentioned above (except [20]) is an algorithm married with a characterisation of the class of divergences which can be (partially or fully) solved.

8.5. Comparison of methods

We can prove by example that our approach is complementary to three of the major approaches outlined above (we say two methods are complementary if each method can solve examples which the other cannot solve).

We describe the class of problems solved by a method by the initials of the method's creators:

- signature enrichment (this paper) TW,
- auxiliary operators (Thomas/Jantke/Lange) TJL,
- meta-rewriting (Kirchner/Hermann) KH,
- recurrence-terms (Chen/Hsiang/Kong) CHK.

We write $A \perp B$ to show that two methods are complementary.

Theorem 8.7.

$$\text{TW} \perp \text{TJL}$$

$$\text{TW} \perp \text{KH}$$

$$\text{TW} \perp \text{CHK}$$

Proof. Proof in each case is by an example from Section 7.

Example 7.4 is not included in TJL – the auxiliary operators method solves no examples in which rules have infinitely many different variables (in the limit).

Example 7.6 is not included in KH – the meta-rewriting method solves only sequences arising from forward and backward crossed rewrite systems. It is shown in [25] that this is not such a system.

Example 7.7 is not included in CHK – recurrence terms as originally defined have no built-in choice, i.e. the method cannot solve examples in which each rule has more than one "child". We note that this defect is corrected in [31].

Examples of divergence which these methods can solve and our system cannot are easily constructed, in each case. □

Advantages of our approach

We will state once again the advantages of our algorithm, which in many cases are not shared by the methods we have mentioned above:

- application of our algorithm allows us to remain within order-sorted term rewriting – an implementation requires little extra work;
- our algorithm is entirely effective, given the appropriate input;
- our algorithm always produces a complete rewrite system on a well-defined class of examples – there are no more theorems to prove;
- our algorithm preserves the term set of the original system;
- our algorithm preserves the complexity of the original system;
- our algorithm has applications in the field of program synthesis;
- although this is subjective, we consider our algorithm to be conceptually clear and natural.

9. Future research

9.1. Applications of inductive inference

In common with most of the methods mentioned in the previous section, a problem with our method is ensuring that the input to the algorithm is of the correct form: in our case, a weakly simple tree grammar deriving the leading instances of the language of varying parts (cf. Section 4.1).

Inductive inference [1] was proposed by [23, 28, 29] as a means of processing the (infinite) set of rewrite rules derived by completion into a form suitable for input to an algorithm to solve divergence. In our case we would aim to learn a regular tree grammar.

An immediate objection to the use of inductive inference is that we require a decision procedure for a given equational theory E (why else run Knuth–Bendix completion?), while inductive inference usually provides correct identification of

languages (grammars, patterns, etc.) only in the limit. However, this is not a real objection, as we can apply Knuth–Bendix completion (KBC) to any guess the inference procedure makes (to prevent under-generalisation – the guess is incorrect unless KBC converges) and check guesses in the equational theory (to prevent over-generalisation – the guess is incorrect if any consequence of the guess is not provable in E). Thus, we may be able to correctly identify the grammar after a finite number of steps.

Our other requirements are that

- our inference proceeds on positive data only, i.e. while all examples of rules in the divergent sequence are presented to the inference procedure at some point, no counter-examples are ever identified;
- a grammar for the language is returned, and not some other representation of the language.

It is known [12] that the full class of regular tree languages cannot be learned under such circumstances. However, special cases of regular tree languages have been shown to be learnable under some circumstances [34], and we believe that learning with the additional checking provided by the equational theory (as mentioned above) will prove to be more powerful than unassisted learning.

The authors are working on an inductive inference algorithm to dovetail with the algorithm presented in this paper.

9.2. Extensions to other classes of languages

Comon [5] has shown that an order-sorted signature is nothing but a regular tree automaton and, hence, that the term set of a signature (or of any given sort) must be a regular tree language. This would appear to suggest that our algorithm cannot possibly be extended to any wider class of languages, but this assumption is incorrect. There are some cases of nonregular tree languages which we may be able to deal with.

We have already mentioned (Section 8) that the term declaration method solves a wider class of problems than the method in this paper. However, we have assumed the use of rewriting with dynamic sorting [32] throughout this paper and dynamic sorting effectively includes sort declarations [33] – as equations (rules) carry sort information, an equation which contributes nothing to the equational theory is exactly a sort declaration. For example, an equation between a term t and a "junk" term of sort S, say (a term introduced to the signature in order to appear only in this equation), is just a sort declaration $t{:}S$. Thus, we may without difficulty extend our method to that class of tree languages accepted by signatures with sort declarations.

Another of the limitations of our method is that it can handle no examples in which the varying parts contain repeated instances of the same variable. We can overcome this in some cases if we relax our requirement that our new signature has the same

term set as the old and introduce operators in order to rename terms. Consider the following example.

Sorts: A

Operators:

$c:A$

$f:AAA\rightarrow A$

Rules ($x_1, x_2, x_3, \ldots$, are variables of sort A):

$f(c, x2, f(x1, x, x)) \rightarrow c$

$f(c, x2, f(x3, x4, f(x1, x, x))) \rightarrow c$

$f(c, x2, f(x3, x4, f(x5, x6, f(x1, x, x))) \rightarrow c$

etc.

This sequence cannot be generalised in its present form by our method because of the repeated occurrence of the variable x in every varying part. If we introduce the new sort $S<A$, the new operator $h:AA\rightarrow S$, and the new rule $f(x1, x, x)\rightarrow h(x1, x)$, which remains within a conservative extension of the original equational theory, the sequence becomes

$f(c, x2, h(x1, x)) \rightarrow c$

$f(c, x2, f(x3, x4, h(x1, x))) \rightarrow c$

$f(c, x2, f(x3, x4, f(x5, x6, h(x1, x))) \rightarrow c$

etc.

The language of varying parts $\mathrm{LVP}(Q^{\infty})$ is

$$\{h(x1, x),\ f(x3, x4, h(x1, x)),\ f(x3, x4, f(x5, x6, h(x1, x))), \ldots\}.$$

$\mathrm{LVP}^{+}(Q^{\infty})$ is a regular tree language and a generalising solution is given by

Sorts: A, S

Ordering: $S<A$

Operators:

$c:A$

$f:AAA\rightarrow A$

$f:AAS\rightarrow S$

$h:AA\rightarrow S$

with the generalising rule

$f(c, x, y) \rightarrow c$

This rule, when added to the following two rules, where $x, x1$ are variables of sort A and y is a variable of sort S, gives a complete system:

$$f(x1, x, x) \to h(x1, x)$$

$$h(c, y) \to c$$

The set of cases in which techniques like this are applicable to expand the class of examples we can solve is under investigation.

9.3. Combination of methods

An interesting approach is to use a combination of methods, for example, signature enrichment and auxiliary operators [23, 28, 29]. Since these two methods are complementary (each solves some examples the other cannot solve, Theorem 8.7) it is reasonable to expect that a combination of the two methods will be more powerful than either method alone. For example, a combined approach could be used to solve a problem with infinitely many variables and one or more occurrences in which a "counting" context is required.

Exactly which of the methods mentioned in Section 8 are suitable for combination in this way requires investigation.

10. Conclusions

Term rewriting is a powerful proof tool for reasoning about algebraic specifications. In practice, many algebraic specifications give rise to infinite complete sets of rules.

We present an effective algorithm which solves some cases of divergence in the Knuth–Bendix completion algorithm, starting from a grammar characterising the infinite rule set. We replace the infinite set of rewrite rules by a finite complete set by enriching the original (order-sorted) signature with new sorts and new operator arities, while remaining within a conservative extension of the original system, and within the original term set. The complexity of the new rewriting system is no worse than that of the original system. We characterise the class of examples to which this approach is applicable by regular tree languages. Our algorithm effectively constructs tree automata which recognise these languages. Our approach is distinguished from others by this characterisation and the fact that we preserve the original term set. We prove that our approach is complementary to some others in the literature.

The algorithm which we have given is, as we have seen, only a part of the full process of transforming an infinite set of rewrite rules R (or more accurately a divergent case of Knuth–Bendix completion) into a finite complete set of rules. We have shown that if we enrich the original signature Σ in an appropriate way then at least in some cases we arrive at a signature Σ' in which there exists a complete set of rules, with respect to the original equational theory, which may not be true in Σ. We find this rule

set effectively. Obviously, this approach, or any other, is possible only if the word problem under consideration is decidable and, therefore, it is only applicable in an enumerable number of cases.

Acknowledgment

We thank Hubert Comon and anonymous referees for their comments on an earlier version of this paper.

References

[1] D. Angluin and C.H. Smith, A survey of inductive inference: theory and methods, *Comput. Surveys* 15(3) (1983) 237–269.
[2] M.A. Ardis, Data abstraction transformations, Tech. Report TR-925, Univ. of Maryland, MA, 1980.
[3] H. Chen and J. Hsiang, Order-sorted equational specification and completion, preprint.
[4] H. Chen, J. Hsiang and H.-C. Kong, On finite representations of infinite sequences of terms, in: M. Okada, ed., *Proc. 2nd Internat. Workshop on Conditional and Typed Rewriting Systems*, Montreal, Lecture Notes in Computer Science, Vol. 516 (Springer, Berlin, 1990) 100–114.
[5] H. Comon, Inductive proofs by specification transformation, in: N. Dershowitz, ed., *Proc. Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Vol. 355 (Springer, Berlin, 1989) 76–91.
[6] H. Comon, Completion of rewrite systems with membership constraints, Rapporte de Recherche no. 699, Laboratoire de Recherche en Informatique, Université de Paris-Sud, 1991.
[7] N. Dershowitz, Synthesis by completion, in: *Proc. 9th Internat. Joint Conference on Artificial Intelligence* (Los Angeles, CA, 1985) 208–214.
[8] N. Dershowitz and E. Pinchover, Inductive synthesis of equational programs, in: *Proc. of AAAI-90*, 1990.
[9] A.J.J. Dick and P. Watson, Order-sorted term rewriting, *Comput. J.* **34**(1) (1991) 16–19.
[10] J.H. Gallier and T. Isakowitz, Rewriting in order-sorted equational logic, in: R.A. Kowalski and K. Bowen, eds., *Logic Programming*, also in: *Proc. 5th Internat. Conf. Symp.*, Vol. 1 (MIT Press, Cambridge, MA, 1988) 280–294.
[11] F. Gecseg and M. Steinby, Tree automata (Akademiai Kiado, Budapest, 1984).
[12] E. Gold, Language identification in the limit, *Inform. and Control* **10** (1967) 447–474.
[13] M. Hermann, Vademecum of divergent term rewriting systems, CRIN Report 88-R-082, Centre de Recherche en Informatique de Nancy, 1988.
[14] M. Hermann, Crossed term rewriting systems, CRIN Report 89-R-003, Centre de Recherche en Informatique de Nancy, 1989.
[15] M. Hermann, Chain properties of rule closures, *Formal Aspects Comput.* **2** (1990) 207–225.
[16] M. Hermann, C. Kirchner and H. Kirchner, Implementations of term rewriting systems, *Comput. J.* **34**(1) (1991) 20–33.
[17] T. Isakowitz and J.H. Gallier, Congruence closure in order-sorted algebra, Tech. Report, Computer and Information Sciences Department, University of Pennsylvania, Philadelphia, PA, 1987.
[18] D. Knuth and P. Bendix, Simple word problems in universal algebra, in: J. Leech, ed., *Computational Problems in Abstract Algebra* (Pergamon Press, Oxford, 1970).
[19] H. Kirchner, Schematization of infinite sets of rewrite rules generated by divergent completion processes, *Theoret. Comput. Sci.* **62** (1989) 303–332.
[20] H. Kirchner and M. Hermann, Meta-rule synthesis from crossed rewrite systems, in: M. Okada, ed., *Proc. Second Internat. Workshop on Conditional and Typed Rewriting Systems*, Montreal, Lecture Notes in Computer Science, Vol. 516 (Springer, Berlin, 1990) 143–154.

[21] C. Kirkwood and K. Norrie, Some experiments using term rewriting techniques for concurrency, Tech. Report CSD-TR-623, Royal Holloway and Bedford New College, Univ. of London, 1990.
[22] G.A. Kucherov, On relationship between term rewriting systems and regular tree languages, in: R.V. Book, ed., *Proc. Rewrite Techniques and Applications*, Como (Italy), Lecture Notes in Computer Science, Vol. 488 (Springer, Berlin, 1991) 374–385.
[23] St. Lange, Towards a set of inference rules for solving divergence in Knuth–Bendix completion, in: K.P. Jantke, ed., *Proc. Analogical and Inductive Inference '89*, GDR, Lecture Notes in Artificial Intelligence, Vol. 397 (Springer, Berlin, 1989) 304–316.
[24] St. Lange and K.P. Jantke, Inductive completion for transformation of equational specifications, in: H. Ehrig, K.P. Jantke, F. Orejas and H. Reichel, eds., *Proc. 7th Workshop on Specification of Abstract Data Types, 1990*, Lecture Notes in Computer Science, Vol. 534 (Springer, Berlin, 1991) 117–140.
[25] A. Sattler-Klein, Divergence phenomena during completion, in: R.V. Book, ed., *Proc. Rewrite Techniques and Applications*, Como (Italy), Lecture Notes in Computer Science, Vol. 488 (Springer, Berlin, 1991) 374–385.
[26] M. Schmidt-Schauss, Computational aspects of an order-sorted logic with term declarations, Lecture Notes in Computer Science, Vol. 395 (Springer, Berlin, 1989).
[27] G. Smolka, W. Nutt, J.A. Goguen and J. Meseguer, Order-sorted equational computation, in: H. Ait-Kaci and M. Nivat, eds., *Resolution of Equations in Algebraic Structures, Vol. 2: Rewriting Techniques* (Academic Press, New York, 1989) 297–367.
[28] M. Thomas and K.P. Jantke, Inductive inference for solving divergence in Knuth–Bendix completion, Tech. Report 88/R6, University of Glasgow, 1988.
[29] M. Thomas and K.P. Jantke, Inductive inference for solving divergence in Knuth–Bendix completion, in: K.P. Jantke, ed., *Proc. Analogical and Inductive Inference '89*, GDR, Lecture Notes in Artificial Intelligence, Vol. 397 (Springer, Berlin, 1989) 288–303.
[30] Y. Toyama, Membership conditional term rewriting systems, *Trans. IEICE* **72**(11) (1989) 1224–1229.
[31] P. Watson, The expressive power of recurrence-terms (in preparation).
[32] P. Watson and A.J.J. Dick, Least sorts in order-sorted term rewriting, Tech. Report TR-CSD-606, Royal Holloway and Bedford New College, Univ. of London, 1989
[33] P. Watson and B.M. Matthews, Term rewriting with dynamic sorting (in preparation).
[34] T. Yokomori, Learning context-free languages efficiently – a report on recent results in Japan, in: K.P. Jantke, ed., *Proc. Internat. Workshop Analogical and Inductive Inference '89*, Lecture Notes in Artificial Intelligence, Vol. 397 (Springer, Berlin, 1989) 104–123.