Algebraic and Logic Programming (ALP'92)
Volterra, Italy, Sept. 1992, Springer LNCS Vol. 632, pages 143-157.

Definitional Trees

Sergio Antoy*

Department of Computer Science
Portland State University
Portland, Oregon 97207-0751

Abstract. Rewriting is a computational paradigm that specifies the actions, but not the control. We introduce a hierarchical structure representing, at a high level of abstraction, a form of control. Its application solves a specific problem arising in the design and implementation of inherently sequential, lazy, functional programming languages based on rewriting. For example, we show how to extend the expressive power of $Log(F)$ and how to improve the efficiency of an implementation of *BABEL*. Our framework provides a notion of degree of parallelism of an operation and shows that the elements of a necessary set of redexes are related by an *and-or* relation. Both concepts find application in parallel implementations of rewriting. In an environment in which computations can be executed in parallel we are able to detect sequential computations in order to minimize overheads and/or optimize execution. Conversely, we are able to detect when inherently sequential computations can be executed in parallel without performing unnecessary rewrites.

1 Introduction and Motivations

Rewrite systems are the underlying model of computation of an increasingly large number of functional, equational, and/or multi-paradigm programming languages. A computation in these languages is a finite or infinite sequence $t_0, t_1, \ldots$ of terms in which the term t_{i+1} is obtained by rewriting some subterms of t_i. Rewrite systems specify the actions, but not the control, that is, which redexes of a term should be rewritten. This information is embodied in a rewrite strategy.

For the sake of both the efficiency and the termination of a computation we are generally interested in parallel strategies, although parallelism is not theoretically necessary for weakly orthogonal rewrite systems, the class considered in this note [10]. Two emerging approaches to efficient parallel rewriting emphasize respectively "quantity" and "quality". The first one attempts to rewrite simultaneously a large, possibly maximal, set of redexes of a term [6,11]. This approach has greater potential, but has the drawback of requiring specialized architectures. The second approach is concerned with the trade-off between the generality of a class of rewrite systems and the efficiency of a normalizing strategy for systems in the class. Our effort belongs to this category [7,8,10,13,20–23].

* Supported by the National Science Foundation Grant No. CCR-8908565.

We introduce a hierarchical structure, called definitional tree, containing rewrite rules and show some of its applications to the control aspects of a rewrite strategy. First, we consider a class of systems whose rules are containable in a simplified, non-parallel form of definitional trees. This class is interesting in its own right, since it properly contains the rewrite systems underlying both DF^* [17–19] and an implementation of *BABEL* [14–16]. Our results extend the expressive power of the first and improve an implementation of the second.

Then, we consider a more general, parallel form of definitional trees which are able to contain the rules of any weakly orthogonal rewrite system. These trees can be used for computing necessary sets of redexes [22], and showing that necessary sets have an internal structure in the form of an *and-or* relation among their elements. We discuss how a parallel rewrite strategy may take advantage of this knowledge. Our reasoning is made possible by the level of abstraction captured in a definitional tree, which is higher than that of corresponding concepts discussed in [22].

2 Preliminaries and Notation

We consider a term rewriting system $\mathcal{R}$ with a many-sorted signature partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{D}$ of *(defined) operations*. For each sort s, we assume an arbitrary, but fixed ordering, called *standard*, among the constructors of s. $\mathcal{X}$ denotes a set of sorted variables. Variables are denoted by upper case letters or, when anonymous, by the symbol "_". For any signature Σ, $T(\Sigma)$ is the set of terms built over Σ. Any term referred to in this note type checks. The leading symbol or principal functor of a term t is called the *root* of t. The elements of $T(\mathcal{C})$, $T(\mathcal{C} \cup \mathcal{X})$, and $T(\mathcal{C} \cup \mathcal{D})$ are respectively called *values*, *constructor terms*, and *ground terms*. We call *non-value* any ground term different from a value. A term t of the form $f(x_1, \ldots, x_n)$, where f is an operation symbol and the arguments are constructor terms, is called an *f-rooted constructor term*. The rules in $\mathcal{R}$ are characterized by having an operation-rooted constructor term as the left side, that is, $\mathcal{R}$ follows the constructor discipline [21].

An *occurrence* is a path identifying a subterm in a term. An occurrence is a possibly empty string of dot-separated integers or occurrences, i.e., the symbol "." is overloaded. The empty occurrence of a term t is denoted by Λ and identifies t itself. The occurrence $p{\cdot}i$, for some occurrence p and positive integer i, identifies the i-th argument of the subterm of t identified by (at) p. For terms t and t', occurrence p, and variable X, t/p denotes the subterm of t at p, and $t[p \leftarrow t']$ denotes occurrence substitution.

3 Sequential Definitional Trees

In this section we define a hierarchical structure, called a *definitional tree*, that contains a set of rewrite rules. We are interested in operations, called *inductively sequential*, whose entire set of defining rules can be stored in this structure. The class of the inductively sequential operations is limited and we will later

extend this class by extending the concept of a definitional tree. In this section we also prove some simple properties of rewrite systems, called also *inductively sequential*, whose operations are all inductively sequential. Our interest in these systems is motivated by the fact that computations in inductively sequential systems can be executed sequentially, that is, without any inherent parallelism, and by the fact that the class of inductively sequential systems is larger than those underlying certain functional lazy programming languages. We will prove these claims in following sections.

The symbols *branch*, *rule*, and *exempt*, appearing in the next definition, are uninterpreted functions used to classify the nodes of the tree. A *pattern* is an operation-rooted constructor term contained in each node of a tree.

Definition 1. $\mathcal{T}$ is a *partial definitional tree*, or *pdt*, if and only if one of the following cases holds:

- $\mathcal{T} = branch(\tau, o, \bar{\mathcal{T}})$, where τ is a pattern, o is the occurrence of a variable of τ, the sort of τ/o has constructors $c_1, \ldots, c_k$ in standard ordering, $\bar{\mathcal{T}}$ is a sequence $\mathcal{T}_1, \ldots, \mathcal{T}_k$ of *pdt*s such that for all i in $1, \ldots, k$ the pattern in the root of $\mathcal{T}_i$ is $\tau[o \leftarrow c_i(X_1, \ldots, X_n)]$, where n is the arity of c_i and $X_1, \ldots, X_n$ are new variables.
- $\mathcal{T} = rule(\tau, l \to r)$, where τ is a pattern and $l \to r$ is a rewrite rule such that $l \equiv \tau$.
- $\mathcal{T} = exempt(\tau)$, where τ is a pattern.

Definition 2. $\mathcal{T}$ is a *definitional tree* of an operation f if and only if $\mathcal{T}$ is a *pdt* with $f(X_1, \ldots, X_n)$ as the pattern argument, where n is the arity of f and $X_1, \ldots, X_n$ are new variables.

Definition 3. We call *inductively sequential* an operation f of a rewrite system $\mathcal{R}$ if and only if there exists a definitional tree $\mathcal{T}$ of f such that the rules contained in $\mathcal{T}$ are all and only the rules defining f in $\mathcal{R}$. We call *inductively sequential* a rewrite system $\mathcal{R}$ if and only if any operation of $\mathcal{R}$ is inductively sequential.

Exempt nodes are present in a tree of an incompletely defined operation. Patterns do not need explicit representation in a definitional tree. However, we will keep them around when their presence simplifies the presentation of our ideas.

For example, consider the rules defining the operations "less than or equal to" and "integer division by 2" defined on the natural numbers, whose constructors in standard ordering are 0 and s.

$$\begin{array}{rr} 0 \leq \_ \to true & half(0) \to 0 \\ s(\_) \leq 0 \to false & half(s(0)) \to 0 \\ s(X) \leq s(Y) \to X \leq Y & half(s(s(X))) \to s(half(X)) \end{array} \tag{1}$$

Corresponding definitional trees are:

$$\begin{array}{l} branch(1, \langle rule(0 \leq \_ \rightarrow true), \\ \qquad branch(2, \langle rule(s(\_) \leq 0 \rightarrow false), rule(s(X) \leq s(Y) \rightarrow X \leq Y)\rangle\rangle)) \\ branch(1, \langle rule(half(0) \rightarrow 0), \\ \qquad branch(1{\cdot}1, \langle rule(half(s(0)) \rightarrow 0), rule(half(s(s(X))) \rightarrow s(half(X)))\rangle\rangle)) \end{array} \tag{2}$$

Figure 1 shows the same trees in pictorial representation. An operation may have distinct definitional trees. We do not address this problem *per se*. In the final part of this note we will show an operation with two "non-isomorphic" definitional trees and discuss some implications of this fact.

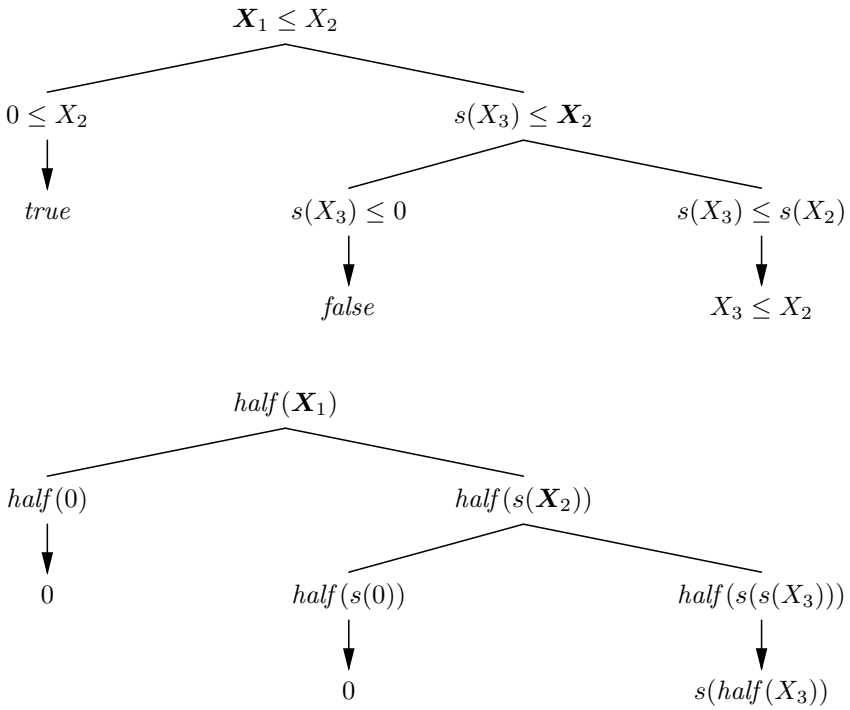

Fig. 1. Pictorial representation of the definitional trees of the operations defined in display (1). The boldfaced symbol in the pattern argument of a *branch* node is the root of the subterm identified by the occurrence argument. The rule argument of a *rule* node is shown "vertically" and the pattern argument is "merged" with the rule's left side.

Definition 4. We call *complete* an operation f if and only if there exists a definitional tree of f with no *exempt* nodes. We call *complete* an inductively sequential rewrite system $\mathcal{R}$ if and only if any operation of $\mathcal{R}$ is complete.

We now state two simple results about inductively sequential rewrite systems. We follow Klop's terminology [12] and call *orthogonal* a left-linear, non-overlapping rewrite system, and *weakly orthogonal* a left-linear system whose critical pairs are all trivial, that is, if $\langle t, t' \rangle$ is a critical pair, then t is syntactically equal to t'. For related concepts and terminology see also [5,8,21].

Lemma 5. *Any inductively sequential rewrite system $\mathcal{R}$ is orthogonal.*

Lemma 6. *If an inductively sequential rewrite system is complete, then a ground term is a normal form if and only if it is a value.*

4 Sequential Normalization

In this section we use definitional trees to compute normal forms. We show how to find a needed redex [8] in a term and consequently how to obtain a normalizing rewrite strategy. Our motivation is twofold: (1) we use these results to extend the expressive power of some inherently sequential, lazy, functional programming languages based on rewriting and/or to improve the efficiency of their implementations and (2) we apply these results to the analysis of necessary sets of redexes used in a parallel normalizing strategy applicable to a larger class of rewrite systems.

The following intuitive, informal description of our strategy might help understanding the considerable details of its definition. Our strategy is based on a function, λ, which takes a non-value t and returns an occurrence o of t. The subterm of t at o must be eventually rewritten in order to rewrite t to a value. The function λ is defined by means of another function, φ, which takes a ground f-rooted term t, for some operation f, and a definitional tree $\mathcal{T}$ of f, and through a traversal of $\mathcal{T}$ returns the occurrence sought by λ.

Definition 7. Let t be a non-value.

$$\lambda(t) = \begin{cases} o{\cdot}\lambda(t/o) & \text{if } t \text{ is constructor-rooted and} \\ & \quad t/o \text{ is any outermost operation-rooted subterm of } t; \\ \varphi(t, \mathcal{T}_f) & \text{if } t \text{ is } f\text{-rooted, for some operation } f, \text{ and} \\ & \quad \mathcal{T}_f \text{ is a definitional tree of } f. \end{cases} \tag{3}$$

$$\varphi(t, \mathcal{T}) = \begin{cases} \Lambda & \text{if } \mathcal{T} = \mathit{rule}(l \to r) \text{ or } \mathcal{T} = \mathit{exempt}; \\ \varphi(t, \mathcal{T}_i) & \text{if } \mathcal{T} = \mathit{branch}(o, \langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle) \text{ and } t/o \text{ has sort } s \text{ and root the} \\ & \quad i\text{-th constructor (in standard ordering) of } s;^{1} \\ o{\cdot}\lambda(t/o) & \text{if } \mathcal{T} = \mathit{branch}(o, \ldots) \text{ and } t/o \text{ is operation-rooted.} \end{cases}$$

We can use the equations of Definition 7 to compute $\lambda(t)$, for some non-value t, by regarding the left side of each equation as a procedure declaration and the right side as the procedure's body. From this *procedural interpretation* of λ, the application of λ to some ground operation-rooted term generates a sequence of

[0] When φ is called from λ, t is an instance of the pattern of $\mathcal{T}_i$.

recursive calls to φ and/or λ consisting of "segments", each beginning with a call to λ followed by one or more calls to φ. The structure of a typical sequence is:

$$\lambda(t_0), \varphi(t_0, \mathcal{T}_{01}), \varphi(t_0, \mathcal{T}_{02}), \ldots, \lambda(t_1), \varphi(t_1, \mathcal{T}_{11}), \varphi(t_1, \mathcal{T}_{12}), \ldots \tag{4}$$

For our purposes, only the arguments of λ and φ in this sequence are interesting. Since our definition of λ and φ is non procedural, some care is required to define the sequence we desire.

Definition 8. If $\mathcal{R}$ is an inductively sequential rewrite system and t is a ground operation-rooted term, then the *λ-sequence S generated by t* is defined as follows:

- Each element of S is an input to either λ or φ. Inputs to λ and φ are respectively called λ-elements and φ-elements of S.
- The first element of S is the λ-element t.
- If some x is a y-element of S, where y is either λ or φ, then $y(x)$ is an instance of the left side of an equation of Definition 7. The instance of the corresponding right side is either Λ, in which case x is the last element of S, or contains a subexpression $y'(x')$, where y' is either λ or φ and x' is the input to y', in which case x' is the y'-element of S following x.

Any subsequence of S beginning with a λ-element x and ending either just before the following λ-element or at the end of the sequence if no λ-element follows x in S is called a *segment* of S.

Lemma 9. *If $\mathcal{R}$ is an inductively sequential rewrite system and t is a non-value, then λ is defined on t.*

Proof. Without loss of generality we consider the case in which t is operation-rooted. We ask the reader to verify that if λ or φ have an input x in their respective domains, then any recursive application of λ or φ generated by the input x has an input in the function's respective domain. By hypothesis, the initial input to λ is in the domain of λ, thus we show only that any λ-sequence S generated by t is finite. We prove that each segment of S is finite and that there are only a finite number of segments in S. If for some integers i and j, $\langle t_i, \mathcal{T}_{ij}\rangle$ and $\langle t_i, \mathcal{T}_{i(j+1)}\rangle$ are two consecutive φ-elements of a segment of S, then $\mathcal{T}_{i(j+1)}$ is a proper subtree of $\mathcal{T}_{ij}$. Thus, the segment is finite. Likewise, if for some integer i, t_i and t_{i+1} are the λ-elements of two consecutive segments of S, then t_{i+1} is a proper subterm of t_i. Thus, S is finite and $\lambda(t)$ is defined.

Rather loosely speaking, we now prove that in order to normalize a term t we must eventually rewrite the subterm of t at occurrence $\lambda(t)$. The difficulty lies in formalizing "eventually." The formalization of this idea requires the concepts of *descendant* (or *residual*) of a term and *needed* redex. A formal definition, quite technical, of *descendants* of a subterm s in a term t after another (not necessarily distinct) subterm of t is rewritten, can be found in [8]. A less formal, but more intuitive, formulation is proposed in [13]. Suppose we have a rewrite sequence $t \xrightarrow{*} t'$ and a subterm s of t. The descendants of s in t' are computed as follows.

Underline the root of s and perform the rewrite sequence $t \xrightarrow{*} t'$. The descendants of s in t' are the subterms of t' which have an underlined root. Then, a redex s in a term t is *needed* if in every rewrite sequence of t to normal form a descendant of s is rewritten.

Lemma 10. *If $\mathcal{R}$ is an inductively sequential rewrite system and, for some integers i and m_i, $t_i, \langle t_i, \mathcal{T}_{i1}\rangle, \langle t_i, \mathcal{T}_{i2}\rangle, \ldots, \langle t_i, \mathcal{T}_{im_i}\rangle$ is the i-th segment of the λ-sequence generated by some ground operation-rooted term t, then (1) t_i is operation-rooted and (2) for all j in $1, \ldots, m_i$, if r is a rewrite rule of $\mathcal{R}$, then r cannot rewrite any descendant of t_i at the root unless r is contained in the* pdt *$\mathcal{T}_{ij}$.*

Proof. The proof of claim (1) is a simple induction on i. The claim is assumed for the first segment, and by definition of φ, every time a new segment S begins, i.e., a recursive call to λ is generated, the argument of λ, i.e., the initial element of S, is operation-rooted. We now prove (2) by induction on j. Let f be the root of t_i, for some operation f. Base case: $j = 1$. Any descendant of t_i can be rewritten at the root only by a rewrite rule of $\mathcal{R}$ defining f. Any such rule is contained in a definitional tree $\mathcal{T}$ of f. By case 2 of the definition of λ and by definition of λ-sequence, $\mathcal{T}_{i1}$ is a definitional tree of f. Ind. case: We assume the claim for some j in $1, \ldots, m_i - 1$, and prove the claim for $j+1$. By the definitions of φ and definitional tree, the following conditions hold: (a) $\mathcal{T}_{ij} = branch(o, \langle \mathcal{T}_1, \ldots, \mathcal{T}_k\rangle)$, for some occurrence o of t_i, $k > 0$, and partial definitional trees $\mathcal{T}_1, \ldots, \mathcal{T}_k$; (b) the root of the subterm of t_i at o is some constructor c; (c) $\mathcal{T}_{i(j+1)} = \mathcal{T}_n$ for some n in $1, \ldots, k$; and (d) for all l in $1, \ldots, k$, if $l \neq n$, then the root of the subterm at occurrence o of the left side of any rule r contained in $\mathcal{T}_l$ is a constructor different from c. Thus r cannot rewrite any descendant of t_i at the root, and the claim holds for $j + 1$ also.

Theorem 11. *If $\mathcal{R}$ is an inductively sequential rewrite system and t is a ground operation-rooted term, then no descendant t'' of t can be rewritten to head normal form unless the subterm of t'' at occurrence $\lambda(t)$ has been rewritten to head normal form.*

Proof. The proof is by induction on the number of segments in the λ-sequence generated by t, which we denote $t, \langle t, \mathcal{T}_1\rangle, \ldots \langle t, \mathcal{T}_m\rangle, t', \langle t', \mathcal{T}'_1\rangle, \ldots$ Base case: There is only one segment and, by definition of φ, $\lambda(t) = \Lambda$. Since t is operation-rooted, t can be rewritten to head normal form only by rewriting a descendant of t at the root. Ind. case: The following conditions hold: (a) $t' = t/o$, for some occurrence o of t; (b) $\lambda(t') = o'$, for some occurrence o' of t'; (c) $\lambda(t) = o{\cdot}o'$; and (d) we assume the claim for t'. By Lemma 10, t'' can be rewritten at the root only by a rule contained in $\mathcal{T}_m$. By definition of definitional tree, the subterm at o in the left side of any rule contained in $\mathcal{T}_m$ is constructor-rooted. By construction, t' is operation-rooted, thus no descendant t'' of t can be rewritten by any rule contained in $\mathcal{T}_m$, hence by any rule of $\mathcal{R}$, unless the subterm of t'' at occurrence o, which is a descendant of t', is a head normal form. By the inductive hypothesis, no descendant of t' can be rewritten to head normal form, without rewriting to

a head normal form its subterm at o'. Since for any term s, occurrence p of s, and occurrence p' of s/p, the equation $(s/p)/p' = s/(p{\cdot}p')$ holds, no descendant t'' of t can be rewritten to head normal form without rewriting the subterm of t'' at occurrence $\lambda(t)$ to a head normal form.

This result directly suggests a one-step rewrite strategy that we can easily prove normalizing. Our strategy is somewhat special, since on input a term t, rather than selecting a redex of t, selects the occurrence of a subterm of t which should be rewritten, but might be irreducible. When the strategy selects the occurrence of an irreducible subterm of t, we "lose interest" in further rewriting t, since the normal form of t, if any, will contain defined symbols. We will come back to this point after Corollary 14.

Definition 12. We call *outermost-needed* the one-step rewrite strategy of an inductively sequential rewrite system that in any non-value t selects for rewriting the subterm of t at occurrence $\lambda(t)$.

Corollary 13. *If $\mathcal{R}$ is an inductively sequential rewrite system and $t \xrightarrow{+} v$ in $\mathcal{R}$, for some non-value t and value v, then $\lambda(t)$ is the occurrence of a needed redex of t.*

Proof. If t is operation-rooted, then some descendant t' of t must be rewritten at the root to normalize t. Rewriting the subterm of t' at $\lambda(t)$ (to head normal form) is necessary to rewrite t', hence t, to head normal form, hence to normalize t. Thus, in any rewrite sequence of t to normal form the subterm at occurrence $\lambda(t)$ of some descendant of t is rewritten, hence $\lambda(t)$ is the occurrence of a needed redex of t. If t is constructor-rooted, then $\lambda(t) = o{\cdot}\lambda(t/o)$, for some occurrence o of an outermost operation-rooted subterm of t. The constructor discipline ensures that t can be rewritten to normal form only if a descendant of t/o is rewritten to normal form, and the previous case applies.

Corollary 14. *If $\mathcal{R}$ is an inductively sequential complete rewrite system, then the outermost-needed strategy is normalizing.*

Proof. We first prove that for any non-value t, $\lambda(t)$ is the occurrence of a needed redex of t. By Lemma 9, $\lambda(t)$ is defined. By Theorem 11, $\lambda(t)$ is operation-rooted. By Lemma 6, the normal form of t is a value, thus by Corollary 13, $\lambda(t)$ is the occurrence of a needed redex of t. Huet and Lévy [8] prove that the repeated rewriting of some needed redex of a term t leads to the normal form of t, if it exists.

Definition 7 must be changed to make the outermost-needed strategy normalizing for incomplete systems also. The required change is:

$$\varphi(t, exempt) = o{\cdot}\lambda(t/o) \tag{5}$$

where o is the occurrence of an arbitrary outermost operation-rooted proper subterm of t. An approach suitable for applications to programming languages consists in raising an exception if $\lambda(t)$ is the occurrence of an irreducible subterm of t.

5 Parallel Definitional Trees

Inductively sequential operations are not as "powerful" as we might occasionally desire. Consider for example the following three alternative definitions of the boolean conjunction operator denoted by the infix symbol "$\vee$".

$$\begin{array}{lll} true \vee \_ \to true & \_ \vee true \to true & \_ \vee true \to true \\ false \vee Y \to Y & X \vee false \to X & true \vee \_ \to true \\ & & false \vee false \to false \end{array} \tag{6}$$

The operator "$\vee$" is inductively sequential according to the first two definitions, but it is not according to the third one known as "*parallel-or*".

We obtain a structure similar to a definitional tree for the *parallel-or* operation by "collapsing" the roots of the definitional trees of the two inductively sequential operations of display (6). Definition 15 formalizes this idea. Figure 2 pictorially represents the resulting "tree". For the sake of symmetry we have duplicated a rule in the tree of Figure 2. Also, there are two *exempt* nodes in the tree, although the operation "$\vee$" is complete. This tree is built according to the algorithm described in the proof of Theorem 19.

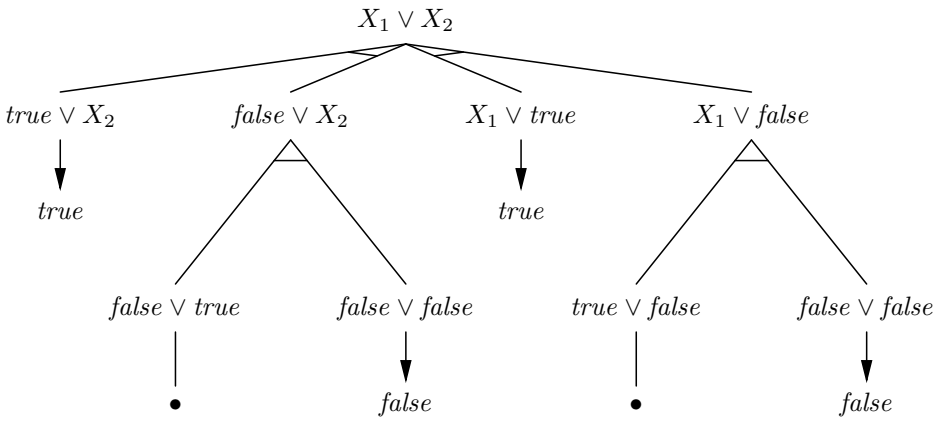

Fig. 2. Pictorial representation of a parallel definitional tree of the operation *parallel-or* defined in display (6). The arcs linking a father to children belonging to the same sequential component of the tree are joined together. The symbol "•" represents an *exempt* node.

Definition 15. $\mathcal{T}$ is a *partial parallel definitional tree*, or *ppdt*, if and only if one of the following cases holds:

$\mathcal{T} = branch(\tau, \bar{o}, \bar{\bar{T}})$, where τ is a pattern, $\bar{o}$ is a list $o_1, \ldots, o_k$ of occurrences of distinct variables of τ, for all j in $1, \ldots, k$, the sort of τ/o_j has constructors

$c_{j_1}, \ldots, c_{j_{k_j}}$ in standard ordering, and $\bar{\bar{\mathcal{T}}}$ is a sequence $\bar{\mathcal{T}}_1, \ldots, \bar{\mathcal{T}}_k$ of sequences of *ppdt*s, such that for all j in $1, \ldots, k$, $\bar{\mathcal{T}}_j = \mathcal{T}_{j_1}, \ldots, \mathcal{T}_{j_{k_j}}$ and for all i in $1, \ldots, k_j$, the pattern in the root of $\mathcal{T}_{j_i}$ is $\tau[o_j \leftarrow c_{j_i}(X_1, \ldots, X_n)]$, where n is the arity of c_{j_i} and $X_1, \ldots, X_n$ are new variables.

$\mathcal{T} = rule(\tau, l \rightarrow r)$, where τ is a pattern and $l \rightarrow r$ is a rewrite rule such that $l \equiv \tau$.

$\mathcal{T} = exempt(\tau)$, where τ is a pattern.

The *ppdt*s $\bar{\mathcal{T}}_1, \ldots, \bar{\mathcal{T}}_k$ are referred to as *sequential components* of $\mathcal{T}$.

A partial parallel definitional tree is a special case of a partial definitional tree, namely, one in which the occurrence argument of any *branch* node is a sequence of occurrences of length 1. When we want to emphasize this difference we will use the adjectives "sequential" or "parallel". When the difference is irrelevant or clear from the context we will simply use "definitional tree".

Definition 16. $\mathcal{T}$ is a *parallel definitional tree* of an operation f if and only if $\mathcal{T}$ is a *ppdt* with $f(X_1, \ldots, X_n)$ as pattern argument, where n is the arity of f and $X_1, \ldots, X_n$ are new variables.

The next results entail the concept of *subsumption ordering* [9], a quasi-ordering relation on terms denoted by the infix operator "$\leq$" and its derivatives "$<$", "$\equiv$", etc. We say that $t \leq t'$ when t' is an instance of t, that is, there exists a substitution σ such that $\sigma(t) = t'$.

Overlapping is allowed in weakly orthogonal rewrite systems. Any such system may contain some rule subsumed by some other rule. Subsumed rules are useless in the sense that their elimination does not change (the existence of) the normal form of any term. Subsumed rules do not always fit in parallel definitional trees, thus we characterize them in the next definition to eliminate them later.

Definition 17. A rule $l \rightarrow r$ in a weakly orthogonal rewrite system $\mathcal{R}$ is *useless* if and only if there exists another rule $l' \rightarrow r'$ in $\mathcal{R}$ such that $l' < l$. A rule is *useful* if and only if it is not useless.

Lemma 18. *If $\mathcal{R}$ is a weakly orthogonal rewrite system and $\mathcal{R}'$ is obtained from $\mathcal{R}$ by removing any useless rule, then, for all ground terms t and t', $t \xrightarrow{*}_{\mathcal{R}} t'$ if and only if $t \xrightarrow{*}_{\mathcal{R}'} t'$.*

Theorem 19. *If f is an operation of a weakly orthogonal rewrite system $\mathcal{R}$, then there exists a parallel definitional tree $\mathcal{T}$ of f such that the rules contained in $\mathcal{T}$ are all and only the useful rules defining f in $\mathcal{R}$.*

Proof. Our proof is constructive by means of an algorithm that on input f outputs a parallel definitional tree $\mathcal{T}$ with the characteristic we are looking for. The construction of $\mathcal{T}$ is iterative. At each iteration we add a new node to $\mathcal{T}$. The addition of a node $\mathcal{N}$ is a two-step process. First, we compute the pattern

argument τ of $\mathcal{N}$, which, except for the root of $\mathcal{T}$ whose pattern argument is always $f(X_1, \ldots, X_n)$, is determined by the father of $\mathcal{N}$. Second, we compare, according to the subsumption ordering, τ with the left sides of the useful rules defining f. The result of the comparison determines whether $\mathcal{N}$ should be a *branch*, *rule*, or *exempt* node. We now describe the construction of a node $\mathcal{N}$ given its pattern argument τ. If $\tau \equiv l$, for some rule $l \to r$, then $\mathcal{N} = rule(l \to r)$; else if $\tau < l$, for some rule $l \to r$, then $\mathcal{N} = branch(\bar{o}, \bar{\bar{\mathcal{T}}})$, where $\bar{o}$ contains all the occurrences of variables of τ and the patterns in the *ppdt*s of $\bar{\bar{\mathcal{T}}}$ are computed according to the definition of parallel definitional tree; else $\mathcal{N} = exempt$. We ask the reader to verify that the construction of $\mathcal{T}$ terminates with a parallel definitional tree of the useful rules of f.

The algorithm implicitly defined by the proof of Theorem 19 tends to build "bushy" parallel definitional trees, since a sequential component of the tree is generated for each variable of each template. These trees are useful to show that our approach to parallelism can be applied to the whole class of weakly orthogonal rewrite systems. We will discuss later how we may use definitional trees to discover or control the degree of parallelism of a computation. Building definitional trees with the above algorithm is inappropriate for these applications. Generally, we are interested in finding whether an operation has a sequential, rather than parallel definitional tree. Often, though not always, "thinner" trees seem preferable, if the computational resources available to an implementation of our strategy are limited.

6 Parallel Normalization

In this section we address the normalization problem in weakly orthogonal rewrite systems by exploiting parallel definitional trees. We reformulate parallel versions of the functions λ and φ defined in Section 4 in much the same way in which parallel trees reformulate sequential ones. The new function λ, on input a ground term t, returns a set S of occurrences of subterms of t that should be rewritten. The old function λ can be regarded as the special case of the new one in which the set S is a singleton.

We further overload the symbol "$\cdot$" as follows: the expression $o \cdot \{o_1, \ldots, o_k\}$, for some occurrence o and set of occurrences $\{o_1, \ldots, o_k\}$, for some $k \geq 0$, denotes the set of occurrences $\{o \cdot o_1, \ldots, o \cdot o_k\}$; in particular, it denotes the empty set of occurrences when $k = 0$.

Definition 20. Let t be a ground term.

$$\lambda(t) = \begin{cases} \cup_{o \in \bar{o}}\, o{\cdot}\lambda(t/o) & \text{if } t \text{ is constructor-rooted and } \bar{o} \text{ is the set of occurrences of outermost operation-rooted subterms of } t; \\ \varphi(t, \mathcal{T}_f) & \text{if } t \text{ is } f\text{-rooted, for some operation } f, \text{ and } \mathcal{T}_f \text{ is a parallel definional tree of } f. \end{cases}$$

$$\varphi(t, \mathcal{T}) = \begin{cases} \phi & \text{if } \mathcal{T} = \mathit{exempt}; \\ \{\Lambda\} & \text{if } \mathcal{T} = \mathit{rule}(l \to r); \\ \cup_{j=1}^{k} \begin{cases} \varphi(t, \mathcal{T}_{ji}) & \text{if } \mathcal{T} = \mathit{branch}(\langle o_1, \ldots, o_k \rangle, \langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle), \text{ for some } k > 0, \\ & \quad t/o \text{ has sort } s \text{ and root the } i\text{-th constructor of } s, \text{ and} \\ & \quad \mathcal{T}_j = \mathcal{T}_{j1}, \ldots, \mathcal{T}_{jk_j}, \text{ for } j \text{ in } 1, \ldots k; \\ o_j{\cdot}\lambda(t/o_j) & \text{if } t/o_j \text{ is operation-rooted.} \end{cases} \end{cases} \tag{7}$$

We now discuss the extension to the parallel case of the results presented in the previous sections for the sequential case. We present our ideas with less detail than before and work out only the key steps of this extension, namely the evolution of the concepts of λ-sequence and needed redex.

From the procedural interpretation viewpoint, λ on input some ground operation-rooted term t generates a tree rather than a sequence of recursive calls. We call this structure the *λ-tree* generated by t and either λ-node or φ-node any node of this tree. If T is a λ-tree, then any path from the root of T down a leaf is just a λ-sequence. The notion of segment is replaced by that of *prefix* of T. A prefix consists of a λ-node n and the descendants of n down all the paths beginning at n and ending either before another λ-node or at a leaf if there is no λ-node down a path.

Lemma 9 is extended to any ground term: *If $\mathcal{R}$ is a weakly orthogonal rewrite system and t is a ground term, then λ is defined on t.* The proof is conceptually identical. To reformulate Lemma 10 we need the concept of i-th *frontier* of a prefix. The i-th *frontier* of a prefix P, for some $i > 0$, is the set of φ-nodes of P at depth i plus all the leaves of P at depth not greater than i. In other words, the set of the end nodes of the prefixes of length up to i of all the segments beginning at the root of P. Then, we can prove, by induction on i, that: *If $\mathcal{R}$ is a weakly orthogonal rewrite system and P is a prefix of the λ-tree generated by some ground operation-rooted term t, then (1) the λ-node t' of P is an operation-rooted term, and (2) for all $i > 0$, if r is a rule of $\mathcal{R}$, then r cannot rewrite any descendant of t' at the root, unless r is contained in a* ppdt *of the i-th frontier of P.*

The concept of needed redex is meaningless in weakly orthogonal rewrite systems. Sekar and Ramakrishnan [22] replace it with that of *necessary set* of redexes. A set S of redexes in a term t is a *necessary set* if in every reduction sequence that takes t to its normal form, at least one redex in S or its descendant is rewritten. Necessary sets take the place of needed redexes in the sense that the parallel rewrite strategy that at every step rewrites all the redexes of a necessary set is normalizing [22]. The definition of necessary set is subtle, since a necessary

set remains necessary even if we add redexes to it. Thus, it is further proved in [22] that when the necessary set considered by the strategy is minimal, the strategy is optimal among those that do not examine the right sides of rules. Thus, the extended Theorem 11 is: *If $\mathcal{R}$ is a weakly orthogonal rewrite system and t is a ground operation-rooted term, then no descendant t' of t can be rewritten to head normal form unless for some occurrence o in $\lambda(t)$, the subterm of t' at o has been rewritten to head normal form.* The proof is similar to that of Theorem 11 with the only difference that now we have sets of redexes rather than a single redex. From this result we derive and easily prove the extension of Corollary 13, that is: *If $\mathcal{R}$ is a weakly orthogonal rewrite system and $t \xrightarrow{*} v$ in $\mathcal{R}$, for some ground term t and value v, then $\lambda(t)$ is a necessary set of occurrence of t.*

A consequence of the last result is that the extension of the outermost-needed strategy to the parallel case yields exactly the strategy based on necessary sets given in [22]. The outermost-needed strategy is the sequential case of this parallel strategy, namely that in which the necessary set is always a singleton. The implementation of the strategy, by means of an automaton discussed in [22], can thus be used for the outermost-needed strategy too.

7 Related Work and Applications

The notion of *sequentiality* has been widely investigated. Seminal work on normalizing strategies appears in [8,20]. The problem is further analyzed and/or summarized in [4,13,21]. The difficulty of the normalization problem led to a number of variations of the concept of sequentiality [3,7,13,23]. The comparison of *inductive sequentiality* with these approaches is outside the scope of this note. Rather, we focus on a few specific issues. We first discuss sequential applications.

For example, the system $Log(F)$ [17–19] is concerned with a class DF^* of rewrite systems and a lazy rewrite strategy for these systems. DF^* bans operations with non-pure columns [1] to guarantee termination. Briefly, for any i, the i-th argument in the left sides of all the rewrite rules defining an operation is either always or never a variable. Further, non-variable arguments in the left side of a rewrite rule are limited to a constructor symbol applied to a tuple of variables. These conditions make programs more difficult to write and to understand, textually longer, and less efficient.

The programming language *BABEL* [14–16], which combines functions, relations, and laziness, frees the programmer from the above restrictions, but enforces them on the implementation. Source programs are transformed into semantically equivalent *uniform*, i.e., *non-subunifiable* and *flat*, programs before execution [14]. Programs in this class are exactly the same as the programs in DF^* and their executions incur a similar loss of efficiency.

The rewrite systems underlying these programs are non-overlapping and we can easily show that they are a subset of the inductively sequential systems. This subset is proper, since, for example, neither operation of display (1) is uniform. Our results allow us both to extend the class DF^* and to avoid intermediate uniform programs in the implementation of *BABEL* without jeopardizing termi-

nation. The arguments of non-pure columns need to be rewritten if and only if they are selected according to λ.

We now discuss how definitional trees can be applied to parallel computations. We have seen, in the *parallel-or* example, that restricting the rules of a rewrite system changes the semantics of computations. Thus, rather than restricting the form of the rewrite rules, an opposite approach consists of relaxing the laziness of the rewrite strategy. For example, the *parallel-outermost* strategy, which rewrites in parallel all outermost redexes of a term, is normalizing [20], although it is likely to perform unnecessary rewrites. Sekar and Ramakrishnan [22] refine this strategy by optimally minimizing unnecessary rewrites. Our research shows an alternative approach to the computation of necessary sets and sheds some light on their internal structure.

Necessary sets are computed in [22] by the algorithm *FindNS*. In this algorithm a set of occurrences denoted $\mathcal{D}'$ is *arbitrarily* chosen as long as the *match set* of the *maximal constructor prefix* of *FindNS*'s argument is covered. Since "for efficiency, it is crucial to select a small $\mathcal{D}'$" [22] the non-determinism is largely removed in an implementation of the rewrite strategy by chosing some $\mathcal{D}'$ with minimum size. A consequence of this minimization is the loss of information useful in controlling some aspects of a parallel rewrite strategy, e.g., synchronization and/or allocation of processors. We will show this point shortly in a set of examples.

Our approach contains a similar aspect of non-determinism. An operation may have several "significantly" different definitional trees, even sequential ones, and our results do not depend on any particular tree. In our approach the minimization of a necessary set is equivalent to minimizing the number of sequential components of a definitional tree. The *equality* operation between natural numbers, defined below, is an example of operation with "non-isomorphic" definitional trees.

$$\begin{aligned} 0 = 0 &\rightarrow \mathit{true} \\ 0 = s(\_) &\rightarrow \mathit{false} \\ s(\_) = 0 &\rightarrow \mathit{false} \\ s(X) = s(Y) &\rightarrow X = Y \end{aligned} \tag{8}$$

There exist two distinct sequential trees of "=", both have a *branch* node at the root. The occurrence arguments in the root nodes of these trees are respectively 1 and 2. This implies that both arguments of the operation must be head normal forms to apply a rewrite rule at the root. This is rather obvious from the defining rules, but if a similar condition is "pushed deeper", i.e., if it occurs only in a proper subset of the rules, then the condition becomes more difficult to detect without suitable conceptual tools. We discuss in an example how we can take advantage of the characteristics of "=" pointed out by its definitional trees.

Consider the operations "$\leq$" of display (1), *parallel-or*, and "=" defined above. These operations are quite different as far as the parallel evaluation of their arguments is concerned. The operation "$\leq$" has a unique sequential definitional tree. Unless we are willing to perform unnecessary rewrites, we must rewrite the first argument of a "$\leq$"-rooted term to head normal form before

deciding on the second argument. The operation *parallel-or* has a parallel definitional tree, but not a sequential one. We want to rewrite *both* arguments of a *parallel-or*-rooted term until *one* of them is a head normal form, and then we can decide whether to keep rewriting the other argument. A minimal necessary set may contain two elements, and these elements are in an *or* relation. The operation "=" has two distinct sequential definitional trees. We can rewrite one argument after the other or *both* arguments in parallel until *both* become head normal forms. A minimal necessary set always contains at most one element. However, there exist non-minimal necessary sets which are still optimal in the sense of [22]. The elements of these sets are in an *and* relation.

Some implications of the structure of a necessary set follow. Although both operations "$\leq$" and "=" are inherently sequential, if both occur in a computation and we have some extra computational resources to allocate, "=" must be preferred to "$\leq$". Although the arguments of both operations *parallel-or* and "=" can be rewritten in parallel without wasted rewrites, the mutual relationships between the elements of the set of parallel computations of each operation reflect the structure of the corresponding necessary set. In the first case the computation of one argument of *parallel-or* must be able to stop the computation of the other argument, whereas in the second case the computation of one argument of "=" is independent of that of the other argument, since the outcome of rewriting one argument has no influence on the need of rewriting the other argument.

8 Concluding Remarks

Rewriting is a computational paradigm that specifies the actions, but not the control. Definitional trees are a high-level abstraction to specify or infer some aspects of the control. The problem of building definitional trees from rewrite systems has been addressed only marginally in Theorem 19. Some algorithms for this task are presented in [24].

We have shown the application definitional trees to two interesting classes of rewrite systems, one appropriate for sequential computations, the other for parallel ones. An application of our results to inherently sequential computations allows us to extend the expressive power of some programming languages based on rewriting and/or the efficiency of their implementations. An application of our results to parallel computations allows us to detect opportunities for parallel rewriting in sequential computations, without wasted rewrites and with a simpler control.

Definitional trees are not limited to weakly orthogonal rewrite systems. Structures very similar to definitional trees, but storing *sets* of rules in *both* leaves and branches, have been applied with interesting results [2] to a class of rewrite systems larger than those discussed in this note.

References

1. Sergio Antoy. Design strategies for rewrite rules. In S. Kaplan aand M. Okada, editor, *CTRS'90*, pages 333–341, Montreal, Canada, June 1990. Lect. Notes in

Comp. Sci., Vol. 516.
2. Sergio Antoy. Non-determinism and lazy evaluation in logic programming. In T. P. Clement and K.-K. Lau, editors, *LOPSTR'91*, pages 318–331, Manchester, UK, July 1991. Springer-Verlag.
3. G. Boudol. Computational semantics of term rewriting systems. In Maurice Nivat and John C. Reynolds, editors, *Algebraic methods in semantics*, chapter 5. Cambridge University Press, Cambridge, UK, 1985.
4. L. G. Bouma and H. R. Walters. Implementing algebraic specifications. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, chapter 5. Addison-Wesley, Wokingham, England, 1989.
5. N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.
6. J. Goguen, J. Meseguer, S. Leinwand, T. Winkler, and H. Aida. The rewrite rule machine project. Technical Report SRI-CSL-89-6, SRI International, Menlo Park, CA, 1989.
7. C. M. Hoffmann and M. J. O'Donnell. Implementation of an interpreter for abstract equations. In *11th ACM Symposium on the Principle of Programming Languages*, Salt Lake City, 1984.
8. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991. Previous version: Call by need computations in non-ambiguous linear term rewriting systems, Technical Report 359, INRIA, Le Chesnay, France, 1979.
9. Gérard Huet. Confluent reductions: Abstract properties and applications to term-rewriting systems. *JACM*, 27:797–821, 1980.
10. J. R. Kennaway. Sequential evaluation strategies for parallel-or and related reduction systems. *Annals of Pure and Applied Logic*, 43:31–56, 1989.
11. Claude Kirchner and Patrick Viry. Implementing parallel rewriting. In *PLILP'90*, pages 1–15, Linköping, Sweden, August 1990. Lect. Notes in Comp. Sci., Vol. 456.
12. J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992. Previous version: Term rewriting systems, Technical Report CS-R9073, Stichting Mathematisch Centrum, Amsterdam, 1990.
13. Jan Willem Klop and Aart Middeldorp. Sequentiality in orthogonal term rewriting systems. Technical Report CS-R8932, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1989.
14. H. Kuchen, R. Loogen, J. J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-based implementation of a functional language. In *ESOP'90*, pages 279–290, 1990. Lect. Notes in Comp. Sci., 432.
15. J. J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Conf. on Algebraic and Logic Progr.*, 1990. Lect. Notes in Comp. Sci., 463.
16. J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.
17. Sanjai Narain. *LOG(F): An optimal combination of logic programming, rewriting, and lazy evaluation*. PhD thesis, University of California, Los Angeles, CA, 1988.
18. Sanjai Narain. Optimization by non-deterministic, lazy rewriting. In *RTA'89*, pages 326–342, Chapel Hill, NC, 1989. Lect. Notes in Comp. Sci., Vol. 355.

19. Sanjai Narain. Lazy evaluation in logic programming. In *Third IEEE Conference on Computer Languages*, pages 218–227, New Orleans, 1990.
20. Michael J. O'Donnell. *Computing in Systems Described by Equations.* Springer-Verlag, 1977. Lect. Notes in Comp. Sci., Vol. 58.
21. Michael J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, 1985.
22. R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 230–241, Philadelphia, PA, June 1990.
23. Satish Thatte. A refinement of strong sequentiality for term rewriting with constructors. *Information and Computation*, 72:46–65, 1987.
24. Yonggong Yan. Building definitional trees. Master's project, Portland State University, Portland, OR, March 1992.