TERMINATION OF NON-SIMPLE REWRITE SYSTEMS

BY

CHARLES GLEN HOOT

B.A., University of California, San Diego, 1982
M.A., Princeton University, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

TERMINATION OF NON-SIMPLE REWRITE SYSTEMS

Charles Glen Hoot, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1996
Nachum Dershowitz, Advisor

Rewriting is a computational process in which one term is derived from another by replacing a subterm with another subterm in accordance with a set of rules. If such a set of rules (*rewrite system*) has the property that no derivation can continue indefinitely, it is said to be terminating. Showing termination is an important component of theorem proving and of great interest in programming languages.

Two methods of showing termination for rewrite systems that are self-embedding are presented. These "non-simple" rewrite systems can not be shown terminating by any of what are called simplification orderings. The first method of termination employs lexicographic combinations of quasi-orderings including the ordering itself applied to multisets of immediate subterms in a general path ordering. Two versions are presented. The well-founded and well-quasi general path orderings respectively require their component orderings to be well-founded and well-quasi orderings. The definitions are shown to result in well-founded and well-quasi orderings, respectively. A general condition is presented for showing termination of a rewrite system with a quasi-ordering. Conditions on the component orderings are presented which guarantee that the general conditions are satisfied. The well-quasi general path ordering is applied to several examples to show termination.

The second method of showing termination is to use sets of derivations called the "forward closures" of a rewrite system. New results are derived that give syntactic conditions under which termination of the forward closures guarantees termination of the rewrite system. A

theorem is presented that shows the relationship of forward closures with innermost rewriting. If there is a class of rewrite systems for which innermost rewriting implies termination, then termination of forward closures will imply termination as well. Restricting the set of forward closures to derivations which satisfy some strategy such as choosing an innermost redex is explored. Syntactic conditions are given for which termination of innermost or outermost forward closures implies termination in general. The method of forward closures is then used to show the termination of some example rewrite systems including the string rewriting system $0011 \rightarrow 111000$.

A test for non-termination of a rewrite system using forward closures (FCT) is presented. A previous method (MSP) using semi-unification is analyzed and it is shown that certain kinds of rewrite rules may be ignored without affecting the ability of MSP to detect non-termination. Using this result one can also show that FCT will detect non-termination in every case that MSP will, but not vice-versa. Results are also presented showing that information can be obtained from forward closures about the termination of innermost derivations from terms of limited size with all subterms in normal form. A method for computing innermost and outermost forward closures is presented which avoids extra checking of earlier parts of the derivations to guarantee the redexes remain innermost/outermost. Also given is a completion like method for generating an innermost locally confluent rewrite system which preserves innermost derivations of a given rewrite system.

Finally, there are appendices describing the interface to code written in common lisp which implements the well-quasi general path ordering and showing its usage to prove termination of a rewrite system for insertion sort.

ACKNOWLEDGEMENTS

I would like to thank Dr. Stephen Tang at the Aerospace Corporation for being a mentor to me and encouraging me to continue my studies at the graduate level. In addition, I would also like to thank my advisor, Professor Dershowitz, for his support, inspiration and occasional prodding. Without his guidance, this thesis would not have been completed. Finally, my wife Susan and daughters Liliana and Magdalena have been extremely patient and caring throughout the many days when I was not able to spend as much time as I would have liked with them.

TABLE OF CONTENTS

1 INTRODUCTION

Rewrite systems are sets of directed equations used to compute by repeatedly replacing terms in a given formula with equal terms, as long as possible. The theory of rewriting is an outgrowth of the study of the lambda calculus and combinatory logic, and has important applications in abstract data type specifications, functional programming, symbolic computation, and automated deduction. For surveys of the theory of rewriting, see Dershowitz and Jouannaud [DJ90], Klop [Klo92] and Plaisted [Pla93b].

If no infinite sequences of rewrites are possible, a rewrite system is said to have the *termination* property. In practice, one usually guarantees termination by devising a well-founded (strict partial) ordering $\succ$ such that $s \succ t$ whenever s rewrites to t (written, $s \to t$). As suggested by Manna and Ness [MN70], it is often convenient to express reduction orderings as a homomorphism from terms to an algebra equipped with a well-founded ordering. The use, in particular, of *polynomial interpretations* which map terms into the natural numbers was developed by Lankford [Lan79]. For a survey of termination methods, see Dershowitz [Der87].

The rule

$$x \times (y + z) \quad \to \quad (x \times y) + (x \times z) \tag{1.1}$$

is terminating. This can be shown by interpreting $\times$ as multiplication, $+$ as $\lambda xy.x + y + 1$, and constants as 2. Since $x \geq 2$ implies $x(y + z + 1) > xy + xz + 1$, the rule is terminating. It can also be proved terminating by considering the multiset of "natural" interpretations of all products in a term, letting $+$ and $\times$ stand for addition and multiplication, and assigning some fixed value to constants; see Dershowitz and Manna [DM79] for similar examples. Syntactic "path" orderings (see Dershowitz [Der87]) work in this case, too. Lipton and Snyder [LS77] gave a particular method for proving termination with interpretations (order-isomorphic to ω) for which rules are "value-preserving", as this example is for the natural interpretation.

Virtually all orderings used in practice are *simplification orderings* [Der82], satisfying the *replacement* property, that $s \succ t$ implies that any term containing s as a subterm is at least as large (under $\succ$) as the same term with s replaced by t, and the *subterm* property, that any term containing s is at least as large as s. Simplification orderings are surveyed by Steinbach

[Ste89]; their well-foundedness is a consequence of Kruskal's Tree Theorem. (See Dershowitz [Der82].) A *non-simple* rewrite system (such as $ffx \to fgfx$) is one for which no simplification ordering will show termination.

Knuth and Bendix [KB70] designed a particular class of well-orderings which assigns a weight to a term that is the sum of the weights of its constituent function symbols. Terms of equal weight and headed by the same symbol have their subterms compared lexicographically. If they are headed by different symbols, a "precedence" ordering determines which term is larger. Another class of simplification orderings, the *path orderings* were introduced around 1980. Plaisted in [Par78] defined the *simple path ordering* which mapped a term t to multiset of paths in the term. The *recursive path ordering* introduced in Dershowitz [Der82], is based on the idea that a term u should be bigger than any term that is built from smaller terms, all held together by a structure of function symbols that are smaller in some precedence ordering than the root symbol of u. The notion of path ordering was extended by Kamin and Lévy [KL80] to compare subterms lexicographically and to allow for a semantic component; see Dershowitz [Der87].

In the thesis quasi-orderings (reflexive-transitive binary relations), rather than partial orderings, are used to prove termination of rewrite systems. If $\succsim$ is a quasi-ordering and $\precsim$ is its inverse, then its strict part $\succ$ $(\succsim - \precsim)$ is a partial order. Its associated equivalence relation $\approx$ is defined as $\succsim \cap \precsim$. A quasi-ordering is *well-founded* if it has no infinite strictly descending sequence of elements. A quasi-ordering is *well-quasi* if in addition to being well-founded it has no infinite set of incomparable elements.

A *precedence* is a well-founded quasi-ordering of function symbols. An ordering can be called *syntactic* if it is based on a precedence and is invariant under shifts of symbols. In other words, one requires that consistently replacing function symbols in two terms with others of the same arity and with the same relative ordering has no effect on the ordering of the two. The recursive path orderings [Der82; KL80; Les90] are syntactic; the Knuth-Bendix and polynomial orderings are not.

Simplification orderings cannot be used to prove termination of "self-embedding" systems, that is, when a term t can be derived in one or more steps from a term t', and t' can be obtained by repeatedly replacing subterms of t with subterms of those subterms. For example, consider the following contrived system for computing factorial in unary arithmetic (expanding on one

in Kamin and Lévy [KL80]):

$$\begin{array}{rcl} p(s(x)) & \rightarrow & x \\ fact(0) & \rightarrow & s(0) \\ fact(s(x)) & \rightarrow & s(x) \times fact(p(s(x))) \\ 0 \times y & \rightarrow & 0 \\ s(x) \times y & \rightarrow & (x \times y) + y \\ x + 0 & \rightarrow & x \\ x + s(y) & \rightarrow & s(x + y)\ . \end{array} \tag{1.2}$$

It would be nice to be able to use a natural interpretation, but that does not prove termination, since the rules preserve the value of the interpretation, rather than cause a decrease. Nor can multisets of the values of the argument of *fact* be used, since some rules can multiply occurrences of that symbol. Though path orderings have been successfully applied to many termination proofs, they suffer from the same limitation as do all simplification orderings: they are not useful when a rule embeds as does $fact(s(x)) \rightarrow s(x) \times fact(p(s(x)))$.

What is needed is a way of combining the semantics given by a natural interpretation with a non-simplification ordering that takes the structure of terms into account. Two closely related orderings are presented and will be called *general path orderings*. In Chapter 3, the *well-quasi general path ordering* (WQGPO) is presented and is proven to be a a well-quasi ordering. In Chapter 4 general conditions are given for showing termination of a well-founded quasi-ordering. Specific conditions for composing component orderings are then given under which the WQGPO can be use to show termination of rewrite systems (both simple and non-simple). In Chapter 5 it is shown that the WQGPO generalizes many of the above-mentioned orderings and a new ordering called the *natural path ordering* is presented. In Chapter 6 extensive examples of the use of the well-quasi general path ordering are given. Included is an example showing termination of an insertion sort over natural numbers. The ordering used is unlike any of the previously mentioned standard techniques for showing termination. Also included are some conditional rewrite systems which make use of value-preserving orderings.

In Chapter 7 the second version of the general path ordering called the *well-founded general path ordering* (WFGPO) is presented. It is less restrictive in that it allows orderings to be combined which are well-founded, but not necessarily well-quasi. An additional restriction

requiring orderings which examine all subterms is needed, however. The well-founded general path ordering is shown to be a well-founded quasi ordering and the conditions under which it can be applied are presented. Finally, comparisons are made between the two general path orderings.

One under-used approach to termination is the use of restricted derivations [Der81; GKM83; Geu89]. The *forward closures* of a given rewrite system are an inductively defined set of derivations. The basic idea is to only consider derivations in which application of rules is in that part of a term created by previous rewrites.

In general, termination of forward closures does not ensure termination of a rewrite system. In this thesis results are presented which weaken the condition under which termination forward closures are sufficient to show the termination of a rewrite system. One important result is a theorem which shows the relation between forward closures and innermost termination of a rewrite system.

Also investigated are restrictions to the set of forward closures based on rewrite strategies. Popular strategies include restricting the position of a rewrite application to an innermost or outermost redex. Syntactic conditions on rewrite systems are presented in Chapter 8 which allow one to show termination via termination of a restricted set of forward closures.

In Chapter 9 an application of forward closures is investigated. Completion is the process by which a set of equations is converted to an equivalent rewrite system which is confluent and terminating. Typically the process of completion involves searching for a rewrite system which allows the completion to orient all of the generated rules. This process can be using heuristics to orient the rules independently of an ordering. In this thesis, a previous method of detecting a non-terminating set of rules due to Purdom [Pur87] is analyzed. A new method using forward closures is proposed which is shown to be strictly more powerful.

Termination of innermost forward closures is sufficient to guarantee innermost termination of a rewrite system. This can be exploited in a number of cases. For example, many programming languages are applicative, and hence innermost derivations may be all that one is interested in when proving termination. In addition, restricting completion to innermost derivations has the benefit of severely limiting the number of possible critical pairs to be considered. Only overlaps at the top position need to be considered [Pla93b]. In Chapter 10, using innermost forward closures with completion are explored. It is shown that while the set of forward closures may

be infinite, innermost termination for terms of restricted size may be determined by examining a finite number of forward closures. In addition, methods for more efficiently computing the the innermost and outermost forward closures of a rewrite system are presented.

2 BACKGROUND

This chapter presents the terminology and notation that will be used throughout the rest of the thesis. Useful results are presented for well-quasi orderings.

2.1 Terminology

The following notation, definitions, and propositions are reasonably standard and usually conform with those presented by Dershowitz and Jouannaud in [DJ90].

2.1.1 Terms

Terms are constructed recursively from a set of function symbols $\mathcal{F}$ and a set of variables $\mathcal{X}$. Each function symbol $f \in \mathcal{F}$ has an *arity* which is the number of subterms that the function f has. *Constants* are those function symbols with an arity of zero. A *unary* function symbol has an arity of one. A *binary* function symbol has an arity of two. The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ contains all the constants and variables. Any term $t = f(t_1, \ldots, t_n)$ is also a member, provided that f has arity n and each of the terms $t_1, \ldots, t_n$ is also in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Terms which are variable free are denoted as *ground* terms. A term t with a *subterm* s will be denoted as $t = C[s]$ where C is the *context* of the subterm. The subterm $s = t|_p$, where p is the *position* of the subterm. Since a term can be represented as a finite tree with all the internal nodes labeled with non-constant function symbols, positions can be denoted as a path from the root of the tree. The position $p = \Lambda$ is the root of the tree. The subterm associated with the position $p = p'.i$ is the ith subterm of the subterm associated with p'. As convenience, the *immediate subterms* of a term t are denoted $t_1, \ldots, t_n$ and $t = f(t_1, \ldots, t_n)$. To avoid confusion subscripting will be reserved for indicating subterms of the terms t, s, u, and v. Proofs that use a sequence or set of terms will use superscripting, e.g., $t^0, t^1, t^2, \ldots$ is a sequence of terms.

The set of variables in a term is denoted by $Var(t)$. The letters a through h will be reserved for function symbols, x, y and z are reserved for variables, and l,r,s, and t are reserved for terms. Most terms will be written in a functional notation. For example, $t = f(g(x, y), a, z)$ is a term composed of the variables x, y, and z, the constant a, the function symbol g (with arity two),

and the function symbol f (with arity three). The exceptions are that for function symbols of arity one the parentheses will usually be dropped and for common binary mathematical functions infix notation will be used. For example, the term $h(h(h(a)))$ would be written as $hhha$, and the term $-(x, +(y, z))$ would be written as $x - (y + z)$.

2.1.2 Substitutions and Unification

A *substitution* is a set of mappings from variables to terms denoted as $\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$. Any variable term x_i is replaced by the corresponding term s_i in the mapping. A substitution is extended recursively to a term $t = f(t_1, \ldots, t_n)$ by applying $t\sigma = f(t_1\sigma, \ldots, t_n\sigma)$. The *composition* of two substitutions σ and μ, is denoted $\sigma \circ \mu$ and represents the composition of the two mappings. Formally, $t\sigma \circ \mu = (t\sigma)\mu$. For convenience, this will usually be written as just $t\sigma\mu$. A *renaming substitution* is one in which the terms $s_1, \ldots, s_n$ are all pairwise different variables. For example, the substitutions $\sigma_1 = \{x \mapsto y, y \mapsto x\}$, $\sigma_2 = \{x \mapsto y, y \mapsto z\}$, and $\sigma_3 = \{x \mapsto w, y \mapsto z\}$ are all renaming substitutions.

Given sets of terms $\bar{t} = \{t^1, \ldots, t^n\}$ and $\bar{s} = \{s^1, \ldots, s^n\}$, the substitution σ is said to be a *unifier* if the following equations are satisfied

$$\begin{aligned} t^1\sigma &= s^1\sigma \\ &\vdots \\ t^n\sigma &= s^n\sigma \ . \end{aligned} \tag{2.1}$$

Typically, one is concerned with the special case of just two terms, in which case t is said to unify with s provided that $t\sigma = s\sigma$. A substitution σ is a *most general unifier* if for any unifier μ, there is some substitution τ such that $\mu = \sigma \circ \tau$. Note that the most general unifier is not unique. It is only unique up to a renaming substitution.

Sets of terms $\bar{t} = \{t^1, \ldots, t^n\}$ and $\bar{s} = \{s^1, \ldots, s^n\}$ *match* if there is a substitution σ where the following equations are satisfied

$$\begin{aligned} t^1\sigma &= s^1 \\ &\vdots \\ t^n\sigma &= s^n \ . \end{aligned} \tag{2.2}$$

Typically, one is concerned with the special case of just two terms, in which case t is said to match s provided that $t\sigma = s$ for some substitution σ. For example, the term $t = fx$ matches the term $s = fga$ with the substitution $\sigma = \{x \mapsto ga\}$.

Sets of terms $\bar{t} = \{t^1, \ldots, t^n\}$ and $\bar{s} = \{s^1, \ldots, s^n\}$ *semi-unify* if there are substitutions σ and μ where the following equations are satisfied

$$\begin{aligned} t^1\sigma\mu &= s^1 \\ &\vdots \\ t^n\sigma\mu &= s^n \ . \end{aligned} \tag{2.3}$$

2.1.3 Rewrite Systems

An *equation* is an unordered pair of terms (s, t) denoted as $s = t$. If s and t contain variables it is understood that they are universally quantified. In other words, for every substitution σ the equation $s\sigma = t\sigma$ is true. An *equational theory* is induced by a set of equations E and is denoted by $=_E$. Two terms u and v are equal under an equational theory $u =_E v$ if

1. $u = s\sigma$ and $v = t\sigma$ for some substitution σ and equation $s = t \in E$,
2. $u = C[s]$ and $v = C[t]$ and $s =_E t$ where C is a non-empty context, or
3. there is some term w such that $u =_E w$ and $w =_E v$.

This is the standard algebraic notion that equals are replaced by equals. Suppose that one is given the set of equations

$$E = \left\{ \begin{aligned} x + 0 &= x \\ x + s(0) &= x \\ x + s(y) &= s(x + y) \end{aligned} \right\} \tag{2.4}$$

Since $0 =_E 0 + s(0) =_E s(0 + 0) = s(0)$ and $s(0) =_E s(0) + s(0) =_E s(s(0) + 0) =_E s(s(0))$ it is also the case that $f(0) =_E f(s(s(0)))$. Notice that in this case all of the terms $s^i(0)$ are equal under E.

A *rewrite rule* $l \to r$ is a directed equation such that the all variables on the right-hand side are also on the left-hand side, i.e., $Var(r) \subseteq Var(l)$. A rewrite system $\mathcal{R}$ is a set of rewrite rules. A *rewrite step* or *derivation step* is obtained by application of a rule $l \to r$ to a term t. The rule can be applied if there is some subterm s of t such that l matches s (there is a

substitution σ where $s = l\sigma$.) In this case the rule matches the *redex* s. The *derivation step* is $t = C[s] \rightarrow C[r\sigma]$. The rewrite relation on a set of terms associated with a rewrite system $\mathcal{R}$ is denoted by $\rightarrow_{\mathcal{R}}$. If there is only one rewrite system, or it is clear from the context which rewrite system is being referred to this will be shortened to just $\rightarrow$. The *derivability* relation $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$. The notation $t \rightarrow^* t'$ indicates that t rewrites to t' in zero or more steps. The notation $t \rightarrow^+ t'$ indicates that t rewrites to t' in one or more steps. A rewrite step $t = C[s] = C[l\sigma] \rightarrow C[r\sigma]$ is *innermost* with respect to $\mathcal{R}$ if there is no proper subterm $u = s|_p$ of s such that $u \rightarrow u'$ for some u'. In other words, every proper subterm of s is in *normal form* (can not be rewritten). A rewrite step $t = C[s] = C[l_i\sigma] \rightarrow C[r_i\sigma]$ is *outermost* with respect to $\mathcal{R}$ if there is no subterm $u = t|_p$ of t where s is a proper subterm of u and $u \rightarrow u'$ for some u'. In other words, there is no redex above s.

A *left-linear* rewrite system has no repeated variables on the left-hand side of a rule. Similarly, a *right-linear* system has no repeated variables on the right-hand side of a rule. A pair of rules $l_i \rightarrow r_i$ and $l_j \rightarrow r_j$ *overlap* if there is some non-variable subterm of l_i which unifies with l_j. Essentially, this represents a situation where there is a term t in which both of the rules can be applied and the redexes share context. A *non-overlapping* rewrite system is one where no left-hand side of a rule unifies with any non-variable subterm of the left-hand side of another rule or with a non-variable proper subterm of itself when variables in the two rules are renamed apart. An *overlaying* rewrite system is one whose only overlaps are at the topmost position, that is, no left-hand side unifies with a non-variable, proper subterm of any left-hand side. A rewrite system is *non-erasing* if any variable on the left-hand side of a rule is also on the right-hand side. An *orthogonal* rewrite system is non-overlapping and left-linear. A *ground* rewrite system is one which has no variables. A *locally confluent* rewrite system is one for which $u \rightarrow s, t$ implies $s, t \rightarrow^* v$, for some v. A *confluent* rewrite system is one for which $u \rightarrow^* s, t$ implies $s, t \rightarrow^* v$, for some v. Note that local confluence does not imply confluence in general. A rewrite system is said to have the *unique normal form* property if $u \rightarrow^* n_1, n_2$ where n_1 and n_2 are normal forms, implies $n_1 = n_2$. Certain symbols may be denoted as *constructors* and must not be the topmost symbol on the left-hand side of any rule. A term is *constructor-based* if all of its proper subterms have only free constructors and variables. A rewrite system is constructor-based if its left-hand sides are constructor-based, and a forward closure is constructor-based if its initial term is constructor-based.

2.2 Termination

A *term* t is *terminating* (and is denoted by $t \in \mathcal{T}_f$) if all derivations from t are finite; t is *non-terminating* (and is denoted by $t \in \mathcal{T}_\infty$) if some derivation from t is infinite; and t is on the *frontier* (and is denoted by $t \in \mathcal{FR}$) if t is non-terminating, but every proper subterm of t is terminating. If a term has no frontier subterms, then it must be terminating. Conversely, if a term has a frontier subterm, it is non-terminating. If no infinite sequences of rewrites are possible, a rewrite system is said to have the *termination* property.

An *well-founded ordering* is a partial ordering which has no infinite descending sequences. Such an ordering can be used to prove the termination of a rewrite relation, by showing that the rewrite relation embeds within the well-founded ordering (and thus that every rewrite step shows a decrease within the ordering).

Virtually all orderings used in practice are *simplification orderings* [Der82], satisfying

- the *replacement* property, that $s \succ t$ implies that any term containing s as a subterm is at least as large (under $\succ$) as the same term with s replaced by t, and
- the *subterm* property, that any term containing s is at least as large as s.

Simplification orderings are surveyed by Steinbach [Ste89]; their well-foundedness is a consequence of Kruskal's Tree Theorem [Kru60].

Definition 1. A term t is *homeomorphically embedded* in a term s written $t \trianglelefteq s$ if

1. t and s are identical,
2. t is homeomorphically embedded in s_i an immediate subterm of s $(t \trianglelefteq s_i)$, or
3. t and s have the same function symbol on top and the immediate subterms of t homeomorphically embed in the corresponding immediate subterms of s $(t = f(t_1, \ldots, t_n)$ and $s = f(s_1, \ldots, s_n)$ and for each pair of subterms $t_i \trianglelefteq s_i$.)

This definition needs to be extended slightly if one wants to handle the possibility of variadic function symbols, but in this thesis that possibility will be excluded. A rewrite derivation $t^1 \rightarrow t^2 \rightarrow \ldots \rightarrow t^i \ldots$ is said to be *self-embedding* if there are terms t^i and t^j where $t^i \trianglelefteq t^j$ and $i < j$.

Proposition 1 (Kruskal's Tree Theorem). *If F is finite set of function symbols, then any infinite sequence $t^1, t^2, \cdots$ of terms in the set $\mathcal{T}(\mathcal{F})$ of terms over F contains two terms t^i and t^j such that $t^i \trianglelefteq t^j$ and $i < j$.*

This was shown by Kruskal in [Kru54] and [Kru60]. The special case where each of the function symbols is of fixed arity was shown earlier by Higman in [Hig52].

Proposition 2. *If a finite rewrite system is non-terminating, then it is self-embedding.*

Proof. This is due to Dershowitz in [Der82]. If the rewrite system $\mathcal{R}$ does not terminate then there must be an infinite derivation. Since there are only a finite number of function symbols, by Proposition 1 there must two terms t^i and t^j such that $t^i \trianglelefteq t^j$ and $i < j$. Therefore the derivation self-embeds. □

This gives one a necessary condition for termination, but it is not sufficient since there are terminating rewrite systems which are self-embedding. For example, $ffx \rightarrow fgfx$ is self-embedding and terminating.

Proposition 3. *Given a rewrite system $\mathcal{R}$ and two terms t and s such that t is homeomorphically embedded in s ($t \trianglelefteq s$) and $t \rightarrow s$, $\mathcal{R}$ can not be shown terminating by any simplification ordering.*

A simplification ordering $\succ$ extends the the relation $\rhd$ [Der82]. Thus, in the simplification ordering $s \succ t$ and the derivation can not be shown to terminating.

2.3 Quasi-Orderings

This section reviews quasi-orderings. It contains several Propositions that will be used later in the thesis.

A relation $\succsim$ on a set S is a *Quasi-ordering* if it is transitive and reflexive. The associated equivalence relation on S is given by $s \equiv t$ if $s \succsim t$ and $t \succsim s$. The associated partial ordering $\succ$ (the strict part of the Quasi-ordering) on S is given by $s \succ t$ if $s \succsim t$ but not $t \succsim s$.

The classic example is the set real numbers with respect to the relation greater than or equal modulo n.

A Quasi-ordering is *well-founded* if the strict part, $\succ$ is well-founded. The previous example is not well-founded since the sequence $\epsilon > \epsilon/2 > \epsilon/4 > \cdots$ is not well-founded. On the other hand, if one restricts the set to the integers, it will be well-founded.

Definition 2. An ordering on S is a *well-quasi-ordering* if for any infinite sequence of of elements $s^1, s^2, s^3, \ldots$, there exist two elements s^i and s^j such that, $s^i \precsim s^j$ and $i < j$.

This definition was first proposed by Kruskal in [Kru60]. Any well-quasi-ordering must also be well-founded. In addition, there can not be an infinite number of incomparable elements. For a history of well-quasi orderings and a survey of their uses, see [Kru72].

Proposition 4. *If $\precsim$ is a well-quasi-ordering over S, then in any infinite sequence of elements $s^1, s^2, s^3, \ldots$, there is an infinite subsequence $s^{i_1}, s^{i_2}, s^{i_3}, \ldots$, such that $s^{i_1} \precsim s^{i_2} \precsim s^{i_3} \precsim \ldots$ and $i_1 < i_2 < i_3 < \ldots$.*

Proof. This is the infinite version of Ramsey's theorem. Assume that this is not the case. Consider an arbitrary infinite sequence of elements. By the definition of a well-quasi-ordering, there must be a pair of elements s^i and s^j such that $s^i \precsim s^j$. From s^j construct a sequence of terms $s^j, s^{j_1}, s^{j_2}, \ldots, s^{j_n}$ where $s^j \precsim s^{j_1} \precsim s^{j_2} \precsim \ldots \precsim s^{j_n}$. This sequence must be finite by assumption and s^{j_n} must be greater than or incomparable to all succeeding elements. Removing the j_n elements from the sequence leaves a new infinite sequence for which a new subsequence of elements can be found. Repeating this an infinite number of times and taking the last element in each subsequence, allows one to construct an infinite sequence of terms, such that each one is greater than or incomparable to all the succeeding terms. But this is a contradiction. □

Proposition 5. *If $\precsim_1, \precsim_2, \ldots, \precsim_n$ are well-quasi-orderings over S then any lexicographic combination is a well-quasi-ordering as well.*

Proof. Consider an infinite sequence of terms $s^1, s^2, s^3, \cdots$. By the previous Proposition, an infinite subsequence of terms can be constructed (call them $t^1, t^2, t^3, \cdots$) such that $t^1 \precsim_1 t^2 \precsim_1 t^3 \precsim \cdots$.

Clearly, this process can be continued with each of the n orderings until a sequence of terms $u_1, u_2, u_3, \cdots$ is obtained which is a subsequence of the original where $u_1 \precsim_1 u_2$, $u_1 \precsim_2 u_2, \cdots, u_1 \precsim_n u_2$. But u_1 is less than or equal to u_2 under the lexicographic

combination of the orderings and u_1 occurs before u_2 in the sequence, therefore the lexicographic combination is also a well-quasi-ordering. □

Definition 3. An *embedding relation* $\trianglerighteq_{\gtrsim}$ defined on the set of finite sequences $\mathcal{S}^*$ over the set S by the quasi-order $\gtrsim$, is given by $(s_1, \ldots, s_m) \trianglerighteq_{\gtrsim} (t_1, \ldots, t_n)$ if $s_{i_j} \gtrsim t_i$ for all $j = 1, \ldots, n$ with $1 \leq i_1 < i_2 < \ldots < i_n \leq m$.

Proposition 6 (Higman's Lemma). *An* embedding relation $\trianglerighteq_{\gtrsim}$ *defined on the set of finite sequences $\mathcal{S}^*$ over the set S is well-quasi ordered if, and only if, the ordering $\gtrsim$ is a well-quasi ordering over S.*

This was shown by Higman in [Hig52] and shows that the embedding relation preserves the well-quasi ordering. The proof proceeds by a minimal counter example.

Proposition 7. *If $\gtrsim$ is a well-quasi-ordering over S, the extension of $\gtrsim$ to multi-sets $\gtrsim_{\mathcal{M}}$ is also a well-quasi-ordering.*

Proof. This is a direct result of Higman's Lemma.

□

3 THE WELL-QUASI GENERAL PATH ORDERING (WQGPO)

In this chapter the definition of the well-quasi general path ordering is given.[1] The well-quasi general path ordering combines mappings from terms to well-quasi-ordered sets. It is shown that the well-quasi general path ordering is, in fact, a well-quasi-ordering and hence is a candidate for showing termination of rewrite systems.

3.1 Definitions

In this section concepts are introduced for use with the either of the general path orderings. Included are some basic kinds of component orderings used by a general path ordering. Finally the definition of the well-quasi general path ordering is given.

Definition 4 (Termination Function). A *termination function* θ takes a term as argument and is of one of the following types:

a. a homomorphism from terms to an algebra (set of values) $\mathcal{A}$, where $\theta(f(s_1, \ldots, s_n)) = f_\theta(\theta(s_1), \ldots, \theta(s_n))$, and f_θ is a function from $\mathcal{A}^n$ to $\mathcal{A}$ for n-ary function symbol f;

b. an extraction function from terms to multisets of selected immediate subterms, that is $\theta(f(s_1, \ldots, s_n)) = \{s_{j_1}, \ldots, s_{j_m}\}$, such that $j_1, \ldots, j_m \in \{1, \ldots, n\}$ where the choice of the subterms depends on the function symbol f.

We say that $s \gtrsim t$ for terms s and t containing variables $\bar{x}$ if $s\sigma \gtrsim t\sigma$ for all ground (variable free) substitutions σ for the variables $\bar{x}$.

Definition 5 (Component Order). Let $\mathcal{T}$ be a set of variable-free terms (over some alphabet). A *component order* $\phi = \langle \theta, \geq \rangle$ consists of a termination function $\theta : \mathcal{T} \to \mathcal{A}$, from terms to an algebra $\mathcal{A}$ along with an associated well-quasi-ordering $\geq$ over $\mathcal{A}$.

[1] The definition of the general path ordering given in [DH95] is a little different. See Chapter 7 for a discussion.

The following definitions are useful ($\simeq$ denotes the equivalence part of $\geq$):

- A homomorphism θ is *value-preserving* with respect to the ordering $\geq$ and rewrite system $\mathcal{R}$ if $\theta(l\sigma) \simeq \theta(r\sigma)$ for all rules $l \to r$ in $\mathcal{R}$ and substitutions σ.
- A homomorphism θ is *monotonic* with respect to the ordering $\geq$ if for all function symbols f, $f_\theta(\ldots x \ldots) \geq f_\theta(\ldots y \ldots)$ whenever $x > y$.
- A homomorphism θ is *strictly monotonic* with respect to the ordering $\geq$ if for all function symbols f, $f_\theta(\ldots x \ldots) > f_\theta(\ldots y \ldots)$ whenever $x > y$.
- A homomorphism θ has the *strict subterm* property with respect to the ordering $\geq$ if for all function symbols f, $f_\theta(\ldots x \ldots) > x$.
- An equivalence relation $\simeq$ is a *congruence* with respect to a homomorphism θ if for all function symbols f, $x \simeq y$ implies $f_\theta(\ldots x \ldots) \simeq f_\theta(\ldots y \ldots)$.
- The multiset $R_i(S)$ of terms of rank i ($i \geq 0$) with respect to the ordering $\geq$ on terms in a multiset of terms S, is inductively defined as

$$R_i(S) = \{u \ : \ u \text{ is maximal with respect to } \geq \text{ in } L_i(S)$$

where

$$L_i(S) = S - \bigcup_{0<j<i} R_j(S).$$

Definition 6. Some important classes of component orders are:

a. $\langle \theta, \geq \rangle$ is a *precedence* when θ is a homomorphism which returns the outermost function symbol of a term and $\geq$ is a precedence ordering;

b. $\langle \theta, \geq \rangle$ is *value-preserving* when θ is a value-preserving homomorphism with respect to $\geq$ and $\geq$ is a well-quasi-ordering;

c. $\langle \theta, \geq \rangle$ is *monotonic* when θ is a monotonic homomorphism with the strict subterm property (with respect to $\geq$) and $\geq$ is a well-quasi-ordering;

d. $\langle \theta, \geq \rangle$ is *strictly monotonic* when θ is a strictly monotonic homomorphism with the strict subterm property (with respect to $\geq$) and $\geq$ is a well-quasi-ordering;

e. $\langle \theta, \geq \rangle$ is *multiset extracting* when θ is an extraction function which depending on the outermost function symbol returns a multiset of the immediate subterms $\mathcal{I}(t) = \{t_1, t_2, \ldots\}$ of a term t, of the following types:

1. a multiset (including the empty multiset) whose elements are the immediate subterms at specified positions $\mathcal{K}$ ($P_{\mathcal{K}}(t) = \{t_i \ : \ i \in \mathcal{K}\}$),
2. a multiset whose elements are the immediate subterms of rank k, $R_k(\mathcal{I}(t))$, or
3. a multiset whose elements are the immediate subterms of rank k or less ($R_{\leq k}(\mathcal{I}(t)) = \bigcup_{i=1}^{k} R_j(\mathcal{I}(t))$)

and $\geq$ is the multiset ordering $\succsim_{\mathcal{M}}$ induced by a well-quasi-ordering $\succsim$ on terms. (See Dershowitz and Manna [DM79] for more on multiset orderings.)

Simple examples of homomorphisms from terms to the natural numbers are size (number of function symbols, including constants), depth (maximum nesting of function symbols), and weight (sum of weights of function symbols). Size and weight are strictly monotonic; depth is monotonic. A simple example of a precedence uses the ordering $+ > s > 0$ with $+_\theta = \lambda x.\text{“+”}$, $s_\theta = \lambda x.\text{“s”}$, and $0_\theta = \lambda x.\text{“0”}$. (The subterm property is guaranteed for strictly monotonic homomorphisms into well-ordered sets [Der82].) An example of a multiset component ordering is $\theta = R_1$; it extracts the maximal immediate subterms in $\succ$. Another example is $\theta = P_{\{1\}}$ which gives the leftmost subterm.

Definition 7 (Well-Quasi General Path Ordering). Let $\phi_0 = \langle \theta_0, \geq_0 \rangle$, ..., $\phi_k = \langle \theta_k, \geq_k \rangle$ be component orders, where for multiset extraction θ_x component orders, $\geq_x$ is the well-quasi general path ordering $\succsim$ itself. The induced *well-quasi general path ordering* $\succsim$ is defined as follows:

$$s = f(s_1, \ldots, s_m) \succsim g(t_1, \ldots, t_n) = t$$

if either of the two following cases hold:

(1) $s_i \succsim t$ for some s_i, $i = 1, \ldots, m$, or

(2) $s \succ t_1, \ldots, t_n$ and $\Theta(s) \geq_{lex} \Theta(t)$, where $\Theta(s) = \langle \theta_0(s), \ldots, \theta_k(s) \rangle$, and $>_{lex}$ is the lexicographic combination of the component orderings $>_x$.

The equivalence part of the ordering is given by $s \approx t$ if $s \succsim t$ and $t \succsim s$.

3.2 Quasi-ordering Proofs

In this section a series of lemmata will be presented which allow one to prove that the well-quasi general path ordering is a quasi-ordering.

Lemma 8 (Exclusion). *If $s \approx t$ then $s \succsim t$ and $t \succsim s$ are either both true by Case (1) or both true by Case (2).*

Proof. Suppose that $s \succsim t$ by Case (1). Then there is an immediate subterm s_i of s such that $s_i \succsim t$. If $t \succsim s$ by Case (2) then for all immediate subterms s_i it must be that $t \succ s_i$. This is a contradiction.

The other case is shown by a symmetric argument. □

The notation $t|_p$ is used to refer to the subterm of t at position p and the notation $u[s]$ (or $u[s]_p$) indicates that u contains s as a subterm (at position p).

In some cases, it is more convenient to give the position of a subterm in the following manner. The notation s_i refers to the ith immediate subterm of s. The notation $s_{i.j}$ refers to the jth immediate subterm of the ith immediate subterm of s.

The following two lemmata must be shown by simultaneous induction over the height of a term.

Lemma 9 (Strict Subterm). *The well-quasi general path ordering satisfies the strict subterm property $s = f(\ldots, s_i, \ldots) \succ s_i$, for all i.*

Proof. By inductive application of Reflexivity $s_i \approx s_i$ and therefore by Case (1) of the ordering $s \succsim s_i$.

Now suppose that $s \approx s_i$. By the previous lemma (Exclusion) it must be that $s_i \succsim s$ by Case (1).

This implies that $s_{i.j} \succsim s$ for some immediate subterm $s_{i.j}$ of s_i. By induction $s_i \succ s_{i.j}$ and Case (1) of the ordering can be applied to show that $s \succsim s_{i.j}$. But this means that $s_{i.j} \approx s$ and by Exclusion that $s_{i.j} \succsim s$ by Case (1).

This implies that $s_{i.j.k} \succsim s$ for some immediate subterm $s_{i.j.k}$ of $s_{i.j}$. By induction $s_{i.j} \succ s_{i.j.k}$ and a series of applications of Case (1) of the ordering can be used to show that $s \succsim s_{i.j.k}$. But this means that $s_{i.j.k} \approx s$ and by Exclusion that $s_{i.j.k} \succsim s$ by Case (1).

Eventually one reaches a subterm of height one where this argument fails and hence by contradiction $s \not\approx s_i$ leaving $s \succ s_i$. □

Lemma 10 (Reflexivity). *The well-quasi general path ordering $\succsim$ is reflexive ($s \approx s$).*

Proof. This can be shown by proving that $s \succsim s$.

By the previous lemma (Strict Subterm) it is known that $s \succ s_i$ for all s_i. Case (2) can be applied if $\Theta(s) \simeq \Theta(t)$

But this is true for the homomorphism components since they must be quasi-orders and by application of induction on the immediate subterms returned by the extraction components one has $t_i \succsim t_i$ and therefore the multiset ordering on immediate subterms is reflexive as well. ($\{t_{j_1}, \ldots, t_{j_m}\} \approx_{\mathcal{M}} \{t_{j_1}, \ldots, t_{j_m}\}$ for any $j_1, \ldots, j_m \in \{1, \ldots, n\}$).

Therefore Case (2) applies. □

Lemma 11. *For the well-quasi general path ordering, $s \succsim t$ implies $u[s] \succ t$ for each non-empty enclosing context $u[\cdot]$ of s.*

Proof. Consider the subterm $u|_p$ which contains s as an immediate subterm. By Case (1), $u|_p \succ t$. Repeated application of the preceding argument leads to $u[s] \succ t$. □

Lemma 12 (Transitivity). *For terms s, t, and u and well-quasi general path ordering $\succsim$:*

(i) $s \succsim t \succsim u$ *implies* $s \succsim u$;

(ii) $s \succsim t \succ u$ *implies* $s \succ u$;

(iii) $s \succ t \succsim u$ *implies* $s \succ u$;

(iv) $s \approx t \approx u$ *implies* $s \approx u$.

Proof. The proof proceeds by induction over the triple of terms $\langle s, t, u \rangle$ with respect to the sum of the heights of the three terms.

(i) Suppose that $s \succnsim t$ by Case (1) of the ordering, then $s_i \succnsim t$ for some i. Since $t \succnsim u$, one can apply induction on the triple $\langle s_i, t, u\rangle$ to get $s_i \succnsim u$. Therefore $s \succnsim u$. by Case (1) of the ordering.

Suppose that $s \succnsim t$ by Case (2) of the ordering, then $s \succ t_1, \ldots, t_n$ and $\Theta(s) \geq_{lex} \Theta(t)$. Now if $t \succnsim u$ by Case (1) of the ordering, then $t_j \succnsim u$ for some j. But one may apply induction to the triple $\langle s, t_j, u\rangle$ to show $s \succ u$. If $t \succnsim u$ by Case (2) of the ordering, then $t \succ u_1, \ldots, u_m$ and $\Theta(t) \geq_{lex} \Theta(u)$. One may apply induction to each of the triples $\langle s, t, u_k\rangle$ to show that $s \succ u_k$ for each k. If each of the component orders is transitive then $\Theta(s) \geq_{lex} \Theta(u)$. When $\geq_x$ is a well-quasi-ordering there is no problem; when $\geq_x$ is a multiset ordering on immediate subterms, the induction hypothesis is needed. Finally, $s \succnsim u$ by application of Case (2).

(ii) By application of the Part (i), one knows that $s \succnsim u$.

Suppose that $s \approx u$. Then $u \succnsim s$. Combined with $s \succnsim t$ and applying Part (i), one gets $u \succnsim t$. But this contradicts the premise that $t \succ u$, hence $s \succ u$.

(iii) Essentially the same argument as for (ii).

(iv) Applying Part (i) for $s \succnsim t \succnsim u$ and $u \succnsim t \succnsim s$ results in $s \succnsim u$ and $u \succnsim s$. Therefore, by definition, $s \approx u$.

□

Theorem 13. *The well-quasi general path ordering is a quasi-ordering.*

Proof. By the previous lemmata $\succnsim$ is reflexive and transitive. □

3.3 Well-Quasi-Ordering Proof

This section contains the proof that the well-quasi general path ordering is a well-quasi-ordering. First, however, some simple, but useful lemmas will be shown which give conditions under which it is easy to determine if two terms are strictly ordered. An alternate definition for the equivalence part of the relation is also presented.

Lemma 14 (Strictness of Case 1). *If $s \succnsim t$ via case (1), then $s \succ t$.*

Proof. One needs to show that $t \not\gtrsim s$.

It must be the case that there is some subterm s_i of s such that $s_i \gtrsim t$. Therefore Case (2) can not apply.

By the strict subterm property one has that $s \succ s_i \gtrsim t \succ t_j$ for every subterm t_j of t. By transitivity, one sees that for every immediate subterm t_j of t it must be that $t_j \prec s$ and Case (1) does not apply. □

Lemma 15 (Definition of Equivalence). *The associated equivalence relation $\approx$ is given by the condition*

$$s = f(s_1, \ldots, s_m) \approx g(t_1, \ldots, t_n) = t$$

if and only if $s \succ t_1, \ldots, t_n$, $t \succ s_1, \ldots, s_m$ and $\theta_0(s) \simeq_0 \theta_0(t), \ldots, \theta_k(s) \simeq_k \theta_k(t)$.

Proof. If $s \approx t$, then $s \gtrsim t$ and $t \gtrsim s$. But by the previous lemma, it can not be that $s \gtrsim t$ by Case (1). Therefore, Case (2) must have been applied in both directions. □

Lemma 16. *If $s \gtrsim t$ by Case (2) and $\Theta(s) > \Theta(t)$ then $s \succ t$.*

Proof. Suppose that $s \approx t$, then by Lemma 8 it must be the case that $t \gtrsim s$ by Case (2). But since $\Theta(s) > \Theta(t)$ Case (2) is not applicable, and by contradiction $s \succ t$. □

Notice that Case (1) of the ordering is always strict. Case (2) of the ordering is always strict if the lexicographical comparison is strict. When the terms are equivalent under the lexicographical comparison, Case (2) of the ordering may result in the terms being either strict or equivalent.

Theorem 17. *The well-quasi general path ordering is a well-quasi-ordering.*

Proof. Consider an infinite sequence of terms $t^1, t^2, t^3, \cdots$ which comprise a minimal counter example (minimal in the sense that each of the terms is smallest in height that begins a counter example).

Each of the subterms must be well-quasi-ordered with respect to $\gtrsim$ or the sequence wouldn't be minimal. Therefore, since Θ is composed of homomorphisms which are well-quasi-orderings and extractions of subterms over which $\gtrsim$ is also well-quasi-ordered, the lexicographic ordering Θ is well-quasi-ordered over the terms in the minimal counter example.

Therefore, in the sequence $\Theta(t^1), \Theta(t^2), \Theta(t^3), \cdots$ by Proposition 4 there must be an infinite subsequence such that $\Theta(s^1) \leq \Theta(s^2) \leq \Theta(s^3) \leq \cdots$.

Now consider the multisets of the subterms of each of the terms s^i. By Proposition 7, the ordering $\gtrsim$ extended to the multisets of the immediate subterms is also well-quasi-ordered (since $\gtrsim$ is well-quasi-ordered with respect to the subterms themselves.) Therefore, there are i and j such that $i < j$ and $\left\{s_1^i, \ldots, s_m^i\right\} \precsim_{\mathcal{M}} \left\{s_1^j, \ldots, s_n^j\right\}$. Furthermore, for every subterm s_q^i of s^i there is some subterm s_r^j such that $s_q^i \precsim s_r^j$. By application of the strict subterm property one gets $s^j \succ s_q^j$. Applying transitivity, one sees that every subterm s_q^i of s^i satisfies $s^j \succ s_q^i$. By construction $\Theta(s^j) \geq \Theta(s^i)$ and Case (2) is applied to get $s^j \gtrsim s^i$. But this contradicts the assumption that the original sequence of terms was a counterexample. Therefore $\gtrsim$ is a well-quasi-ordering. □

4 TERMINATION PROOFS

A general path ordering can be used to prove termination if certain general conditions are met. The first lemma presented guarantees a strict decrease in the multiset ordering induced by a quasi-ordering. Next a theorem is shown which gives general conditions under which a quasi-ordering can be used to show termination of a rewrite system. Finally, the last theorem gives specific conditions for the component orderings of a well-quasi general path ordering to ensure that the the general conditions are satisfied.

Lemma 18. *If $\gtrsim$ is a quasi-order with the strict subterm property,*

$$s \to t \text{ and } s \gtrsim t \text{ imply } f(\ldots, s, \ldots) \gtrsim f(\ldots, t, \ldots) ,$$

for all terms s, t, ... and function symbols f, and $l\sigma \succ r\sigma$ for all rules $l \to r$ and substitutions σ, then for any rewrite step $u \to v$ $U_{\mathcal{M}} \succ_{\mathcal{M}} V_{\mathcal{M}}$ where $\succ_{\mathcal{M}}$ is the multiset ordering induced by $\gtrsim$, $U_{\mathcal{M}} = \{t | t \text{ is a subterm of } u\}$, and $V_{\mathcal{M}} = \{t | t \text{ is a subterm of } v\}$.

Proof. To begin, note that given a position p, the multiset of subterms can be split into three parts: the subterms at or below p, the subterms above p, and the subterms disjoint from p.

If $u \to v$ then there is some subterm $u|_p$ of u such that $u|_p = l\sigma$. Therefore

$$u = u[l\sigma]_p \to u[r\sigma]_p = v.$$

By assumption $l\sigma \succ r\sigma$. In addition, repeated application of the strict subterm property with transitivity gives $r\sigma \succ r\sigma|_p$ for all proper subterms of $r\sigma$. Thus the subterm $l\sigma$ in $U_{\mathcal{M}}$ is replaced in $V_{\mathcal{M}}$ by the strictly smaller $r\sigma$ and its subterms.

The only other subterms which are affected by the rewrite are those rooted at symbols on the path from $l\sigma$ to the top of u. One can show that $w[l\sigma]_p \gtrsim w[r\sigma]_p$ for all contexts w by induction on the depth of position p in w. If w is the empty context, it is given that $l\sigma \succ r\sigma$. Otherwise, let $w = f(\cdots s[l\sigma]_q \cdots)$. By induction $s[l\sigma]_q \succ s[r\sigma]_q$, and by the given implication $w[l\sigma]_p = f(\cdots s[l\sigma]_q \cdots) \gtrsim f(\cdots s[r\sigma]_q \cdots) = w[r\sigma]_p$. □

Theorem 19 (General Termination). *Let $\succsim$ be a general path ordering. A rewrite system $\mathcal{R}$ terminates if*

- *$l\sigma \succ r\sigma$ for all rules $l \to r$ in $\mathcal{R}$ and substitutions σ and,*
- *$s \to t$ and $s \succsim t$ implies $f(\ldots, s, \ldots) \succsim f(\ldots, t, \ldots)$.*

Proof. Since the general path ordering is a quasi-order with the strict subterm property, by Lemma 18 one knows that each rewrite results in a strict decrease in $\succ_{\mathcal{M}}$. Since $\succ$ is a well-quasi-order it is well-founded as well. Therefore, $\succ_{\mathcal{M}}$ is also well-founded and termination follows. □

Theorem 20 (Specific Termination). *Let $\phi_0, \ldots, \phi_{i-1}$ $(i \geq 0)$ be monotonic, all but possibly the last strict, and let $\phi_i, \ldots, \phi_k$ be precedence, value-preserving, or multiset extraction component orders. A rewrite system terminates if $l\sigma \succ r\sigma$ in the corresponding general path ordering $\succsim$ for all rules $l \to r$ and ground substitutions σ, provided*

(i) if $\theta_x = R_k$ there is some $y < x$ such that $\theta_y = R_{k-1}$ or $\theta_y = R_{\leq k-1}$; and

(ii) $\simeq_x$ is a congruence for each homomorphism θ_x.

Note that whenever $\geq_x$ is a partial-order, congruence is guaranteed.

4.1 Examples

Before giving a proof, consider the following examples illustrating the need for restrictions on the components: (Parentheses are omitted for the unary function symbols 0, 1, f, g.)

Consider the non-terminating two rule rewrite system

$$\begin{aligned} 0011x &\to 111000x \\ 0x &\to 11x\ . \end{aligned} \tag{4.1}$$

A well-quasi general path ordering with first component, the precedence $0 > 1$, and the second, the strictly monotonic homomorphism which counts the number of symbols in a term, shows a decrease for both rules. But this violates the condition requiring monotonic homomorphisms to precede the other types of component orderings.

Consider the non-terminating two rule rewrite system

$$\begin{aligned} ffx &\rightarrow fgfx \\ gx &\rightarrow x\ . \end{aligned} \tag{4.2}$$

A well-quasi general path ordering with first component, a monotonic homomorphism θ_{ff} which counts the number of pairs of f's, and second, the precedence $f > g$, shows a decrease for both rules. But this violates the condition requiring that homomorphisms be congruences, since $\theta_{ff}(f(g(a))) \neq \theta_{ff}(f(f(a)))$ even though $\theta_{ff}(g(a)) = \theta_{ff}(f(a))$.

Consider the non-terminating two rule rewrite system

$$\begin{aligned} h(a,b) &\rightarrow h(a,a) \\ a &\rightarrow b\ . \end{aligned} \tag{4.3}$$

A well-quasi general path ordering with first component, the precedence $f > a > b$, and second, the multiset extraction of rank two, shows a decrease for both rules, since $\{b\} > \emptyset$. But this violates the condition requiring that the rank extracting component be preceded by a rank extracting component which extracts terms of rank one.

4.2 Proof of Termination of GPO with Restrictions

The following is the proof of Theorem 20. It proceeds by considering $\approx$ and $\succ$ separately.

Proof. By Theorem 19, it suffices to show

$$s \rightarrow t \text{ and } s \succsim t \ \text{ imply } \ u = f(\ldots s \ldots) \succsim f(\ldots t \ldots) = v\ ,$$

for all terms s, t, ... and function symbols f.

Consider the case that $s \approx t$. To complete the proof it will be shown that $u \approx v$. If $s \approx t$, then by lemma 15 $s \succ t_1, \ldots, t_m$ and $t \succ s_1, \ldots, s_n$ and $\Theta(s) \simeq_{lex} \Theta(t)$. For each of the subterms $v_i \neq t$, it is the case that $u_i = v_i$. For the subterm t itself, $s \approx t$, and consequently $u = f(\ldots s \ldots) \succ v_i$ for each i by Case (1) of the ordering. Similarly, $v = f(\ldots t \ldots) \succ u_j$ for each j. One just needs to show that $\theta_x(u) \simeq_x \theta_x(v)$ for each component order so that lemma 15 can be applied.

For precedence and value-preserving component orders this is trivial. For monotonic component orders, the extra condition guarantees that $\simeq_x$ is a congruence and hence $\theta_x(f(\ldots s \ldots)) = f_{\theta_x}(\ldots \theta_x(s) \ldots) \simeq_x f_{\theta_x}(\ldots \theta_x(t) \ldots) = \theta_x(f(\ldots t \ldots))$.

For θ_i that return multisets, one needs to consider each of the extraction functions separately:

1. Extract subterms at positions $\mathcal{K}$. If $s \neq u_k$ for any $k \in \mathcal{K}$, then each $u_k = v_k$ and $P_{\mathcal{K}}(u) = P_{\mathcal{K}}(v)$. Otherwise, the multisets are identical except that s is replaced by t and therefore $P_{\mathcal{K}}(u) \approx_{\mathcal{M}} P_{\mathcal{K}}(v)$.

2. Extract subterms of rank k. Since s is equivalent to t, they have the same rank. Therefore $R_k(\mathcal{IS}(u)) \approx_{\mathcal{M}} R_k(\mathcal{IS}(v))$ for all k.

3. Extract terms of rank k or less. By the same argument as in the previous case, $R_{\leq k}(\mathcal{IS}(u)) \approx_{\mathcal{M}} R_{\leq k}(\mathcal{IS}(v))$.

The proof now turns to the strict case, $s \succ t$. As before one can show that $u \succ v_i$ for each i. It just remains to be shown that $\Theta(u) \geq_{lex} \Theta(v)$ in order to apply Case (2) of the well-quasi general path ordering. Note that for the recursive definition to give $s \succ t$, there must be some subterm (possibly non-proper) $s|_p$ of s such that $s|_p \succsim t$ by Case (2) of the ordering and hence $\Theta(s|_p) \geq_{lex} \Theta(t)$. Consider a monotonic homomorphism ϕ_x. There are two cases:

Case A ($s|_p = s$): Suppose that θ_y with $y \leq x$ is the first monotonic homomorphism which shows an increase. For each of the preceding homomorphisms $\theta_z(s) \simeq_z \theta_z(t)$ and therefore $\theta_z(f(\ldots, s, \ldots)) \simeq_z \theta_z(f(\ldots, t, \ldots))$ by congruence for $z \leq y$, while for the yth homomorphism $\theta_y(s) >_y \theta_y(t)$. If the homomorphism is strict, this implies $\theta_y(f(\ldots, s, \ldots)) >_y \theta_y(f(\ldots, t, \ldots))$ and the lexicographic comparison is strictly greater. If the homomorphism is not strict, then $\theta_y(f(\ldots, s, \ldots)) \geq_y \theta_y(f(\ldots, t, \ldots))$ and the status of the lexicographical comparison may depend on the succeeding component orderings.

Case B ($s|_p \neq s$): Consider θ_0. By repeated application of the strict subterm property of the monotonic homomorphism components, one has $\theta_0(s) >_0 \theta_0(s|_p) \geq_0 \theta_0(t)$. If θ_0 is strict, this implies $\theta_0(f(\ldots, s, \ldots)) >_0 \theta_0(f(\ldots, t, \ldots))$ and the lexicographic comparison is strictly greater. If θ_0 is not strict, then $\theta_0(f(\ldots, s, \ldots)) \geq_0 \theta_0(f(\ldots, t, \ldots))$ and the status of the lexicographical comparison may depend on the succeeding component orderings.

In either case, any component orderings following a non-strict homomorphism need not show an increase for s or $s|_p$, respectively, compared with t. As a consequence, none of the succeeding component orderings may safely rely on the lexicographic status of s or its subterms. In addition, since the monotonic homomorphisms depend on the lexicographical status of subterms, it is not safe to have other types of component orders preceding. This is the reason for the restrictions:

- there may only be one non-strict monotonic homomorphism and each of the strict monotonic homomorphisms must precede it, and
- no other type of component ordering may precede a monotonic homomorphism.

Consider now a value-preserving homomorphism and a rewrite $s = c[l\sigma] \rightarrow c[r\sigma] = t$. It is given that $\theta(l\sigma) \simeq_x \theta(r\sigma)$. Combined with congruence of the ordering this results in $\theta(f(\ldots, s, \ldots)) \simeq_x \theta(f(\ldots, t, \ldots))$.

When the termination function is a precedence, its value does not depend on subterms and trivially $\theta(f(\ldots, s, \ldots)) \simeq_x \theta(f(\ldots, t, \ldots))$.

Now consider component orderings that compare multisets of subterms:

1. Extract subterms at positions in $\mathcal{K}$. If $s \neq u_k$ for all $k \in \mathcal{K}$, then each $u_k = v_k$ and $P_{\mathcal{K}}(u) = P_{\mathcal{K}}(v)$. Otherwise the multisets are identical except that s is replaced by t and therefore $P_{\mathcal{K}}(u) \succ_{\mathcal{M}} P_{\mathcal{K}}(v)$.

2. Extract subterms of rank k. Suppose that $s \in R_i(\mathcal{IS}(u))$. Then there is no change in multisets of rank less than i. For the multiset of rank i, the only possible new members are t and terms from R_{i+1} that were dominated by s. Thus $R_i(\mathcal{IS}(u)) \succ_{\mathcal{M}} R_i(\mathcal{IS}(v))$. If $k > i$, there may be an increase, but it is guaranteed that either R_i or some $R_{\leq j}$ containing rank i is before θ_x lexicographically, and either of these will show an increase.

3. Extract subterms of rank less than or equal k. Suppose $s \in R_i(u)$. By an argument similar to that above, $R_{\leq k}(\mathcal{IS}(u)) = R_{\leq k}(\mathcal{IS}(v))$ for $k < i$ and $R_{\leq k}(\mathcal{IS}(u)) \succ_{\mathcal{M}} R_{\leq k}(\mathcal{IS}(v))$ for $k = i$. One just needs to consider the case $k > i$. Think of the process of going from $R_{\leq k}(\mathcal{IS}(u))$ to $R_{\leq k}(\mathcal{IS}(v))$ as adding t to the set of immediate subterms then removing s. When t is added other terms may move to higher rank, but not lower rank. So the only possible new term in $R_{\leq k}(\mathcal{IS}(u) \cup \{t\})$ is t. When s is removed, terms may be added from

rank $k+1$ (note that terms may only move one rank position when a single term is added or deleted). Consider a term w of rank $j+k+1$ which is a member of $R_{\leq k}(\mathcal{IS}(v))$, but was not a member of $R_{\leq k}(\mathcal{IS}(u) \cup \{t\})$. It must have been added because a term x_k of rank k moved to rank $k-1$ and $x_k \succ w$. Inductively, a chain of terms can be constructed such that $x_i \succ x_{i+1} \succ \cdots \succ x_k \succ w$. But there was only the single term s which was removed at level i and therefore $s = x_i \succ w$. In combination with $s \succ t$, it must be that $R_{\leq k}(\mathcal{IS}(u)) \gtrsim R_{\leq k}(\mathcal{IS}(v))$.

□

Whereas only lexicographic and multiset mappings are used in the general path orderings, in [KL80], Kamin and Lévy consider the more general case of orderings based on a mapping $\triangleright$ from well-founded quasi-orderings to well-founded quasi-orderings. They allow a component order $\theta t = \langle t_1, \ldots, t_n \rangle$ and $\geq = \triangleright \succ$, where $\triangleright$ recursively makes finitely many comparisons of subterms. In particular, one can use weighted multisets, as in Martin [Mar89].

5 INSTANCES OF THE WELL-QUASI GENERAL PATH ORDERING

In this chapter it is shown that several common orderings are specific instances of the well-quasi general path ordering. A new kind of ordering called the "Natural Path Ordering" is defined which is a instance of a General Path Ordering. It combines precedence with a value-preserving homomorphism. An example of its use is then presented.

5.1 Existing Simplification Orderings

The following simplification orderings are special cases of the well-quasi general path ordering for which the conditions of Theorem 20 hold. The only caveat is that the precedences used in the orderings must be well-quasi-orderings instead of just well-founded. The precedence used in the recursive path ordering is well-quasi-ordered since it is required to be total. For the other orderings, as long as the signature of the rewrite system is finite, one can restrict the precedence to those symbols and it will be well-quasi-ordered. Therefore, they can then be applied to any finite rewrite system.

Knuth-Bendix ordering (Knuth and Bendix [KB70]) θ_0 gives the sum of (non-negative integer) "weights" of the function symbols appearing in a term; $\geq_0$ is the $\geq$ ordering on the natural numbers; ϕ_1 gives a (total) precedence; $\phi_2, \ldots, \phi_{n+1}$ give (a permutation of) the immediate subterms.

Polynomial path ordering (Lankford [Lan79]) θ_0 is a strict monotonic homomorphism with each f_θ a polynomial with positive integer coefficients; $\geq_0$ is the $\geq$ ordering on the natural numbers; ϕ_1 gives a precedence; $\phi_2, \ldots, \phi_{n+1}$ give a permutation of the immediate subterms.

Multiset path ordering (the original version of the "recursive path ordering", Dershowitz [Der82]) ϕ_0 is a precedence; ϕ_1 extracts the multiset of immediate subterms.

Extended path ordering (Dershowitz [Der82]) ϕ_0 extracts one of the immediate subterms; ϕ_1 extracts a multiset of the remaining immediate subterms.

Lexicographic path ordering (Kamin and Lévy [KL80]) ϕ_0 is a precedence; $\phi_1, \ldots, \phi_n$ give a permutation of the subterms.

Recursive path ordering ("with status", Lescanne [Les90]) ϕ_0 is a total precedence; $\phi_1, \ldots, \phi_n$ give a permutation of the subterms or multisets of subterms, depending on the function symbol.

5.2 The Natural Path Ordering

The following is not a simplification ordering or an instance of the well-quasi general path ordering, but it is an instance of the well-founded general path order (to be discussed in Chapter 7).

Value-preserving path ordering (Plaisted [Pla79], Kamin and Lévy [KL80]) θ is a value-preserving homomorphism and $\geq$ is a well-founded quasi-order; ϕ_0 is a precedence; θ_1 is θ applied to the first subterm and $\geq_1$ is $\geq$; θ_2 is θ applied to the second subterm and $\geq_2$ is $\geq$; and so forth.

As an example of the use of the value-preserving path ordering, consider System 1.2. The precedence is $fact >_0 \times >_0 + >_0 s$; θ_1 interprets everything naturally: $fact$ as factorial, s as successor, p as predecessor, $\times$ as multiplication, $+$ as addition, and 0 as zero. The ordering $\geq_1$ is the well-founded greater-than relation on natural numbers. Let all constants be interpreted as natural numbers, making all terms non-negative. Each rule causes a strict decrease with respect to the general path ordering and the rewrite system terminates. This approach works for primitive-recursive functions in general.

Note that to use a natural interpretation, one must always make sure that all terms and subterms in any derivation are interpretable as natural numbers; otherwise a rule like $fact(x) \rightarrow fact(p(x))$ would give pretense of being terminating.

The idea embodied in the value-preserving ordering is enlarged in the following way, intended to mirror the standard structural induction proof method for recursive programs:

Definition 8 (Natural Path Ordering). A *natural path ordering* is a special case of the well-quasi general path ordering with two component orderings: ϕ_0 is a precedence and ϕ_1 is defined for each f (of arity n), as $\theta_1 f(t_1, \ldots, t_n) = f_{\theta_1}(\theta_1 t_1, \ldots, \theta_1 t_n)$, where θ_1 is a value-preserving homomorphism to some arbitrary algebra $\mathcal{A}$, and f_{θ_1} a mapping from $\mathcal{A}^n$ to a well-quasi-ordered set $(W, \geq)$.

Theorem 19 applies.

As an example, consider the following rewrite system for computing the average of two integers:

$$\begin{array}{rcl} a(sx, y) & \rightarrow & a(x, sy) \\ a(x, ssy) & \rightarrow & sa(sx, y) \\ a(0, 0) & \rightarrow & 0 \\ a(0, s0) & \rightarrow & 0 \\ a(0, ss0) & \rightarrow & s0 \ . \end{array} \tag{5.1}$$

A multiset path ordering will not work for the arguments of a in the first rule and a lexicographical path ordering will not work for the first two rules. The natural path ordering is sufficient for proving termination with ϕ_0 as $a >_0 s >_0 0$ and ϕ_1 given by $\theta_1(a(x, y)) = 2\theta(x) + \theta(y)$, where θ is the value-preserving homomorphism: $a_\theta = \lambda xy.\lfloor \frac{x+y}{2} \rfloor$, $s_\theta = \lambda x.x + 1$, and $0_\theta = \lambda x.0$.

6 WQGPO EXAMPLES

In this section several examples are presented applying the well-quasi general path ordering to specific rewrite systems to prove termination. The examples are self-embedding, so the standard simplification orderings do not work. The first example is an insertion sort over the natural numbers. It is shown to be terminating without the use any semantic interpretation of terms. This allows for an easier proof of termination without the need for any inductive arguments or arguments over the natural numbers. The next two examples are conditional rewrite systems and use value-preserving homomorphisms with the well-quasi general path ordering. The third example is an insertion sort over the integers and can be shown to decrease with a monotonic homomorphism combined with a precedence and an extraction.

6.1 Insertion Sort for Natural Numbers

The following rewrite system sorts a list of natural numbers into decreasing order via an insertion sort:

$$sort(nil) \rightarrow nil \tag{5.6.1}$$

$$sort(cons(x,y)) \rightarrow insert(x, sort(y)) \tag{5.6.2}$$

$$insert(x, nil) \rightarrow cons(x, nil) \tag{5.6.3}$$

$$insert(x, cons(v,w)) \rightarrow choose(x, cons(v,w), x, v) \tag{5.6.4}$$

$$choose(x, cons(v,w), y, 0) \rightarrow cons(x, cons(v,w)) \tag{5.6.5}$$

$$choose(x, cons(v,w), 0, s(q)) \rightarrow cons(v, insert(x,w)) \tag{5.6.6}$$

$$choose(x, cons(v,w), s(p), s(q)) \rightarrow choose(x, cons(v,w), p, q)\ . \tag{5.6.7}$$

Note that these rules are locally confluent since there are no critical pairs, and the proof of termination will imply that they are confluent as well.

Four component orders are used. They are

$$\begin{aligned}
\phi_0 = &\quad \text{the precedence } \mathit{sort} > \mathit{insert} \simeq \mathit{choose} > \mathit{cons} \\
\phi_1 = &\quad \text{the extraction based on the outermost symbol } f \\
&\theta_1 = \begin{cases} P_{\{1\}} & f = \mathit{sort} \\ P_{\{2\}} & f = \mathit{choose}, \mathit{insert} \\ \emptyset & \text{otherwise} \end{cases} \\
\phi_2 = &\quad \text{the precedence } \mathit{sort} > \mathit{insert} > \mathit{choose} > \mathit{cons} \\
\phi_3 = &\quad \text{the extraction based on the outermost symbol } f \\
&\theta_3 = \begin{cases} P_{\{1\}} & f = \mathit{sort} \\ P_{\{2\}} & f = \mathit{insert} \\ P_{\{3\}} & f = \mathit{choose} \\ \emptyset & \text{otherwise .} \end{cases}
\end{aligned}$$

The ordering interleaves precedences with recursive comparisons of subterms and thus is unlike either the semantic path ordering [KL80] or semantic labeling [Zan92b]. No semantic interpretation of the function symbols is required to prove termination in this example.

If one were to use an ordering just based on the precedence ϕ_2, all of the rules except for the sixth would be oriented in the appropriate direction. Unfortunately, the fourth and sixth rules interact with each other. In particular, there is a *choose* and an *insert* on opposite sides of each rule. The precedence ϕ_0 is chosen to guarantee a decrease in the lexicographical part when ordering Rule 6 by Case (2) of the well-quasi general path ordering while leaving Rule 4 equal. Since Rule 6 is ordered by Case (2) of the ordering with a strict decrease in the lexicographical comparison, Lemma 16 applies and the decrease is strict for application of Rule (6). The first condition for Case (2) requires that the left-hand side of Rule 6 be strictly greater than each of the two subterms on the right. The non-trivial comparison is $\mathit{choose}(x, \mathit{cons}(v, w), 0, s(q))$ with $\mathit{insert}(x, w)$. These terms are equal under the precedence ordering ϕ_0, but by selecting the second subterm of both *choose* and *insert* a strict decrease in the lexicographic part is achieved. This allows us to conclude via Lemma 16 that $\mathit{choose}(x, \mathit{cons}(v, w), 0, s(q)) \succ \mathit{insert}(x, w)$. Therefore, Rule 6 is correctly ordered.

Now consider Rule 4. Fortunately, the second subterm on both sides of Rule 4 is identical, leaving the lexicographical order unaffected. The precedence ordering ϕ_2 breaks that tie and guarantees a decrease via Lemma 16. Verifying the first condition of Case (2) for Rule 4 is easy.

Rule 1 is a trivial application of Case (1). Rule 2 is nearly as trivial. The only observation to make is that the first condition for Case (2) requires $sort(cons(x, y)) \succ sort(y)$, which itself requires an application of Case (2) where the lexicographic part requires the extraction and comparison of $cons(x, y)$ with y which gives a strict decrease in the lexicographic comparison. Rules 3 and 5 are also straightforward.

Rule 7 meets the first conditions for Case (2), but is equal for the lexicographical part with respect to the first three component orderings. The addition of a fourth component breaks the tie by extracting the third subterm for *choose* (the fourth subterm would also have worked).

Therefore, by the well-quasi general path ordering, this system of rules terminates.

6.2 Using Extraction Components Non-Recursively with Conditional Rules

A *conditional rule* is an equational implication in which the conclusion is a rewrite rule. The following system incorporates both conditional and unconditional rules:

$$\begin{array}{rrcl} & p0 & \to & 0 \\ & psx & \to & x \\ & pos?\ sx & \to & true \\ & pos?\ 0 & \to & false \\ & f0 & \to & 0 \\ pos?\ x \simeq true \ \Rightarrow & fx & \to & fpx \end{array} \tag{6.6}$$

In general, if the conditions of a rule,

$$u_1 = v_1 \wedge \cdots \wedge u_n = v_n \ \Rightarrow \quad l \quad \to \quad r,$$

are satisfied for a particular instance of the left-hand side, the rule rewrites any term containing the instance. The conditions are satisfied if $u_i\sigma \to \cdots \to w$ and $v_i\sigma \to \cdots \to w$ (for some w), for each condition $u_i = v_i$ and for substitution σ.

Verifying termination of conditional systems can be considerably more difficult than for most unconditional systems. More often than not, purely syntactic approaches fail. Consider the very simple system above. To prove termination, one needs to find a measure τ that decreases with each rule application. In particular, it must be that $\tau(x) > \tau(px)$ for all x satisfying the condition in the last rule. For that, one must first characterize those x such that *pos*? x rewrites to *true*. This is in contrast to the unconditional case [Der87], in which termination can usually be separated from other aspects of correctness.

In the following examples, a value preserving homomorphism will be used when showing termination. The value preserving homomorphism is exploited by the joinability condition. If the terms are joinable, they must have the same interpretation under the value-preserving homomorphism, and this will be used when showing that the rule decreases. In particular, it is convenient to extract a multiset of subterms and compare with the value preserving homomorphism. This case, however, is not covered by the standard definition of the well-quasi general path ordering since it only allows comparisons recursively in the general path ordering. Therefore, it must be shown that with a value-preserving homomorphism that Theorem 19 will be satisfied.

Theorem 21. *Let $\phi_0, \ldots, \phi_k$ be component orderings and $\mathcal{R}$ be a rewrite system that satisfy the conditions of Theorem 20 (specific termination) with the exception that certain position based extraction components are compared in $\geq$ which is* not $\gtrsim$ *but some homomorphism $\geq_{vp}$. If $\geq_{vp}$ is value-preserving with with respect to $\mathcal{R}$, then $\mathcal{R}$ terminates.*

Proof. As in the proof of specific termination, given that

$$u = f(\ldots s \ldots) \gtrsim f(\ldots t \ldots) = v \ ,$$

and $s \to t$, one just needs to show that $\theta_x(u) \geq_x \theta_x(v)$ for the extraction components.

Since $\geq_{vp}$ is value preserving with respect to $\mathcal{R}$ and it is a congruence, it must be the case that if $s \to t$ then $s \geq_{vp} t$. Any position based extraction component will extract the same terms with the possible exception that s is replaced by t. In $\geq_x$, which is the multiset extension of $\geq_{vp}$, the multisets are equivalent and hence $\theta_x(u) \geq_x \theta_x(v)$. □

The next two examples are conditional rewrite systems and use value-preserving homomorphisms with an extraction component in the well-quasi general path ordering.

6.2.1 Greatest Common Divisor (GCD)

In this section there related conditional rewrite systems are presented for computing the greatest common divisor. Each is shown to be terminating with related instances of the well-quasi general path ordering using value preserving homomorphisms.

Consider the following recursive program for computing the greatest common divisor:

function $gcd(x, y)$
begin
    if $y = 0$ **then** x
    elseif $y > x$ **then** $gcd(y, x)$
    else $gcd(x - y, y)$
end.

This program can be translated into the following (infinite) conditional rewrite system:

$$
\begin{array}{rrcll}
y\ gt\ x \downarrow t & gcd(x, y) & \rightarrow & gcd(y, x) & \\
x\ ge\ s(y) \downarrow t & gcd(x, s(y)) & \rightarrow & gcd(x - s(y), s(y)) & \\
 & gcd(x, 0) & \rightarrow & x & \\
 & s(x)\ gt\ s(y) & \rightarrow & x\ gt\ y & \\
 & s^i(0)\ gt\ 0 & \rightarrow & t & i \geq 1 \\
 & s(x)\ ge\ s(y) & \rightarrow & x\ ge\ y & \\
 & s^i(0)\ ge\ 0 & \rightarrow & t & i \geq 0 \\
 & s(x) - s(y) & \rightarrow & x - y & \\
 & x - 0 & \rightarrow & x\ . &
\end{array}
\tag{6.7}
$$

Notice that without the conditions this rewrite system is non-terminating. In addition, the left-hand side of the second rule embeds in the right-hand side, so no simplification ordering can be used to show termination. Hence, one wants to try interpretations where $x - s(y)$ is less than x.

The reason for using an infinite set of rewrite rules schematically represented by $s^i(0)$ for gt and ge is so that a value-preserving homomorphism can be constructed easily. The more natural rule, $s(x)\ gt\ 0 \rightarrow t$, permits terms which do not have interpretations as natural numbers to be greater than zero. Alternatively, one could use membership or sorts to express the above restriction.

To show that the conditional rewrite system is terminating, the well-quasi general path ordering will be used with a value-preserving homomorphism, $\theta_{\mathcal{H}}$ which maps to a well-founded set. The range of the homomorphism consists of the natural numbers, *true*, and $\perp$ (this set will be denoted as $\mathcal{NAT}_{\perp,true}$). The well-founded ordering, $>_{\mathcal{H}}$, is the standard greater than ordering on the natural numbers combined with, $0 > \perp$, and $\perp >$ *true*. The homomorphism is:

$$
\begin{array}{rll}
gcd :: & \lambda x, y. & \textbf{if } x = \perp \textbf{ or } y = \perp \textbf{ then } \perp \\
 & & \textbf{elseif } x = true \textbf{ or } y = true \textbf{ then } true \\
 & & \textbf{else } gcd(x, y). \\
- :: & \lambda x, y. & \textbf{if } x = \perp \textbf{ or } y = \perp \textbf{ then } \perp \\
 & & \textbf{elseif } x = true \textbf{ or } y = true \textbf{ then } true \\
 & & \textbf{elseif } x \geq y \textbf{ then } x - y \\
 & & \textbf{else } \perp. \\
gt :: & \lambda x, y. & \textbf{if } x = \perp \textbf{ or } y = \perp \textbf{ then } \perp \\
 & & \textbf{elseif } x = true \textbf{ or } y = true \textbf{ then } \perp \\
 & & \textbf{elseif } x > y \textbf{ then } true \\
 & & \textbf{else } \perp. \\
ge :: & \lambda x, y. & \textbf{if } x = \perp \textbf{ or } y = \perp \textbf{ then } \perp \\
 & & \textbf{elseif } x = true \textbf{ or } y = true \textbf{ then } \perp \\
 & & \textbf{elseif } x \geq y \textbf{ then } true \\
 & & \textbf{else } \perp. \\
s :: & \lambda x. & \textbf{if } x = \perp \textbf{ then } \perp \\
 & & \textbf{elseif } x = true \textbf{ then } true \\
 & & \textbf{else } x + 1. \\
0 :: & 0. & \\
t :: & true. &
\end{array}
\tag{6.8}
$$

The component orderings are combined in the following way:

$$
\begin{aligned}
\phi_0 = &\quad \text{the precedence } gcd > gt > ge > - > s > 0 > t. \\
\phi_1 = &\quad \text{the extraction based on the outermost symbol } f \\
&\quad \theta_1 = \begin{cases} P_{\{1\}} & f = -, gt, \text{ or } ge \\ \emptyset & \text{otherwise} \end{cases} \\
&\quad \text{with } >_1 = \succ \text{ (applied recursively).} \\
\phi_2 = &\quad \text{the extraction based on the outermost symbol } f \\
&\quad \theta_2 = \begin{cases} P_{\{2\}} & f = gcd \\ \emptyset & \text{otherwise} \end{cases} \\
&\quad \text{with } >_2 = >_{\mathcal{H}}. \\
\phi_3 = &\quad \text{the extraction based on the outermost symbol } f \\
&\quad \theta_3 = \begin{cases} P_{\{1\}} & f = gcd \\ \emptyset & \text{otherwise} \end{cases} \\
&\quad \text{with } >_3 = >_{\mathcal{H}}
\end{aligned}
\tag{6.9}
$$

First, one must verify that the homomorphism $\theta_{\mathcal{H}}$ is value-preserving for each of the rewrite rules. Of particular interest are the three rules for gcd. The first rule is value-preserving independent of the condition. For the second rule, the joinability of the two terms in the condition requires that the interpretations be the same (provided that all the rules are value-preserving). In this case, the interpretation of $x\ ge\ s(y)$ is *true* only if $\theta_{\mathcal{H}}(x) \geq_{\mathcal{H}} \theta_{\mathcal{H}}(y) + 1$ with both $\theta_{\mathcal{H}}(x)$ and $\theta_{\mathcal{H}}(y)$ natural numbers. With this condition and knowledge of the gcd function, one can then show that the rewrite rule is value preserving if the condition is met. The third rule is easily shown to be value preserving and is the reason that $gcd(t, 0)$ is mapped to *true* instead $\perp$ (as one might have expected).

The proof of termination with the well-quasi general path ordering using the component orderings specified above proceeds as for the unconditional case with the following exceptions. First, the conditions on the interpretation may be used in the proofs of termination for the rules. Second, the left-hand side of the rule must be larger than each of the terms in the condition. (The second term in each of the conditions is a ground term in normal form, so for this particular rewrite system one need only consider the first term.)

All of the non-conditional rules are handled by the precedence or Case (1) of the well-quasi general path ordering. For the first rule, the left and right-hand sides are equal under precedence. With the second component ordering, one compares $\theta_{\mathcal{H}}(y)$ with $\theta_{\mathcal{H}}(x)$. Normally, one would not be able to prove this, but, the joinability of the conditional part gives $\theta_{\mathcal{H}}(y) >_{\mathcal{H}} \theta_{\mathcal{H}}(x)$. For the second rule, the left and right-hand sides are equal under both the precedence and the second component ordering. With the third component ordering, one compares $\theta_{\mathcal{H}}(x)$ with $\theta_{\mathcal{H}}(x - s(y))$. By the condition, one knows that both $\theta_{\mathcal{H}}(x)$ and $\theta_{\mathcal{H}}(y)$ are natural numbers, so one needs to show that $\theta_{\mathcal{H}}(x) >_{\mathcal{H}} \theta_{\mathcal{H}}(x) - \theta_{\mathcal{H}}(y) - 1$. By the condition, it must be that $\theta_{\mathcal{H}}(x) \geq_{\mathcal{H}} \theta_{\mathcal{H}}(y) + 1$. Thus, one only needs to show $\theta_{\mathcal{H}}(y) + 1 >_{\mathcal{H}} 0$, but $\theta_{\mathcal{H}}(y)$ is a natural number, so this is true. (Note, one could have argued that if $\theta_{\mathcal{H}}(x) <_{\mathcal{H}} \theta_{\mathcal{H}}(y) + 1$ then $\theta_{\mathcal{H}}(x - s(y)) = \bot$, which is less than any natural number.)

An alternative formulation of the conditional rewrite system for gcd is obtained by noticing that the rules for gt and ge are nearly the same as the rules for subtraction.

$$\begin{array}{rrcll} y - x \downarrow s^i(0) & gcd(x, y) & \rightarrow & gcd(y, x) & i \geq 1 \\ x - s(y) \downarrow s^i(0) & gcd(x, s(y)) & \rightarrow & gcd(x - s(y), y) & i \geq 0 \\ & gcd(x, 0) & \rightarrow & x & \\ & s(x) - s(y) & \rightarrow & x - y & \\ & x - 0 & \rightarrow & x\ . & \end{array} \tag{6.10}$$

One needs an infinite set of rules for similar reasons. Now the well-founded set for the value preserving homomorphism, $\theta_{\mathcal{H}_2}$, only needs the addition of $\bot$. The homomorphism is:

$$\begin{array}{rll} gcd :: & \lambda x, y. & \textbf{if } x = \bot \textbf{ or } y = \bot \textbf{ then } \bot \\ & & \textbf{else } gcd(x, y). \\ - :: & \lambda x, y. & \textbf{if } x = \bot \textbf{ or } y = \bot \textbf{ then } \bot \\ & & \textbf{elseif } x \geq y \textbf{ then } x - y \\ & & \textbf{else } \bot. \\ s :: & \lambda x. & \textbf{if } x = \bot \textbf{ then } \bot \\ & & \textbf{else } x + 1. \\ 0 :: & 0. & \end{array} \tag{6.11}$$

The component orderings are as they were before except that the precedence does not need t, gt, and ge; and the ordering used for the components which extract subterms of gcd is $>_{\mathcal{H}_2}$. (Notice that the old ordering, $>_{\mathcal{H}}$, is an extension of $>_{\mathcal{H}_2}$.) The termination argument is similar.

The final version of gcd replaces the rule schemata with a condition which tests a term to see if it is an natural number.

$$\begin{array}{lrcl}
nat(y - s(x)) \downarrow t & gcd(x, y) & \rightarrow & gcd(y, x) \\
nat(x - s(y)) \downarrow t & gcd(x, s(y)) & \rightarrow & gcd(x - s(y), y) \\
 & gcd(x, 0) & \rightarrow & x \\
 & s(x) - s(y) & \rightarrow & x - y \\
 & x - 0 & \rightarrow & x \\
 & nat(0) & \rightarrow & t \\
 & nat(s(x)) & \rightarrow & nat(x)\ .
\end{array} \tag{6.12}$$

As in the original version, the well-founded set used with the value preserving homomorphism, $\theta_{\mathcal{H}_3}$, will include *true*. The homomorphism is:

$$\begin{array}{rll}
gcd :: & \lambda x, y. & \textbf{if } x = \bot \textbf{ or } y = \bot \textbf{ then } \bot \\
 & & \textbf{elseif } x = true \textbf{ or } y = true \textbf{ then } true \\
 & & \textbf{else } gcd(x, y). \\
- :: & \lambda x, y. & \textbf{if } x = \bot \textbf{ or } y = \bot \textbf{ then } \bot \\
 & & \textbf{elseif } x = true \textbf{ or } y = true \textbf{ then } true \\
 & & \textbf{elseif } x \geq y \textbf{ then } x - y \\
 & & \textbf{else } \bot. \\
nat :: & \lambda x. & \textbf{if } x = \bot \textbf{ or } x = true \textbf{ then } \bot \\
 & & \textbf{else } true. \\
s :: & \lambda x. & \textbf{if } x = \bot \textbf{ then } \bot \\
 & & \textbf{elseif } x = true \textbf{ then } true \\
 & & \textbf{else } x + 1. \\
0 :: & 0. & \\
t :: & true. &
\end{array} \tag{6.13}$$

The component orderings are as they were before except that the precedence is $gcd > - > nat > s > 0 > t$; and the ordering used for the components which extract subterms of gcd is $>_{\mathcal{H}_3}$. The termination argument, though slightly more complicated, is similar.

6.2.2 The "91" Function

One well know example of a recursive function is the "91" function given by:

$$\begin{aligned} F_{91}(x) &= \textbf{if } x > 100 \text{ then } x - 10 \\ &\quad \text{else } F_{91}(F_{91}(x + 11)) \ . \end{aligned} \tag{6.14}$$

This recursive function returns 91 if $x \leq 100$, otherwise it returns $x - 10$.

A conditional rewrite system corresponding to this recursive function is:

$$\begin{aligned} nat(x - 101) \downarrow t \qquad F(x) &\rightarrow x - 10 \\ nat(100 - x) \downarrow t \qquad F(x) &\rightarrow F(F(s^{11}(x))) \\ s(x) - s(y) &\rightarrow x - y \\ x - 0) &\rightarrow x \\ nat(0) &\rightarrow t \\ nat(s(x)) &\rightarrow nat(x) \end{aligned} \tag{6.15}$$

A value preserving interpretation of this rewrite system can be constructed over the set of natural numbers augmented by $\perp$ and *true*. The mapping $\mathcal{H}$ from terms to $\mathcal{NAT}_{\perp,true}$ is given by the following definitions:

$$
\begin{array}{rll}
F :: & \lambda x. & \textbf{if } x = \bot \textbf{ or } x = \mathit{true} \textbf{ then } \bot \\
 & & \textbf{elseif } x \leq 100 \textbf{ then } 91 \\
 & & \textbf{else } x - 10. \\
- :: & \lambda x, y. & \textbf{if } x = \bot \textbf{ or } y = \bot \textbf{ then } \bot \\
 & & \textbf{elseif } x = \mathit{true} \textbf{ or } y = \mathit{true} \textbf{ then } \mathit{true} \\
 & & \textbf{elseif } x \geq y \textbf{ then } x - y \\
 & & \textbf{else } \bot. \\
s :: & \lambda x. & \textbf{if } x = \bot \textbf{ or } x = \mathit{true} \textbf{ then } \bot \\
 & & \textbf{else } x + 1. \\
\mathit{nat} :: & \lambda x. & \textbf{if } x = \bot \textbf{ or } x = \mathit{true} \textbf{ then } \bot \\
 & & \textbf{else } \mathit{true}. \\
0 :: & 0. & \\
t :: & \mathit{true}. &
\end{array}
\tag{6.16}
$$

In this case, a sufficient well-founded ordering over $\mathcal{NAT}_{\bot,\mathit{true}}$ is given by $>_{\mathcal{H}} = \mathit{true} < \bot < 0 < 1 < 2 < \ldots$ An instance of the well-quasi general path ordering which proves that the conditional rewrite system (6.15) is terminating is given by:

$$
\begin{array}{rcl}
\phi_0 & = & \text{the precedence } F > - > s > 0 > \mathit{nat} > t \\
\phi_1 & = & \text{extract } P_{\{1\}} \text{ for } - \text{ and } \mathit{nat} \text{ with } \succ \text{ applied recursively} \\
\phi_2 & = & \text{extract } P_{\{1\}} \text{ for } F \\
 & & \text{and use the value preserving homomorphism } \mathcal{H} \text{ with } >_{\mathcal{H}} \ .
\end{array}
\tag{6.17}
$$

6.3 Insertion Sort over Integers

As a final example, consider the following conditional rewrite system which sorts a list of integers into ascending order via an insertion sort. It is a modification of the rewrite system presented

earlier for sorting natural numbers.

$$
\begin{array}{rrcll}
 & p(s(x)) & \rightarrow & x & \\
 & s(p(x)) & \rightarrow & x & \\
 & s(x)\ eq\ s(y) & \rightarrow & x\ eq\ y & \\
 & p(x)\ eq\ p(y) & \rightarrow & x\ eq\ y & \\
 & p(x)\ eq\ y & \rightarrow & x\ eq\ s(y) & \\
 & x\ eq\ p(y) & \rightarrow & s(x)\ eq\ y & \\
 & 0\ eq\ 0 & \rightarrow & t & \\
 & s(x)\ gt\ s(y) & \rightarrow & x\ gt\ y & \\
 & p(x)\ gt\ p(y) & \rightarrow & x\ gt\ y & \\
 & p(x)\ gt\ y & \rightarrow & x\ gt\ s(y) & \\
 & x\ gt\ p(y) & \rightarrow & s(x)\ gt\ y & \\
 & s^i(0)\ gt\ 0 & \rightarrow & t & i > 0 \\
 & sort(nil) & \rightarrow & nil & \\
 & sort(cons(x,y)) & \rightarrow & insert(x, sort(y)) & \\
 & insert(x, nil) & \rightarrow & cons(x, nil) & \\
x\ gt\ y \downarrow t & insert(x, cons(y,z)) & \rightarrow & cons(y, insert(x,z)) & \\
x\ eq\ y \downarrow t & insert(x, cons(y,z)) & \rightarrow & cons(x, cons(y,z)) & \\
y\ gt\ x \downarrow t & insert(x, cons(y,z)) & \rightarrow & cons(x, cons(y,z))\ . &
\end{array}
\tag{6.18}
$$

With an appropriate value-preserving interpretation, it can be shown that the conditions on the final three rules are mutually exclusive. Hence, the entire rewrite system is locally confluent. The rules for gt include one infinite schema. In this case, completion is easier to do with the rule schema than the other approaches mentioned earlier. Notice also, that if one is willing to forego confluence, the rule $p(x)\ gt\ y \rightarrow x\ gt\ s(x)$ and the corresponding rule for eq may be discarded. In that case, termination can be shown by the standard recursive path ordering alone with right-to-left lexical status for both gt and eq.

With the additional two rules for gt and eq, the recursive path ordering with status is insufficient for showing termination. This is due to the combination of $p(x)\ gt\ y \rightarrow x\ gt\ s(x)$ with the rule $x\ gt\ p(y) \rightarrow s(x)\ gt\ y$. For one rule, there is a decrease in the first argument, and for the other the decrease is in the second. Unfortunately, with either multiset or lexicographic

status, neither rule decreases. After a little consideration, it is apparent that both of these rules decrease the number of p's. The following components allow one to show termination with the well-quasi general path ordering:

$$\begin{array}{rcl}
\phi_0 & = & \text{the strictly monotonic homomorphism } \theta_0 \\
 & & \text{with } >_0 \text{ the usual greater-than for natural numbers} \\
\phi_1 & = & \text{the precedence } sort > insert > gt > eq > cons > nil > p > s > t > 0 \\
\phi_2 & = & \text{the extraction based on the outermost symbol } f \\
 & & \theta_2 = \begin{cases} P_{\{1\}} & f = sort, gt, eq \\ P_{\{2\}} & f = insert \\ \emptyset & otherwise\ , \end{cases} \\
 & & \text{with } \succ \text{ applied recursively.}
\end{array} \tag{6.19}$$

The monotonic homomorphism θ_0 counts p's in a term and is given by:

$$\begin{array}{rll}
cons :: & \lambda x, y. & x + y \\
insert :: & \lambda x, y. & x + y \\
gt :: & \lambda x, y. & x + y \\
ge :: & \lambda x, y. & x + y \\
sort :: & \lambda x. & x \\
s :: & \lambda x. & x \\
p :: & \lambda x. & x + 1 \\
0 :: & 0 & \\
t :: & 0 & \\
nil :: & 0\ . &
\end{array} \tag{6.20}$$

This homomorphism is strict in its arguments and maps to the natural numbers. As specified above, the corresponding well-founded component ordering is the usual greater-than for natural numbers. The proof of termination proceeds with no difficulties.

7 THE WELL-FOUNDED GPO

In [DH95] a slightly different version of the general path ordering was presented. It differs in the following ways:

- It requires that the homomorphism components are well-founded quasi-orderings but not necessarily well-quasi-orderings.
- The definition gives separate definitions for the strict and equivalence parts of the ordering.
- The ordering is more restrictive in what it allows to be comparable.
- There is an extra condition required for Theorem 20 (Specific Termination) which gives specific conditions on the components which guarantee termination of the rewrite system.

To avoid confusion in this thesis, the "general path ordering" presented in [DH95] is referred to as the well-founded general path ordering and and symbolically the ordering will be denoted with a dot as shown here by $\dot{\succsim}$ and $\dot{\approx}$. The component orderings associated with a well-founded general path ordering (which must themselves be well-founded) will also be denoted with a dot as shown by $\dot{\geq}$ and $\dot{=}$.

Definition 9 (Well-Founded General Path Ordering). Let $\phi_0 = \langle \theta_0, \geq_0 \rangle$, ..., $\phi_k = \langle \theta_k, \geq_k \rangle$ be component orders, where for multi set extraction θ_x component orders, $\geq_x$ is the well-founded general path ordering $\dot{\succsim}$ itself. The induced *well-founded general path ordering* $\dot{\succsim}$ is defined as follows:

$$s = f(s_1, \ldots, s_m) \succ g(t_1, \ldots, t_n) = t$$

if either of the two following cases hold:

(1) $s_i \dot{\succsim} t$ for some s_i, $i = 1, \ldots, m$, or

(2) $s \succ t_1, \ldots, t_n$ and $\Theta(s) >_{lex} \Theta(t)$, where $\Theta(s) = \langle \theta_0(s), \ldots, \theta_k(s) \rangle$, and $>_{lex}$ is the lexicographic combination of the component orderings $>_x$,

while

$$s = f(s_1, \ldots, s_m) \dot{\approx} g(t_1, \ldots, t_n) = t$$

in the well-founded general path ordering if

(3) $s \succ t_1, \ldots, t_n$, $t \succ s_1, \ldots, s_m$ and $\theta_0(s) \doteq_0 \theta_0(t), \ldots, \theta_k(s) \doteq_k \theta_k(t)$.

Note that $\succcurlyeq$ is the union of $\succ$ and $\dot{\approx}$, which are mutually recursive.

Lemmas 27, 28, 29 and 30 (below) guarantee that $\succ$ is the strict part of $\succcurlyeq$, while $\dot{\approx}$ is the equivalence part.

7.1 Quasi-Ordering Proofs

To show that $\succcurlyeq$ is indeed a well-founded quasi-ordering requires the following lemmata. The content of these proofs is fairly different from those associated with the well-quasi general path ordering. It is shown here that the definitions for $\dot{\approx}$ and $\succ$ are compatible and that their union is a quasi-ordering. In addition, it will be shown that $\succcurlyeq$ is well-founded.

Lemma 22 (Symmetry). *If $s \dot{\approx} t$ then $t \dot{\approx} s$.*

Proof. This is trivial, since $\doteq_x$ is reflexive for the component quasi-orders $\geq_x$. When $\doteq_x$ is the multiset extension of $\dot{\approx}$, induction on the combined size of the terms s and t is required. □

Lemma 23. *For the well-founded general path ordering, $s \succcurlyeq t$ implies $s \succ t|_p$ for each proper subterm $t|_p$ of t.*

Proof. Assume that the lemma holds for any pair of terms smaller in combined size than $\langle s, t \rangle$.

Suppose $s \succ t$ by Case (1) of the well-founded general path ordering. Then for some i, $s_i \succcurlyeq t$. By the induction hypothesis, however, $s_i \succ t|_p$. One may then apply Case (1) resulting in $s \succ t|_p$.

Suppose $s \succcurlyeq t$ by Case (2) or (3). Thus it is known that $s \succ t_1, \ldots, t_n$. Suppose that $t|_p$ is a subterm of some t_i. Then induction can be applied on the pair $\langle s, t_i \rangle$. □

The following two lemmata must be shown by simultaneous induction over the height of a term.

Lemma 24 (Subterm). *The well-founded general path ordering satisfies the strict subterm property* $f(\ldots, s_i, \ldots) \succ s_i$, *for all* i.

Proof. By inductive application of reflexivity (Lemma 25) to the subterm s_i one obtains $s_i \dot{\approx} s_i$, and Case (1) applies. □

Lemma 25 (Reflexivity). *The well-founded general path ordering* $\succsim$ *is reflexive.*

Proof. Assume that $\succsim$ is reflexive for all terms with height less than k. Consider a term $f(t_1, \ldots, t_n)$ of height k. By the strict subterm property (Lemma 24) for terms of height k, $f(t_1, \ldots, t_n)$ is strictly greater than each of its subterms. Therefore, the first and second conditions for equivalence are satisfied. Since each of the θ's is a function, $\theta_f(t_1, \ldots, t_n) \doteq_x \theta_f(t_1, \ldots, t_n)$ as long as each of the component orderings is reflexive. The only non-trivial case is the multiset ordering on immediate subterms. But by the induction hypothesis, $t_i \succsim t_i$ for every subterm t_i, and therefore the multiset ordering on immediate subterms is reflexive ($\{t_{j_1}, \ldots, t_{j_m}\} \dot{\approx}_{\mathcal{M}} \{t_{j_1}, \ldots, t_{j_m}\}$ for any $j_1, \ldots, j_m \in \{1, \ldots, n\}$). Consequently, the third condition is satisfied and $f(t_1, \ldots, t_n) \dot{\approx} f(t_1, \ldots, t_n)$. □

Lemma 26. *For the well-founded general path ordering,* $s \succsim t$ *implies* $u[s] \succ t$ *for each non-empty enclosing context* $u[\cdot]$ *of* s.

Proof. Consider the subterm $u|_p$ which contains s as an immediate subterm. By Case (1), $u|_p \succ t$. Repeated application of the preceding argument leads to $u[s] \succ t$. □

Lemma 27 (Transitivity). *For terms* s, t, *and* u *and well-founded general path ordering* $\succsim$ *:*

(i) $s \succ t \succ u$ *implies* $s \succ u$;

(ii) $s \dot{\approx} t \succ u$ *implies* $s \succ u$;

(iii) $s \succ t \dot{\approx} u$ *implies* $s \succ u$;

(iv) $s \dot{\approx} t \dot{\approx} u$ *implies* $s \dot{\approx} u$.

Proof. The proof proceeds by induction over the triple of terms $\langle s, t, u\rangle$ with respect to the sum of the heights of the three terms.

(i) Suppose that $s \succ t$ by Case (1) of the well-founded general path ordering, then $s_i \succcurlyeq t$ for some i. Now if $t \succ u$, one can apply induction on the triple $\langle s_i, t, u\rangle$ to get $s_i \succ u$. Thus, $s \succ u$ by Case (1) of the well-founded general path ordering.

Suppose that $s \succ t$ by Case (2) of the well-founded general path ordering, then $s \succ t_1, \ldots, t_n$ and $\Theta(s) \geq_{lex} \Theta(t)$. Now if $t \succ u$ by Case (1) of the well-founded general path ordering, then $t_j \succ u$ for some j. But induction may applied to the triple $\langle s, t_j, u\rangle$ to show $s \succ u$. If $t \succ u$ by Case (2) of the well-founded general path ordering, then $t \succ u_1, \ldots, u_m$ and $\Theta(t) \geq_{lex} \Theta(u)$. One may apply induction to each of the triples $\langle s, t, u_k\rangle$ to show that $s \succ u_k$ for each k. If each of the component orders is transitive then $\Theta(s) \geq_{lex} \Theta(u)$. When $\geq_x$ is a well-founded quasi-order there is no problem; when $\geq_x$ is a multiset ordering on immediate subterms, the induction hypothesis is needed.

(ii) We know that $s \succ t_1, \ldots, t_m$ and $\Theta(s) \doteq_{lex} \Theta(t)$.

Suppose that $t \succ u$ by Case (1) of the well-founded general path ordering, then $t_i \succcurlyeq u$ for some i. By induction on the triple $\langle s, t_i, u\rangle$, one obtains $s \succ u$.

Suppose that $t \succ u$ by Case (2) of the well-founded general path ordering, then $t \succ u_1, \ldots, t_m$ and $\Theta(t) \geq_{lex} \Theta(u)$. But, $s \succ u_k$ for each triple $\langle s, t, u_k\rangle$.

To show $s \succ u$, one merely needs to demonstrate the second condition of Case (2). But this holds for the quasi-orders and multiset orders by induction.

(iii) Essentially the same argument as for (i).

(iv) We know that $t \succ s_1, \ldots, s_l$, $\Theta(s) \doteq_{lex} \Theta(t)$, $t \succ u_1, \ldots, u_n$, and $\Theta(t) \doteq_{lex} \Theta(u)$. For each triple $\langle s, t, u_i\rangle$ one can apply (ii) to get $s \succ u_i$. For each triple $\langle u, t, s_j\rangle$ one can apply (ii) to get $u \succ s_j$. The lexicographic part holds for the quasi-orderings and, by induction, for the multiset orderings. Therefore, all three conditions of Case (3) hold and $s \dot{\approx} u$.

□

Lemma 28 (Irreflexivity). *For any s, $s \not\succ s$.*

Proof. Apply induction on the height of terms. Assume, on the contrary, that $s \succ s$ for some s.

Suppose that $s \succ s$ by Case (1) of the well-founded general path ordering, then $s_i \succcurlyeq s$ for some i. But by transitivity and the strict subterm property one obtains $s_i \succ s_i$. By induction $s_i \not\succ s_i$, which is a contradiction.

One cannot have $s \succ s$ by Case (2) of the well-founded general path ordering, since $\Theta(t) \doteq_{lex} \Theta(u)$, (using induction for the multiset components).

Therefore, neither case is applicable and $s \not\succ s$. □

Lemma 29. *If $s \dot{\approx} t$ then $t \not\succ s$.*

Proof. Were $t \succ s$, then by transitivity $s \succ s$ contradicting the previous lemma (Lemma 28). □

The converse follows from:

Lemma 30. *If $s \succ t$, then $t \not\succcurlyeq s$.*

Proof. Were $t \succcurlyeq s$, then by transitivity $s \succ s$, contradicting Lemma 28. □

Theorem 31. *The well-founded general path ordering is a quasi-ordering.*

Proof. By the previous lemmata $\succcurlyeq$ is reflexive and transitive. □

Theorem 32. *The well-founded general path ordering $\succcurlyeq$ is well-founded.*

Proof. To prove the well-foundedness of $\succ$, suppose the contrary and consider a minimal infinite descending sequence $t^1 \succ t^2 \succ \cdots$, minimal in the sense that from all proper subterms of each term in the sequence there are only finite descending sequences. (By the subterm property, any term in a descending sequence can be replaced by any proper subterm that initiates an infinite descending sequence. Thus, a minimal descending sequence can always be constructed from an arbitrary descending sequence.) Case (1) of the definition of $\succ$ cannot be the justification for any pair $t^j \succ t^{j+1}$, since then $t^{j-1} \succ t^j|_p \succ t^{j+2}$, for some proper subterm $t^j|_p$ of the jth term in the example, and the example would not be minimal. Therefore, every pair must use Case (2) and consequently $\Theta(t^j) \succcurlyeq_{lex} \Theta(t^{j+1})$. But a lexicographic combination of well-founded orderings (including $\succ$ on multisets of proper subterms which by assumption are well-founded), is well-founded, and the descending sequence cannot be infinite. □

7.2 A Comparison of the Different Versions of GPO

Theorem 33. *Given a well-founded general path ordering* $\gtrsim$ *and a well-quasi general path ordering* $\succsim$ *which use the same component orderings, if* $s \dot{\gtrsim} t$ *then* $s \succsim t$ *and also if* $s \succ t$ *then* $s \succ t$.

Proof. The proof proceeds via simultaneous induction on the sum of the height of the terms s and t.

In the base case, both s and t are terms with no subterms. If $s \dot{\gtrsim} t$, it must be because of case (3) of the well-founded general path ordering and thus $\Theta(s) \doteq \Theta(t)$. But any homomorphism component will give the same order on s and t and the multiset extraction components will match since there are no subterms, so $\Theta(s) \simeq \Theta(t)$. Case (2) of the well-quasi general path ordering applies in both directions, giving $s \succsim t$. Similarly, if $s \succ t$, then $s \succ t$.

Now consider the inductive case. If $s \dot{\gtrsim} t$ due to case (3) of the well-founded general path ordering, then $s \succ t_i$ for all i and $t \succ s_j$ for all j and $\Theta(s) \doteq \Theta(t)$. By induction $s \succ t_i$ for all i and $t \succ s_j$ for all j. Any homomorphism component will give the same order on s and t. By induction, the extraction components will match, so $\Theta(s) \simeq \Theta(t)$. Therefore, case (2) of the well-quasi general path ordering applies in both directions and $s \succsim t$.

If $s \succ t$ by case (1) of the well-founded general path ordering, there is some subterm $s|_p$ such that $s|_p \gtrsim t$. By induction $s|_p \succsim t$ and Case (1) of the well-quasi general path ordering applies. By Lemma 14, it is strict and $s \succ t$.

If $s \succ t$ by Case (2) of the well-founded general path ordering, then $s \gtrsim t_i$ for all i and $\Theta(s) > \Theta(t)$. By induction $s \succsim t_i$ for all i. Any homomorphism component will give the same order on s and t and, by induction, the extraction components will match, so $\Theta(s) > \Theta(t)$. Therefore, case (2) of the well-quasi general path ordering applies. Since the lexicographical comparison is strict, Lemma 16 applies and $s \succ t$. □

In other words, given well-quasi ordered homomorphism components, if two terms are ordered under the well-founded general path ordering, they will have the same ordering under the well-quasi general path ordering. If two terms are incomparable under the well-founded general path ordering, they may be either comparable or incomparable under the well-quasi general path ordering.

7.3 Termination of Rewrite Systems for the Well-Founded GPO

The argument for the Theorem 20 (Specific Termination) for the well-founded general path ordering is essentially similar to that for the well-quasi general path ordering with one important exception. For any non-strict monotonic homomorphism (including precedence) or value preserving homomorphism, one can not guarantee that $f(\ldots s \ldots) \gtrsim f(\ldots t \ldots)$ because when $\Theta(s) \doteq \Theta(t)$ it may not be the case that $t \succ s_i$ for all i.

The consequence of this is that the proof of Specific Termination in [DH95] does not apply to the well-founded general path ordering.

Consider the following simple rewrite system:

$$\begin{aligned} a &\rightarrow b \\ f(x) &\rightarrow g(x) \ . \end{aligned}$$

While it is clearly terminating, it does not satisfy the conditions for Theorem 19 (General Termination) with the following well-founded general path ordering with a single component order.

$$\phi_0 = \quad \text{the precedence } f > g, \text{ and } a > b$$

Consider the following terms in order:

- $[a$ versus $b]$ Trivially $a \succ b$ by Case (2).
- $[b$ versus $f(a)]$ Since $a \gtrsim b$, then $f(a) \succ b$ by Case (1).
- $[a$ versus $f(b)]$ Since $b \not\gtrsim a$, Case (1) is not applicable. In addition, $\phi_0(a) = a$ is incomparable to $f = \phi_0(f(b))$ so Cases (2) and (3) don't apply either. Therefore, a and $f(b)$ are incomparable.
- $[f(a)$ versus $f(b)]$ Since a and $f(b)$ are incomparable, Case (1) is not applicable. Case (2) is not applicable either since $\phi_0(f(a)) = f = \phi_0(f(b))$. Finally, Case (3) is not applicable since $f(b) \not\succ a$. Therefore, $f(a) \not\gtrsim f(b)$.

But Theorem 19 requires that if $s \rightarrow t$ and $s \succ t$ then $f(\ldots, s, \ldots) \gtrsim f(\ldots, t, \ldots)$. In our example, $a \rightarrow b$ and $a \succ b$, but $f(a) \not\gtrsim f(b)$.

To solve this problem one can add conditions on the component orderings for Theorem 20 when used with the well-founded general path ordering. Two possibilities, either of which will solve the problem are:

- (Subterms Constraint) Guarantee that there is always a component which must show a decrease.
- (Well-Quasi Constraint) Force the non-extraction components to be well-quasi orderings.

7.4 Addition of the Subterms Constraint

The problem with the proof of Theorem 20 for the well-founded general path ordering is that if there is a homomorphism component where $\theta_i(f(\ldots,s,\ldots)) \doteq \theta_i(f(\ldots,t,\ldots))$, then all the other components may also be $\doteq$ and Case (2) will not apply. Unfortunately, one can not guarantee that $f(\ldots,t,\ldots) \succ t$, and Case (3) need not apply either. One possibility is to require that there be at least one component ordering that will show a decrease for $f(\ldots,s,\ldots)$ with respect to $f(\ldots,t,\ldots)$, whenever there is a homomorphism that is not strict. One method of doing this is to add extraction components such that all subterms are extracted by some component. This guarantees that there is at least one component that shows a decrease and Case (2) of the well-founded general path ordering will apply.

Consider the previous example once more. One can add the following component to the ordering:

$$\phi_1 = \text{ the extraction based on the outermost symbol } q$$
$$\theta_1 = \left\{ P_{\{1,\ldots,n\}} \quad \text{where } q \text{ has } n \text{ subterms} \right.$$

Now Case (2) is applicable since $\phi_0(f(a)) = f = \phi_0(f(b))$ and $\phi_1(f(a)) = \{a\} \succ_{\mathcal{M}} \{b\} = \phi_1(f(b))$. Therefore, $f(a) \succ f(b)$.

7.4.1 Specific Orderings Covered

All of the previous specific orderings except for the Natural Path Ordering are covered by the well-founded general path ordering with the addition of the Subterms Constraint. Those orderings that use a precedence, e.g. the recursive path ordering, will now also be applicable to rewrite systems with an infinite signature.

In addition, the following three path orderings are covered. (They are not covered by the Well-Quasi General Path Ordering since they use component orderings that are well-founded, but not well-quasi-orderings.)

Semantic path ordering (Kamin and Lévy [KL80]) θ_0 is the identity homomorphism; $\geq_0$ is a well-founded ordering; $\phi_1, \ldots, \phi_n$ give a permutation of the subterms.

For this ordering, one must separately insure that $s \to t$ implies $s \geq_0 t$. Indeed any terminating system can be (uninterestingly) proven terminating in this way [KL80], by taking $\geq_0$ to be the reflexive-transitive closure of $\to$.

Extended Knuth-Bendix ordering (Dershowitz [Der82], Steinbach and Zehnter [SZ90]) ϕ_0 is a monotonic interpretation; ϕ_1 is a precedence; $\phi_2, \ldots, \phi_{n+1}$ give the subterms in order, permuted, or multisets of immediate subterms, depending on the function symbol.

For a system like

$$\begin{aligned} fsx &\to shdfx \\ f0 &\to 0 \\ d0 &\to 0 \\ dsx &\to ssdx \\ hssx &\to shx \ , \end{aligned} \tag{7.1}$$

a precedence ($f > h > d > s > 0$) ought to be considered first, before looking at subterms, as with a lexicographic path ordering.

The next special case is not a simplification ordering, but the conditions of Theorem 20 hold for it as well.

Value-preserving path ordering (Plaisted [Pla79], Kamin and Lévy [KL80]) θ is a value-preserving homomorphism and $\geq$ is a well-founded quasi-order; ϕ_0 is a precedence; θ_1 is θ applied to the first subterm and $\geq_1$ is $\geq$; θ_2 is θ applied to the second subterm and $\geq_2$ is $\geq$; and so forth.

As an example of the use of the value-preserving path ordering, consider System 1.2. The precedence is $fact >_0 \times >_0 + >_0 s$; θ_1 interprets everything naturally: $fact$ as factorial, s as

successor, p as predecessor, $\times$ as multiplication, $+$ as addition, and 0 as zero. The ordering $\geq_1$ is the well-founded greater-than relation on natural numbers. Let all constants be interpreted as natural numbers, making all terms non-negative. Each rule causes a strict decrease with respect to the general path ordering and the rewrite system terminates. This approach works for primitive-recursive functions in general.

7.5 Addition of the Well-Quasi-Order Constraint

The other option is to require that all of the homomorphisms be well-quasi ordered. In that case, one can appeal to Theorem 33 to show that the well-founded general path ordering is a sub-ordering of some well-quasi general path ordering. Therefore, this ordering with the results presented in this thesis can then be used to show termination.

This is often not too much of a burden. As was mentioned earlier, precedence is relatively benign. If there are a finite number of symbols in the signature of the rewrite system, then precedence will map to a finite set of values, and is well-quasi-ordered. This will apply for any finite rewrite system. Once again, if one considers our simple example, the component ϕ_0 is well-quasi-ordered for the symbols f, g, a, and b. Therefore, the rewrite system terminates for terms made from those symbols. It is well known that if one can show termination of a rewrite system for a given alphabet, it is also terminating for extensions of that alphabet that do not include any symbols in the rules [Der95]. Therefore the above rewrite system terminates in general.

Other commonly used well-founded orderings used to prove termination are total. In this case, as well, the ordering is a well-quasi ordering. An example of this is the standard ordering $\leq$ applied to the set of natural numbers.

7.6 Incrementality of the General Path Ordering

There is a practical advantage to using the well-founded ordering whenever possible. Since the definitions of $\succ$ and $\dot{\approx}$ do not involve any negative conditions, the following theorem can be shown.

Theorem 34 (Incrementality). *If a well-founded general path ordering $\succcurlyeq$ with a component ordering $\phi_i = \langle \theta, \geq \rangle$ proves termination of a set of rules $\mathcal{R}$, then the well-founded general path*

ordering $\succnsim'$ *which is the same as* $\succnsim$ *except for* $\phi_i' = \langle \theta, \geq' \rangle$, *where* $\geq'$ *is an extension of the ordering* $\geq$, *also proves termination of* $\mathcal{R}$.

Proof. For any termination proof that uses the ith component ordering, the same proof can be constructed, since the mapping is identical and orderings $\geq$ and $\geq'$ are the same for any pair of values $\theta(t_1)$ and $\theta(t_2)$ used to show termination. □

Incrementality is important when an ordering is sought to orient a set of equations. Thus, as a special case, with a precedence one can delay deciding whether $f > g$, or $f < g$, or $f \doteq g$ until necessary to establish the ordering of two terms, (as for the standard recursive path ordering). In general, one can successively refine the well-founded ordering of a homomorphism component.

Unfortunately, for the well-quasi general path ordering incrementality no longer holds. This is due to the definition of $s \succ t$ which is true if $s \succnsim t$ is true but $t \succnsim s$ is *not*. The addition of the negative condition causes incrementality to break down.

As an example, consider the terms $f(a)$ and $g(b)$ with the well-quasi general path ordering with a single precedence component ordering:

$$\phi_0 = \quad \text{the precedence } f \simeq g > b$$

(the symbol a is incomparable with all the others.)

Consider the following terms in order:

- $[a$ versus $b]$ Trivially, a is incomparable to b.
- $[f(a)$ versus $b]$ Since b has no subterms $\theta_0(f(a)) = f >_0 b = \theta_0(b)$ is sufficient for the application of Case (2) and $f(a) \succnsim b$. Application of Lemma 16 results in $f(a) \succ b$.
- $[g(b)$ versus $a]$ Since b is incomparable to a, Case (1) can not be applied. Since g is incomparable to b in θ_0, Case (2) can not be applied in either direction. Therefore, $g(b)$ is incomparable to a.
- $[f(a)$ versus $g(b)]$ Since $f(a) \succ b$ and $\theta(f(a)) = f \simeq_0 g = \theta(g(b))$, Case (2) applies in the forward direction giving $f(a) \succnsim g(b)$. In the other direction, however, Case (2) does not

apply since $g(b) \not\succ a$. Case (1) can not apply either or Lemma 8 is violated. Therefore, $f(a) \succ g(b)$.

But, if the precedence is extended to:

$$\phi_0 = \quad \text{the precedence } f \simeq g > b \simeq a$$

One finds that $f(a) \succ b$ and $g(b) \succ a$ via Case (1) and Lemma 14. Case (2) will now apply in both directions and now $f(a) \approx g(b)$ violating incrementality.

It is interesting to note that if the original precedence is used with the well-founded general path ordering then the comparison of terms proceeds as:

- $[a$ versus $b]$ Trivially, a is incomparable to b.
- $[f(a)$ versus $b]$ Since b has no subterms $\theta_0(f(a)) = f >_0 b = \theta_0(b)$ is sufficient for the application of Case (2) and $f(a) \succ b$.
- $[g(b)$ versus $a]$ Since b is incomparable to a, Case (1) can not be applied. Since g is incomparable to b in θ_0, Cases (2) and (3) can not be applied in either direction. Therefore, $g(b)$ is incomparable to a.
- $[f(a)$ versus $g(b)]$ Since $\theta(f(a)) = f \doteq_0 g = \theta(g(b))$ only Case (3) can be applied. But this is not possible since $g(b)$ is not comparable to a. Therefore, $f(a)$ is incomparable to $g(b)$.

As expected from Theorem 33, the problem arises with terms that are unordered by the well-founded general path ordering, but are ordered by the well-quasi general path ordering.

7.7 Reconciling Well-Quasi and Well-founded General Path Orderings

One question that will remain unanswered in this paper is whether one must require that the component orderings be well-quasi-orderings (i.e., whether Theorem 3 (Special Termination) in the [DH95] is in fact true, if not proven so.) While this chapter shows that for many common cases it will be true, there is no guarantee. If it can be shown, then the well-founded general path ordering can be used to the exclusion of the well-quasi general path ordering.

8 FORWARD CLOSURES

In this chapter a second approach to showing the termination of (potentially non-simple) rewrite systems is presented. The forward closures of a rewrite system are a possibly infinite set of selected derivations. In certain cases, the ability to show that each of the forward closures terminates is enough to show that the rewrite system in general terminates. Syntactic conditions are given under which the termination of the forward closures suffices. An important result is presented that relates the forward closures to the innermost derivations of a rewrite system. It is shown that the forward closures of a rewrite system terminate if, and only if, the rewrite system is innermost terminating.

Since the number of forward closures of a rewrite system may be infinite, it may be desirable to further restrict the the forward closures to just those derivations that satisfy some rewrite strategy. (A typical strategy is allow rewriting only at outermost redexes.) With these kinds of restrictions on the forward closures, syntactic conditions for showing general termination are given.

8.1 Introduction

Consider a recursive definition like

$$f(x) \;=\; \text{if } x > 0 \text{ then } f(f(x-1)) + 1 \text{ else } 0 \;.$$

By a straightforward use of structural induction, one can prove that the least fix point (over the natural numbers) is the always-defined identity function. This definition translates into the rewrite system:

$$\begin{aligned} fsx &\rightarrow sffpsx \\ f0 &\rightarrow 0 \\ psx &\rightarrow x \;. \end{aligned} \tag{8.1}$$

It would be nice to be able to mimic the proof for the recursive function definition in the rewriting context, but several issues arise:

1. In the functional case, one can show that call-by-value terminates, which implies that all fix point computation rules also terminate. It will be seen under what conditions the same holds for rewriting.

2. For rewriting in general, one must consider the possibility that the x to which the definition of $f(x)$ is applied is itself a term containing occurrences of the defined function f (or of mutually-recursive defined functions), something usually ignored in the (sufficiently complete) functional case.

3. One cannot use a syntactic simplification ordering like the simple path ordering [Pla78], since the first rule is embedding. In fact, termination must be combined with the semantics $(f(x) = x)$, as is done for the functional proof.

First a few definitions: A *non-overlapping* system is one where no left-hand side of a rule unifies with any non-variable subterm of the left-hand side of another rule or with a non-variable proper subterm of itself, with variables in the two rules renamed apart. A *left-linear* system has no repeated variables on the left-hand side of a rule. Similarly, a *right-linear* system has no repeated variables on the right-hand side of a rule. An *orthogonal* system is non-overlapping and left-linear. An *overlaying* system is one whose only overlaps are at the topmost position, that is, no left-hand side unifies with a non-variable proper subterm of any left-hand side.

As an example of an orthogonal system, consider:

$$\begin{aligned} fsx &\rightarrow sfpsx \\ f0 &\rightarrow 0 \\ psx &\rightarrow x\ . \end{aligned} \tag{8.2}$$

The general path ordering works with component orders ϕ_0 and ϕ_1, where ϕ_0 is a precedence with $f >_0 s, p$, and ϕ_1 is a natural interpretation with $f_\theta = \lambda x.x$, $p_\theta = \lambda x.x - 1$, $s_\theta = \lambda x.x + 1$, and $0_\theta = \lambda x.0$.

The following rewrite system is overlaying and locally confluent (see definition in the next section):

$$\begin{array}{rcl} x \times 0 & \to & 0 \\ x \times sy & \to & (x \times y) + x \\ x + 0 & \to & x \\ 0 + x & \to & x \\ x + sy & \to & s(x + y) \\ sx + y & \to & s(x + y) \ , \end{array} \tag{8.3}$$

8.2 Locally Confluent Overlaying Systems and Termination

A *locally confluent* system is one for which $u \to s, t$ implies $s, t \to^* v$, for some v, where $\to^*$ is the reflexive transitive closure of the rewrite relation.

Proposition 35 (Gramlich [Gra92]). *A locally confluent overlaying system is terminating if, and only if, innermost rewriting always leads to a normal form.*

An *innermost* derivation is one in which the redex chosen at every rewrite step contains no rewritable proper subterm. In particular, orthogonal systems are locally confluent and have no (non-trivial) overlays; the proposition for this case was shown by O'Donnell [O'D77]. Geupel [Geu89] showed that left-linearity is unnecessary, that is, a non-overlapping system is terminating if, and only if, innermost rewriting always leads to a normal form.

An alternate proof to the one in [Gra92] is presented. (See also Middeldorp [Mid94].) It is similar in style to Geupel's proof [Geu89] that forward closures suffice for showing termination of non-overlapping rewrite systems.

Proof. A *term* t is *terminating* (written as $t \in \mathcal{T}_f$) if all derivations from t are finite; t is *non-terminating* ($t \in \mathcal{T}_\infty$) if some derivation from t is infinite; and t is on the *frontier* ($t \in \mathcal{FR}$) if t is non-terminating, but every proper subterm of t is terminating. If a term has no frontier subterms, then it must be terminating. Conversely, if a term has a frontier subterm, it is non-terminating.

For a locally confluent rewrite system, any terminating term t has a unique normal form $\hat{t}$ by Newman's Lemma [New42]. The *inner normalization* function N for a locally confluent

rewrite system is defined as follows:

$$N(t) = \begin{cases} f(N(t_1), \ldots, N(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \in \mathcal{T}_\infty \\ \hat{t} & \text{if } t \in \mathcal{T}_f. \end{cases}$$

Clearly, $t \to^* N(t)$.

If the rewrite system is non-terminating, an infinite derivation can be constructed as follows: Let $t^1 = s^1$ be a frontier term. It initiates an infinite derivation of the form

$$t^1 = s^1 \to^*_{\text{below top}} s^{1'} \to_{\text{at top}} t^2 \to \cdots ,$$

where all the steps in $s^1 \to \ldots \to s^{1'}$ are below the top position and t^2 contains a frontier subterm s^2 at some position p_2. Continuing in this way, the infinite derivation

$$t^1 \to^+ t^2 \to^+ t^3 \to^+ \cdots$$

is found where $t^i = u^2[u^3[\cdots u^i[s^i]_{p_i} \cdots]_{p_3}]_{p_2}$, each s^i is a frontier subterm of u^i, and

$$s^i \to^*_{\text{below top}} s^{i'} \to_{\text{at top}} u^{i+1}[s^{i+1}]_{q_{i+1}} ,$$

where $p_{i+1} = p_i \cdot q_{i+1}$. (This is a constricting derivation á la Plaisted [Pla93a], making the proof a little simpler.)

Notice that each redex in the infinite derivation is either terminating (those below p_i in s^i) or on the frontier (at p_i in s^i). Let us consider these cases separately.

- The redex is a terminating subterm: Since each of the terms in $s^i \to^* s^{i'}$ is on an infinite path, the position of the frontier is unaffected and hence by local confluence $N(s^i) = N(s^{i'})$. Since both s^i and $s^{i'}$ are nonterminating, by the definition of N one has $N(t^i[s^i]) = N(t^i[s^{i'}])$.

- The redex is a frontier subterm: In this case $s^{i'} \to u^{i+1}[s^{i+1}]_{q_{i+1}}$ with some rule $g(c_1, \ldots, c_n) \to r$ and substitution σ. Since $s^{i'}$ is on the frontier, each of its subterms must be terminating and therefore each of the terms in the image of σ is terminating as well. Since the rewrite system is overlaying, each

of the contexts $c_1, \ldots, c_n$ is in normal form, so the rewrites below p_i are all within the terminating terms introduced by σ. In other words, $N(s^i) = g(c_1, \ldots, c_n)\widehat{\sigma}$, where $\widehat{\sigma}$ is σ with each of the terms in its image rewritten to normal form. By application of the same rule $N(s^i) \rightarrow r\widehat{\sigma}$.

Consider $u^{i+1}[s^{i+1}] = r\sigma$. Since the terms in the image of σ are terminating, by the definition of N one has $N(r\sigma) = N(r\widehat{\sigma})$. (By definition, it is known that $r\widehat{\sigma} \rightarrow^* N(r\widehat{\sigma})$.) Since both s^i and $u^{i+1}[s^{i+1}]$ are frontier terms (in t^i and t^{i+1}, respectively), one sees that $N(t^i[s^i]) \rightarrow^+ N(t^{i+1}[s^{i+1}])$.

Thus from the infinite derivation $t^1 \rightarrow^+ t^2 \rightarrow^+ t^3 \rightarrow^+ \cdots$ another infinite derivation $N(t^1) \rightarrow^+ N(t^2) \rightarrow^+ N(t^3) \rightarrow^+ \cdots$ can be constructed. Each of the rewrite steps corresponding to a frontier redex in the original derivation will be innermost after the application of N. The remaining steps are all under the position of the immediately preceding frontier step and are applied to terminating subterms. By local confluence, these rewrites can be rearraged to be innermost as well. Thus, from any infinite derivation some innermost infinite derivation can be constructed. □

Notice that given any non-terminating term v, the above construction can be used to obtain the derivation $v[t^1] \rightarrow^+ v[N(t^1)] \rightarrow^+ v[N(t^2)] \rightarrow^+ \cdots$ and so each term is terminating if and only if it is innermost terminating.

As an example of the use of Proposition 35, consider System 8.3. It must be shown that, under the assumption that variables are bound to normal forms, each rule leads to a normal form. Consider the second rule. If x and y are in normal form, then after applying the rule the innermost redex is the newly produced multiplication. But it can be shown that this will terminate since its second argument is smaller. Addition can be considered separately from multiplication, and it too terminates regardless of changes in the first summand. Therefore, every innermost derivation terminates, and hence the system terminates.

8.3 Introduction to Forward Closures

The question of when termination of ground constructor instances of left-hand sides suffices for establishing termination in all cases is now considered.

Definition 10. The *forward closures* of a given rewrite system are a set of derivations inductively defined as follows:

- Every rule $l \to r$ is a forward closure.
- If $c \to \cdots \to d$ is a forward closure and $l \to r$ is a rule such that $d = u[s]$ for nonvariable s and $s\mu = l\mu$ for most general unifier μ, then $c\mu \to \cdots \to d\mu[l\mu] \to d\mu[r\mu]$ is also a forward closure.

The idea, first suggested by Lankford and Musser [LM78], is to restrict application of rules to that part of a term created by previous rewrites. *Innermost* (*outermost*) forward closures are defined as those closures which are innermost (outermost) derivations. More generally, arbitrary redex choice strategies may be captured in an appropriate forward closure. A forward closure which has a right most term which initiates a non-terminating sequence of rewrites will be denoted as an *infinite forward closure*. (It corresponds to a infinite derivation in the limit.)

For example, the forward closures of System 8.2 are

$$\begin{array}{lcll} fs^n x & \to^+ & s^n fpsx & n > 0 \\ fs^n 0 & \to^+ & s^n 0 & n \geq 0 \\ fs^n x & \to^+ & s^n fx & n > 0 \\ psx & \to & x\ . & \end{array}$$

In fact, since there is only one possible redex in every forward closure, these are the innermost and outermost forward closures as well.

For an example where the innermost and outermost forward closures are not identical, consider the rewrite system:

$$\begin{array}{rcl} fsx & \to & sfpsfx \\ f0 & \to & 0 \\ psx & \to & x\ . \end{array} \tag{8.4}$$

The forward closure

$$fssx \to sfpsfsx \to sffsx \to sfpsfx$$

is outermost, but not innermost. The forward closure

$$fssx \to sfpsfsx \to sfpssfpsfx$$

is innermost, but not outermost.

Proposition 36 (Dershowitz [Der81]). *A right-linear rewrite system is terminating if, and only if, there are no infinite forward closures.*

In particular, forward closures suffice for string-rewriting systems.

Thus, for a system like

$$\begin{aligned} fsx &\to ssfpsx \\ f0 &\to 0 \\ psx &\to x\ , \end{aligned} \tag{8.5}$$

we can restrict our attention to forward closures. (This is not exactly a string rewriting system since the second rule applies only at the *end* of a string.) Since f's won't nest, termination can be shown by comparing the argument on the left, sx, with the one on the right, psx, using a natural semantic comparison.

Proposition 37 (Geupel [Geu89]). *A non-overlapping rewrite system is terminating if, and only if, there are no infinite forward closures.*

This improves the result in [Der81] for orthogonal systems. In general, though, a rewrite system need not terminate even if all its forward closures do [Der81].

Consider the following rewrite system for symbolic differentiation with respect to t:

$$
\begin{aligned}
D\ t &\to 1 \\
D\ z &\to 0 \qquad\qquad z \text{ does not contain } t \\
D\ (x+y) &\to D\ x\ +\ D\ y \\
D\ (x\cdot y) &\to y\cdot D\ x\ +\ x\cdot D\ y \\
D\ (x-y) &\to D\ x\ -\ D\ y \\
D\ (-x) &\to -D\ x \\
D\ (x/y) &\to D\ x/y\ -\ x\cdot D\ y/y^2 \\
D\ (\ln\ x) &\to D\ x/x \\
D\ (x^y) &\to y\cdot x^{y-1}\cdot D\ x\ +\ x^y\cdot(\ln x)\cdot D\ y\ .
\end{aligned}
\tag{8.6}
$$

It is orthogonal, so the above method applies. Since D's are not nested on the right, forward closures cannot have nested D's. Since the arguments to D on the left are always longer than those on the right, all forward closures must lead to terminating derivations. Hence, regardless of the rewriting strategy and initial term, rewriting terminates.

8.4 The Relation of Forward Closures to Innermost Termination

The following theorem establishes the connection between forward closures and innermost derivations. In particular, if one has some set of conditions for which innermost termination suffices to show termination, then forward closures will show termination for rewrite systems satisfying that set of conditions.

Theorem 38. *A rewrite system has an infinite innermost derivation if, and only if, it has an infinite innermost forward closure.*

Proof. Consider a term t which has an infinite innermost derivation. It must have a subterm $t|_p$ which has an infinite innermost derivation such that the top position is eventually rewritten:

$$t|_p = s^0 \to s^1 \to \cdots \to s^i \to_{\text{top}} s^{i+1} \to \cdots\ .$$

But for the top of s^i to be rewritten all of its immediate subterms must be in normal form. Consider this term. The first rewrite step from s^i must be at the top. Every rewrite afterwards must be applied in context created by previous rules since an application anywhere else would

be to a redex that is a proper subterm of s^i, which were all in normal form. If one considers $l \to r$, the rewrite rule applied in the first step from s^i, there must be some substitution μ such that $l\mu = s^i$. Each time the forward closure is extended, a substitution σ_j must be applied and the first term in the forward closure is given by $l\sigma_1\sigma_2 \cdots \sigma_j$.

If every substitution after a certain point in the derivation corresponds to a match, then one has found an infinite forward closure. For this not to be true, one of two possibilities had to have occurred at an infinite number of rewrite steps. The first possibility is that the substitution did not correspond to a match because of an equality constraint on variables. For example, if the right most term is $q(x, y)$ and the left hand side of the rule is $q(z, z)$ the substitution will be something like $\{x \mapsto z, y \mapsto z\}$. Notice though, that the number of variables remaining in the terms of the extended forward closure has been reduced. The second possibility is that the substitution wasn't a match because there was some symbol on the left hand side of the rule which wasn't in the rightmost term of the forward closure. For example, if the right most term is $q(x, y)$ and the left hand side of the rule is $q(a(w), z)$ the substitution will be something like $\{x \mapsto a(w), y \mapsto z\}$. In this case, after the substitution the number of symbols in the leftmost term of the forward closure has increased.

Consider the number of symbols in $l\mu = s^i$ minus those in $l\sigma_1\sigma_2 \cdots \sigma_j$ paired with the number of variables in $l\sigma_1\sigma_2 \cdots \sigma_j$. At each step corresponding to one of the two possibilities, one or the other of these is reduced and therefore, there can not not be an infinite number of substitutions that are not matches. Thus, the derivation from s^i is an instance of an infinite innermost forward closure. □

The set of forward closures can be restricted even more.

Theorem 39. *A rewrite system has an infinite innermost derivation if, and only if, it has an infinite leftmost/rightmost innermost forward closure.*

Proof. The proof is essentially the same. One just needs to show that one can reorder the derivation steps so that they are leftmost/rightmost as well as innermost. Clearly, this is possible and the proof proceeds as before. □

8.5 Overlay Rewrite Systems with Forward Closures

In this section, syntactic conditions will be presented which guarantee that the termination of forward closures suffices to show termination of the rewrite system.

Theorem 40. *A locally-confluent overlaying rewrite system is terminating if, and only if, it has no infinite leftmost/rightmost innermost forward closure.*

In particular, non-overlapping, and hence orthogonal, systems satisfy the prerequisites for application of this termination test; one need only prove termination of such innermost derivations.

Proof. From Proposition 35 if the rewrite system is non-terminating it will have an innermost non-terminating derivation. But by Theorem 39 this implies the existence of an infinite innermost forward closure. □

This method applies to most of the previous examples. Since one only needs to consider innermost derivations, it can be assumed that problematic expressions like psx on the right of System 1.2 rewrite immediately to x (and that the x is in normal form). Since only the forward closures need to be considered, it can be assumed that x contains no function symbols other than s and 0, without having to show that *fact* is sufficiently complete (which it would not be were the rule $fact(0) \to s0$ omitted). By "sufficiently complete", it is meant that every ground term containing the symbol *fact* and constructors reduces to a term containing only constructors.

For System 8.1, one can compare the multiset of right-hand side arguments $\{fpsx, psx\}$ of the recursive function symbols with that of left-hand side, $\{sx\}$. Semantics are necessary for this comparison. If we let $psx = x$ and $fx = x$ (just as would be done when using θ_0 with a natural path ordering), we have $\{sx\}$ greater (in the multiset ordering) than $\{x, x\}$. But one must ensure that the semantics are consistent with the rules (which is analogous to showing that $f(x) = x$ is a fix point of the definition). This can be done using standard rewriting techniques ("proof by consistency"; see Bachmair and Dershowitz [BD94]). Indeed, adding $fx \to x$ to System 8.1 yields a terminating confluent overlay system.

It is instructive to compare the above examples with the following nonterminating rewrite system:

$$\begin{aligned} fsx &\rightarrow ssffpsx \\ f0 &\rightarrow 0 \\ psx &\rightarrow x \ . \end{aligned} \tag{8.7}$$

It is the rewriting analogue of the recursively-defined function

$$f(x) \ = \ \text{if } x > 0 \text{ then } f(f(x-1)) + 2 \text{ else } 0 \ ,$$

which does not terminate for 2. Indeed, $f(x) = x$ would be inconsistent with the rules (allowing one to prove $s0 = ss0$).

Lemma 41. *If a left-linear rewrite system is constructor-based, then all of its forward closures begin with constructor-based instances of left-hand sides of rules.*

A term is *constructor-based* if all of its proper subterms have only free constructors and variables. A rewrite system is constructor-based if its left-hand sides are constructor-based, and a forward closure is constructor-based if its initial term is constructor-based.

Proof. Since forward closures are only extended via substitution, a trivial induction shows that every forward closure's initial term is an instance of the left-hand side of some rule.

Consider the inductive definition of forward closures. For the base case, each rule is a forward closure which, trivially, is constructor-based. Assume that $c[\vec{x}] \rightarrow \cdots \rightarrow d[\vec{x}]$ is a constructor-based forward closure. It is extended by applying the substitution σ, found by unifying the left-hand side of a rule, $f(k_1[\vec{y}], \ldots, k_n[\vec{y}])$, with some subterm of d. Suppose that the extension is not constructor based. This can only happen if the substitution, σ, maps some $x_i \in \vec{x}$ to a term with a function symbol in it. The term $f(c_1[\vec{x}], \ldots, c_n[\vec{x}])$ is unified with $f(k_1[\vec{y}], \ldots, k_n[\vec{y}])$. Since the rule itself is constructor-based, the only source of a function symbol is one of the contexts, c_i, from d. But these can only unify with variables, $\vec{y}$, from the rule. Since the rules are left-linear, each occurrence is distinct and therefore, the only mappings in σ which have function symbols are for variables in $\vec{y}$, not $\vec{x}$. This is a contradiction, and the extension is constructor based. □

As a counter-example illustrating the need for left-linearity, consider the rewrite system:

$$\begin{aligned} f(x,x) &\rightarrow f(ga,x) \\ gb &\rightarrow c\ . \end{aligned} \tag{8.8}$$

It is constructor-based, but the forward closure $f(ga,ga) \rightarrow \cdots \rightarrow f(ga,ga)$ is not.

A left-linear, locally confluent, constructor-based rewrite system is overlaying, and hence, by Theorem 40, is terminating if and only if its innermost forward closures are terminating. But by Proposition 41, all its forward closures begin with constructor-based instances of left-hand sides. Thus, termination proofs need not consider initial terms containing nested defined function symbols (even when the symbol is not completely defined). That makes proving termination of such systems no more difficult than proving termination of ordinary recursive functions: the instances of rule variables can be presumed to be in normal form and the context can be ignored.

8.6 Non-erasing Systems and Forward Closures

Let us now consider non-erasing rewrite systems. Recall that a system is *non-erasing* if any variable on the left-hand side of a rule is also on the right-hand side.

Proposition 42 (O'Donnell [O'D77]). *A non-erasing orthogonal system is terminating if, and only if, it is normalizing (every term has a normal form).*

Therefore, the first rule of System 8.2 (which has a self-embedding) may be immediately followed by an application of the last rule, effectively replacing the former with $fsx \rightarrow sfx$. Now termination can be shown with a standard recursive path ordering with precedence $f >_0 s$, demonstrating that the original system is normalizing, and, hence, terminating.

The previous proposition can be improved.

Lemma 43. *If a term has an infinite derivation in a non-erasing non-overlapping system, then all derivations from that term are infinite.*

Note that both non-overlapping string systems and non-erasing orthogonal rewrite system are special cases covered by this lemma.

Proof. The inner normalization function N is used. From the proof of Proposition 35, if t is a frontier term, then $N(t)$ is also non-terminating. As a consequence, for an arbitrary non-terminating term t, it must be that $N(t)$ is non-terminating as well.

Consider an arbitrary non-terminating term t and an arbitrary rewrite step applied to that term at redex s. The rewrite must occur in one of the following positions:

- The redex $s = l\sigma$ is a terminating term. But $t[l\sigma] \to t[r\sigma] \to^* N(t)$ by local confluence and since $N(t)$ is non-terminating, $t[r\sigma]$ is as well.

- The redex is a frontier term. It must be that there is exactly one rule, $l \to r$, applicable at that redex. From the proof of Proposition 35 presented in this thesis, one knows that the rule will still be applicable to $N(s)$. In addition, $N(s[r\sigma])$ is still non-terminating. Suppose that there was some other rule, $l' \to r'$, which was applicable, but led to a terminating term. This rule would also be applicable to $N(s)$. But since $N(s)$ is an instance of the right-hand sides of both rules they overlap, which is a contradiction. Therefore, $t[s[l\sigma]] \to t[s[r\sigma]] \to N(t[s[r\sigma]])$ and $N(t[s[r\sigma]])$ is non-terminating.

- The redex is non-terminating, but is not a frontier term. There must be some subterm $s|_p$ which is the frontier. Suppose that the rule, $l \to r$ has the top symbol of $s|_p$ as part of its context $c[\cdot]$. Consider applying N to the entire term. The subterms of the context $c[\cdot]$ are terminating, so they must be preserved; the top symbol of $c[\cdot]$ heads the subterm $r|_p$ and won't be rewritten, either. Since N maps terminating terms to their unique normal forms, repeated variables will observe the same rewrite and the applicability of the rule is unaffected by N. But there is some other rule, $l' \to r'$, which is applicable at the top of $N(s|_p)$. But that means there is an instance to which both rules apply and overlap. Therefore, the rule may only bind $s|_p$ by a variable. Since the system is non-erasing, the frontier term $s|_p$ must also be in the result of the rewrite, $t[r\sigma]$, which consequently must also be non-terminating.

Since there is no rule application which can lead to a term that is terminating, every derivation from a non-terminating term must be infinite. □

The following non-overlapping rewrite system shows that the non-erasing property is necessary:

$$\begin{aligned} gx &\rightarrow a \\ b &\rightarrow gb. \end{aligned} \tag{8.9}$$

Clearly, the term b has both infinite and terminating derivations.

To see that this result can not be extended to non-erasing, locally-confluent overlaying systems consider:

$$\begin{aligned} a &\rightarrow a \\ a &\rightarrow b. \end{aligned} \tag{8.10}$$

Unfortunately, the term a has both infinite and terminating derivations.

The following generalizes Proposition 42.

Theorem 44. *A non-erasing non-overlapping system is terminating if, and only if, it is normalizing.*

This is a corollary of Lemma 43. Gramlich [Gra94] gives an independent proof of this.

Theorem 45. *A non-erasing non-overlapping system is terminating if, and only if, no right-hand side of an arbitrary strategy basic forward closure initiates an infinite derivation.*

A *basic* forward closure $l\sigma \rightarrow r\sigma \rightarrow \cdots$ is one for which the substitution σ, used in the first step of the closure, is irreducible.

Proof. Suppose the system has an infinite derivation. Then by Theorem 40, there is a innermost forward closure leading to an infinite derivation. But the left-hand side of the infinite forward closure is a term which has an infinite derivation, and hence all derivations from it must be infinite as well (by Lemma 43). Furthermore, all derivations from it are instances of basic forward closures. Therefore, for an arbitrary strategy there is a corresponding infinite basic forward closure of the appropriate type. □

As an example, consider the following system:

$$\begin{aligned} fsx &\rightarrow psffx \\ f0 &\rightarrow 0 \\ psx &\rightarrow x\ . \end{aligned} \tag{8.11}$$

Its outermost forward closures are:

$$\begin{array}{rcll} fs^n x & \to^+ & f^{n-1}psffx & n > i \\ fs^n x & \to^+ & f^{n+1}x & n > i \\ fs^n 0 & \to^+ & f^m 0 & n \geq 0, n \geq m \\ psx & \to & x\ . & \end{array}$$

For a forward closure which is an instance of $fs^n x \to^+ f^{n-1}psffx$, one only needs to consider the extension with the rule $psx \to x$, since any other choice would not lead to an outermost forward closure. Verification of termination is easy now. Terms of the form $f^{n-1}psffx$ derive in one step $f^{n+1}x$ which is in normal form. Terms of the form $f^m 0$ derive 0 in m steps. Since no right-hand side admits a non-terminating rewrite sequence, the system is terminating.

System 7.1 can be shown terminating via similar reasoning (though the expressions for the forward closures are more complicated).

8.7 Example of Using Forward Closures to Prove Termination

Zantema's Problem [Zan92a] is to prove termination of the following one-rule string-rewriting system:

$$1100 \to 000111, \tag{8.12}$$

corresponding to the term-rewriting rule $1100x \to 000111x$. (Theorem 44 applies as well, since string rewriting systems are non-erasing and this rule is non-overlapping.)

First note that for any term of the form $\alpha 00\beta$, if $\alpha 00$ is a normal form then any term derived from $\alpha 00\beta$ must have the form $\alpha 00\gamma$. Consider the right-hand side of the rule. It has the above form with suffix $\beta = 111$. There are two ways to construct a new outermost forward closure from 111:

$$\alpha 0011100 \to \alpha 001000111 = \alpha' 00111$$

and

$$\alpha 00111100 \to \alpha 00\underline{11100}0111.$$

Since there is a redex (underlined) in the right hand side of the second forward closure, any outermost forward closure extending it must rewrite the redex:

$$\alpha 00111100 \to \alpha 000001110111 = \alpha' 001110111.$$

This gives us a new possibility $\beta = 1110111$, which can be used to construct a new outermost forward closure as:

$$\alpha 00111011100 \to \alpha 0011101000111 = \alpha' 00111$$

and

$$\alpha 001110111100 \to \alpha 00111011000111.$$

As before, one needs to reduce the right hand side for any outermost forward closure:

$$\begin{aligned} \alpha 001110111100 &\to \alpha 001\underline{11000}01110111 \\ &\to \alpha 0010001\underline{1100}1110111 \\ &\to \alpha 001000100011111110111 = \alpha' 001111110111 \ . \end{aligned}$$

The third possibility is $\beta = 1111110111$, which can be used to construct a new outermost forward closure as:

$$\alpha 00111111011100 \to \alpha 0011111101000111 = \alpha' 00111$$

and

$$\alpha 001111110111100 \to \alpha 00111111011000111 \ .$$

The second of these has a redex which must be rewritten:

$$\begin{aligned} \alpha 001111110111100 &\to \alpha 0011111\underline{1100}001110111 \\ &\to \alpha 0011\underline{11000}111001110111 \\ &\to \alpha 00\underline{11000}1110111001110111 \\ &\to \alpha 000001110111011\underline{1100}1110111 \\ &\to \alpha 0000011101110100011111110111 \\ &= \alpha' 001111110111 \ . \end{aligned}$$

For termination, it must be the case that no right-hand side of an outermost forward closure initiates a non-terminating derivation. Each of the right-hand sides of the form α00111, α001110111, and α001111110111 are already in normal form. Consider the right-hand side α00111011000111. It has only one possible derivation, leading to the normal form α001111110111. The right-hand side α00111111000011110111 is a little more complicated. The next term in the sequence is α00111100011100111110111, which has two possible rewrites. But notice that each of the succeeding terms in the outermost derivation preserve the inner rewrite. Therefore, they can be performed independently and α'001111110111 is the final form of all possible rewrites. None of the right-hand sides initiates an infinite rewrite, so the system is terminating.

Note that all derivations of a non-overlapping string-rewriting system have the same length. Hence, it has been shown (as Zantema conjectured) that $2n$ is an upper-bound on the length of any derivation from a string of size n (in worst case six steps are needed to decrease the size of the suffix β by three). Other solutions to this problem are due to Geser [Ges93] and Bittar [Bit93]. See also McNaughton [McN94] who considers termination of semi-Thue systems such as this example.

9 FORWARD CLOSURES AND COMPLETION

In the following chapter an application of forward closures for determining if a rewrite system terminates is examined. It is considered in the context of completion of an equational theory. One particular viewpoint of completion is as a search over rule orientations via an ordering. In this approach the ordering is replaced by a non-termination test.

9.1 Completion as Search

In the classical Knuth-Bendix completion process [KB70], one starts with a set of equations. Before beginning an ordering is chosen. Equations are oriented into rewrite rules via the ordering (typically one at a time). Critical pairs are computed for the rewrite rules, and any which are not joinable are added to the set of equations. Unfortunately, an equation may be found which can not be ordered, causing the method to fail. A modification of the above allows pairs of terms which can not be ordered to be kept as an equation and applied in either direction, provided that for the particular instance there is a decrease based on the ordering [HR86].

One can view the completion process as a search through orientations of the equations, where one is looking for a rewrite system which is

- terminating,
- locally confluent, and
- finite.

Note that completion can be used for computation, even if the resulting set of rules is not finite, as long as the completion process is fair (no equation is ignored forever).

Typically, however, one desires a finite rewrite system as the result of completion. Via the above process, one often is in the position of guessing an ordering and then starting over with a new ordering when an equation which can not be ordered is encountered, or it looks like the completion process is not going to generate a finite set of equations. One way to attempt to deal with this problem is to use a path ordering (typically the recursive path ordering [Der82] seen

in Section 5.1) which is only partially specified. When two terms are encountered which are incomparable, the completion process presents the user with extensions to the ordering (if they exist) which will orient the pair. If the user has some idea as to which direction an equation should have as a rewrite rule, then this may require less searching through orderings during completion. Unfortunately, there is still a search involved and controlling it are the orderings chosen.

In addition, simplification orderings which are probably among the easiest to understand and apply have a serious defect. They can not be used to show the termination of any rewrite system which has an "embedding". As an example, consider the single rewrite rule:

$$ff(x) \quad \rightarrow \quad fgf(x) \tag{9.1}$$

Since the left-hand side embeds in the right hand side there is no simplification ordering which can show that this rewrite system terminates.

The desired goal is to allow for the decoupling of the search process embedded in completion with determining an ordering for the rewrite system. *Partial completion* would proceed as a search over rule orientations without an ordering and, hopefully, a finite set of rules would be produced. This requires a heuristic method for grading a particular orientation of a set of equations. The result of partial completion would be a finite, locally confluent rewrite system. To show that the rewrite system was in fact confluent would require a separate proof of termination.

One obvious advantage to the search approach is that there are often equations which can only be oriented in one direction. Three basic kinds of equations which are easily oriented are:

1. Equations where a variable only occurs in one term. For example, the equation $f(x, a) = gb$ can only be oriented as $f(x, a) \rightarrow gb$.

2. Equations where one of the terms matches a proper subterm of the other term. For example, the equation $x + 0 = x$ can only be oriented as $x + 0 \rightarrow x$.

3. Equations where one of the terms has a constructor symbol at the top. For example, if s is a constructor, the equation $fsx = sfx$ can only be oriented as $fsx \rightarrow sfx$.

One potential disadvantage of the search approach is that the current orientation of equations may give a non-terminating rewrite system. Hence, one will need to guard against non-termination when the joinability of critical pairs is considered. For standard completion, this was not a problem since the rules always showed a decrease in the given ordering.

9.2 Previous Approaches for Avoiding Non-termination

There are two previous approaches to this problem. The first, by Plaisted [Pla86], proposes that during any derivation, a check for embeddings be employed. The other, by Purdom [Pur87], proposes a specific test for detecting non-termination of a rewrite system. Both of these approaches are examined in more detail in this section.

9.2.1 Plaisted's Approach

The approach introduced by Plaisted [Pla86] was to check for embeddings. If an embedding is found, then there may be a non-terminating derivation. On the other hand, if there is no embedding, then the rewrite system must be terminating as the following theorem by Kruskal shows.

Proposition 46. *Suppose t^1, t^2, ..., is an infinite sequence of ground terms over a finite set of function symbols. Then there exist an i and a j with $i < j$, such that t^i is homeomorphically embedded in t^j.*

Unfortunately, to use this as a proof of termination requires that one show that *all* derivations do not have an embedding. As employed by Plaisted, only derivations arising from critical pair computations are checked. An extension mentioned by Plaisted is to choose some set of terms V. For these terms, the embedding relation for each pair is pre-computed. As equations are turned into rewrite rules, the derivability relation between terms is maintained. If at some point derivability and embedding both relate two terms in the set V, then one would backtrack from this particular orientation of equations.

The rationale for not allowing a set of rule orientations which has an embedding is that if an embedding exists, there is no simplification ordering which will allow one to show termination. Plaisted considers this desirable since, in practice, non-simplification orderings are difficult to understand.

One potential problem with this approach is that even if a set of equations can be oriented and passes Plaisted's test, there still may not be any simplification ordering which will show termination. This is because only selected derivations have been tested and found free of embeddings. In fact, the rewrite system may be non-terminating, or require a non-simplification ordering in order to show termination.

This method's most serious defect when employed as a heuristic to guide search is that it may discard orientations of rules which are, in fact, terminating.

9.3 Purdom's Approach

Purdom proposed in [Pur87] that instead of checking for embeddings, one should employ a test for non-termination. The basic idea is that one is allowed to make arbitrary instantiations of the rules. Then one checks to see if there is a sequence of instantiated rules, each of which matches a subterm of the previous such rule in the sequence.

9.3.1 Single Derivation Step

The simplest test which Purdom gave was that one would check a sequence consisting of a single rule application.

Definition 11. [Single Step Purdom (SSP)] A set of rules, $\mathcal{R}$, has a non-terminating derivation if there exist substitutions σ and μ and a position p such that for some rule $l_i \rightarrow r_i$ in $\mathcal{R}$, the following equation is satisfied:

$$l_i\sigma\mu = (r_i|_p)\sigma. \tag{9.2}$$

In the above definition, the substitution σ gives one the instantiation of the rule and explains why one does not rename variables apart for l_i and $(r_i|_p)$. The substitution μ is the match. Solving problems of the form $t_i\sigma\mu = s_i\sigma$ is called *semi-unification*. The proof that this demonstrates non-termination is trivial.

One should note that even if the rewrite system consists of a single rule, there is no general method for showing termination, since one is able to simulate a Turing machine with a single rule [Dau89]. The non-decidability of the halting problem for a Turing machine then corresponds to the non-decidability of termination of rewrite systems. But even for a single left-linear, right-linear, non-erasing, non-overlapping, overlaying rule with unary function symbols, Purdom's

method may not discover non-termination. This is disappointing since it covers most of the common syntactic categorizations of rewrite systems. This is demonstrated by the following example:

$$fgx \quad \rightarrow ggffx \ . \tag{9.3}$$

This rewrite system has the non-terminating derivation $fggx \rightarrow ggffgx \rightarrow gg\underline{fgg}ffx \rightarrow \ldots$, but SSP will not discover this derivation. This is due to the fact that SSP requires that there be a "replication" after just a single application, whereas this derivation needs two steps.

Lemma 47. *If s and t are terms such that s is equal to a proper subterm of t, then there are no substitutions σ and μ such that the equation $t\sigma\mu = s\sigma$ is satisfied.*

Proof. Consider the size of a term t (denoted $|t|$) given by counting the symbols and variables. If s is equal to a proper subterm of t, then for any arbitrary instantiation σ, $|t\sigma| > |s\sigma|$. In addition, one knows that for any match μ $|t\sigma\mu| \geq |t\sigma|$. By transitivity, one has $|t\sigma\mu| > |s\sigma|$, and hence they can not be equal. □

Lemma 47 can be used to show that SSP fails to detect non-termination for System 9.3. Each of the possible positions shall be considered in turn. With p the top, $fgx\sigma\mu \neq ggffx\sigma$, since the outermost symbols f and g don't match. Similarly, each of the subterms until fx also fail. Now to solve $fgx\sigma\mu = fx\sigma$, one must be able to solve $gx\sigma\mu = x\sigma$. But by Lemma 47, this can not be done. The last possibility is $fgx\sigma\mu = x\sigma$, but Lemma 47 applies directly, and there is no solution. Hence, SSP can not demonstrate the non-termination of System 9.3.

9.3.2 Multiple Derivation Steps

To address problems such as System 9.3 and to allow considering rewrite sequences with more than a single rule, Purdom generalized SSP to allow for multiple derivation steps.

Definition 12. [Multiple Step Purdom (MSP)] A set of rules, $\mathcal{R}$, has a non-terminating derivation if there exist substitutions σ and μ and k positions p_i such that for rules $l_{n_i} \rightarrow r_{n_i}$ in $\mathcal{R}$

(n_i ranges over the rules), the following equations are satisfied:

$$\begin{aligned} l_{n_1}\sigma\mu &= (r_{n_k}|_{p_k})\sigma \\ l_{n_2}\sigma\mu &= (r_{n_1}|_{p_1})\sigma \\ &\vdots \\ l_{n_k}\sigma\mu &= (r_{n_{k-1}}|_{p_{k-1}})\sigma \ . \end{aligned} \tag{9.4}$$

In this case, the variables between each of the rules $l_{n_i} \rightarrow r_{n_i}$ are renamed. Thus, again the substitution σ corresponds to instantiating the selected rules, and μ gives the matches. Purdom's original method, SSP, is just the special case for $k = 1$.

There are a couple of problems with this method.

1. Each rewrite step in the non-terminating derivation must be at or below a previous step.

2. The presence of certain kinds of rules in a rewrite system have no effect in determining the outcome of MSP. In particular, collapsing rules can not contribute usefully to a non-terminating derivation.

3. MSP requires the use of semi-unification. While work has been done in [KMNS88] to improve the efficiency of the decidability of semi-unification to polynomial time, it is still desirable to avoid its use if possible.

As an example of the first problem, consider the following rewrite system:

$$\begin{aligned} f(0,1,a) &\rightarrow f(a,a,a) \\ a &\rightarrow 0 \\ a &\rightarrow 1 \ . \end{aligned} \tag{9.5}$$

Notice that the above rewrite system is variable free. This makes it easier to show that the non-terminating derivation $f(0,1,a) \rightarrow f(a,a,a) \rightarrow f(0,a,a) \rightarrow f(0,1,a)$ can not be discovered by MSP, since the substitutions can be ignored.

Suppose that either of the last two rules is the ith step in the derivation. Then 0 or 1 must be equal to the left-hand side of one of the rules. But this is not possible, so neither of the last two rules can be in the derivation. This leaves just the first rule. Now either $f(a,a,a)$ or a must match the left-hand side of the first rule. But, neither one does, so the first rule can not be in

the derivation as well and MSP fails. Notice that in the above non-terminating derivation, two steps were done in parallel and therefore MSP failed to detect non-termination.

It is not necessary that there be steps in parallel for MSP to fail. Consider the following rewrite system:

$$\begin{aligned} f(1,a) &\rightarrow f(a,a) \\ a &\rightarrow 1 \ , \end{aligned} \tag{9.6}$$

which is closely related to the previous one. It has the non-terminating derivation $f(1,a) \rightarrow f(a,a) \rightarrow f(1,a)$. By reasoning similar to that for System 9.5, one can show that MSP fails for this rewrite system as well. In this case, the second step in the derivation was above the first one.

9.3.3 Rule Removal

In this section, it will be shown that the presence of certain kinds of rules can not contribute to the success or failure of Purdom's method.

9.3.3.1 Semi-Unification

First, a lemma will be shown that allows one to essentially consider each of the rules separately with regard to the matching substitution μ when doing semi-unification.

Lemma 48. *Given σ and μ which solve*

$$\begin{aligned} l_{n_1}\sigma\mu &= (r_{n_k}|_{p_k})\sigma \\ l_{n_2}\sigma\mu &= (r_{n_1}|_{p_1})\sigma \\ &\vdots \\ l_{n_k}\sigma\mu &= (r_{n_{k-1}}|_{p_{k-1}})\sigma \ , \end{aligned} \tag{9.7}$$

there exist σ' and μ' which also solve the equations such that given rewrite rule $l_{n_i} \rightarrow r_{n_i}$ with renamed variables $\bar{x}_i$, σ' maps $\bar{x}_i$ to a terms over a unique set of variables $\bar{y}_i$, and μ' maps $\bar{y}_i$ to terms over $\bar{y}_{f(i)}$ where $f(i) = i - 1$ except for $f(1) = k$.

Proof. Proof by construction. Suppose that σ is the mapping $\bar{x}_1 \mapsto \bar{T}_1[\bar{y}], \ldots, \bar{x}_k \mapsto \bar{T}_k[\bar{y}]$, where $\bar{y}$ contains all the variables in the contexts $\bar{T}_1, \ldots, \bar{T}_k$, and the substitution μ is the mapping $\bar{y} \mapsto \bar{Q}[\bar{y}]$.

Construct σ' in the following manner. For each rule, construct the mapping $\bar{x}_i \mapsto \bar{T}_i[\bar{y}_i]$ where $\bar{y}_i$ is a renaming of $\bar{y}$ to unique variables. Construct μ' as the mappings $\bar{y}_i \mapsto \bar{Q}[\bar{y}_{f(i)}]$.

Consider the equation

$$l_{n_1}\sigma\mu \;=\; (r_{n_k}|_{p_k})\sigma \;. \tag{9.8}$$

The two terms $l_{n_1}\sigma$ and $l_{n_1}\sigma'$ differ only in that each variable in $\bar{y}$ is replaced by its renamed equivalent from $\bar{y}_i$. Lets call this context A and write $l_{n_1}\sigma = A[\bar{y}]$. Similarly, the two terms $(r_{n_k}|_{p_k})\sigma$ and $(r_{n_k}|_{p_k})\sigma'$ differ only in that each variable in $\bar{y}$ is replaced by its renamed equivalent from $\bar{y}_k$. Lets call this context B and write $(r_{n_k}|_{p_k})\sigma = B[\bar{y}]$.

By the equation, it is known that $A[\bar{y}\mu] = B[\bar{y}]$. Now consider $A[\bar{y}_1\mu] = B[\bar{y}_k]$. Notice that μ' maps the renamed variables to the terms $\bar{Q}[\bar{y}_{f(i)}]$, and therefore, the only possible difference between $A[\bar{y}_1\mu]$ and $B[\bar{y}_k]$ is in the variable positions. Again consider the original. By construction, if a variable $y_i \in \bar{y}$ appears, it came from replacing some variable $y_j \in \bar{y}$ by $\bar{Q}_j[\bar{y}] \in \bar{Q}$, which for μ' means that a variable $y'_i \in \bar{y}_k$ came from replacing some variable $y'_j \in \bar{y}_1$ by $\bar{Q}_j[\bar{y}_k] \in \bar{Q}$, but these will match the variables in $B[\bar{y}_k]$ and therefore, the variables all match as well and σ' and μ' are solutions to each of the equations. □

9.3.3.2 Example

As an example of the above manipulation of σ and μ, consider the following rewrite system:

$$\begin{aligned} fx &\rightarrow r(x,x) \\ r(y,z) &\rightarrow fs(h(y,y),z) \\ s(w,v) &\rightarrow fgw \end{aligned} \tag{9.9}$$

with the substitutions

$$\sigma = \left\{ \begin{aligned} x &\mapsto s(p,q) \\ y &\mapsto y \\ z &\mapsto s(m,n) \\ w &\mapsto p \\ v &\mapsto q \end{aligned} \right\} \tag{9.10}$$

and

$$\mu = \left\{ \begin{array}{rcl} y & \mapsto & s(p,q) \\ p & \mapsto & h(y,y) \\ q & \mapsto & s(m,n) \\ m & \mapsto & p \\ n & \mapsto & q \ . \end{array} \right\} \tag{9.11}$$

These satisfy the equations

$$\begin{array}{rcccl} f x \sigma \mu & = & f s(h(y,y), s(m,n)) & = & f g w \sigma \\ r(y,z)\sigma\mu & = & r(s(p,q), s(p,q)) & = & r(x,x)\sigma \\ s(w,v)\sigma\mu & = & s(h(y,y), s(m,n)) & = & s(h(y,y),z)\sigma \ . \end{array} \tag{9.12}$$

In this case, the set of variables $\bar{y}$ is $\{y, p, q, m, n\}$. The renamings (one for each rewrite rule) are $\bar{y}_1 = \{y', p', q', m', n'\}$, $\bar{y}_2 = \{y'', p'', q'', m'', n''\}$, and $\bar{y}_3 = \{y''', p''', q''', m''', n'''\}$.

The new substitutions are

$$\sigma = \left\{ \begin{array}{rcl} x & \mapsto & s(p', q') \\ y & \mapsto & y'' \\ z & \mapsto & s(m'', n'') \\ w & \mapsto & p''' \\ v & \mapsto & q''' \end{array} \right\} \tag{9.13}$$

and

$$\mu = \left\{ \begin{array}{rcl} y' & \mapsto & s(p''', q''') \\ p' & \mapsto & h(y''', y''') \\ q' & \mapsto & s(m''', n''') \\ m' & \mapsto & p''' \\ n' & \mapsto & q''' \\ y'' & \mapsto & s(p', q') \\ p'' & \mapsto & h(y', y') \\ q'' & \mapsto & s(m', n') \\ m'' & \mapsto & p' \\ n'' & \mapsto & q' \\ y''' & \mapsto & s(p'', q'') \\ p''' & \mapsto & h(y'', y'') \\ q''' & \mapsto & s(m'', n'') \\ m''' & \mapsto & p'' \\ n''' & \mapsto & q'' \ . \end{array} \right\} \tag{9.14}$$

It is easy to verify that these substitutions solve the equations.

9.3.3.3 The Main Theorem

Now it will be shown that certain kinds of subterms of the right hand sides of the rewrite rules need not be considered when attempting to solve the MSP equations.

Theorem 49. *Given a sequence of rules $n_1, n_2, \ldots, n_k$ (each of which satisfy the condition that all variables on the right-hand side are also on the left-hand side) and positions $p_{n_1}, p_{n_2}, \ldots, p_{n_k}$, which satisfy the equations*

$$\begin{array}{rcl} l_{n_1}\sigma\mu & = & (r_{n_k}|_{p_k})\sigma \\ l_{n_2}\sigma\mu & = & (r_{n_1}|_{p_1})\sigma \\ & \vdots & \\ l_{n_{k-1}}\sigma\mu & = & (r_{n_{k-2}}|_{p_{k-2}})\sigma \\ l_{n_k}\sigma\mu & = & (r_{n_{k-1}}|_{p_{k-1}})\sigma \ . \end{array} \tag{9.15}$$

if the subterm $(r_{n_k}|_{p_k})$ of the kth rule is identical to a proper subterm of its left-hand side l_{n_k}, then there is a proper subsequence of the rules excluding the kth rule for which MSP is satisfied.

Proof. First consider the case where k is one. By direct application of Lemma 47, it is known that there are no substitutions σ and μ which solve the equation.

If k is greater than one, then first by Lemma 48, σ and μ will be converted to σ' and μ', where the variable sets for each of the rules are denoted $\bar{x}_i$, the terms in σ and μ are over $\bar{y}$, and the terms in σ' and μ' are over $\bar{y}_i$.

It is given that there is some non-trivial q such that $l_{n_k}|_q = r_{n_k}|_{p_k}$. It must be that

$$\begin{aligned} l_{n_1}\sigma'\mu' &= (r_{n_k}|_{p_k})\sigma' \\ &= l_{n_k}|_q\sigma' \ . \end{aligned} \tag{9.16}$$

In addition, since

$$l_{n_k}\sigma'\mu' = (r_{n_{k-1}}|_{p_{k-1}})\sigma' \ , \tag{9.17}$$

one must be able to extract the subterms at position q on both sides of the equation giving

$$l_{n_k}|_q\sigma'\mu' = ((r_{n_{k-1}}|_{p_{k-1}})\sigma')|_q \ . \tag{9.18}$$

A little manipulation yields

$$l_{n_1}\sigma'\mu'\mu' = ((r_{n_{k-1}}|_{p_{k-1}})\sigma')|_q \ . \tag{9.19}$$

First, notice that since μ' only maps the variables in $\bar{y}_1$ to terms containing the variables in $\bar{y}_k$, and variables in $\bar{y}_k$ to terms containing the variables in $\bar{y}_{k-1}$, a new matching substitution μ'' can be constructed which drops mappings from $\bar{y}_k$, changes the mappings from $\bar{y}_1$ to be those from the substitution $\mu' \circ \mu'$, and leaves the others unchanged. This new matching μ'' with σ' will satisfy the $k-2$ equations remaining that don't involve rule n_k.

Now, if $r_{n_{k-1}}|_{p_{k-1}}$ contains position q, then one can find a new position $\tilde{p}_{k-1} = p_{k-1}.q$ where

$$l_{n_1}\sigma'\mu'' = (r_{n_{k-1}}|_{\tilde{p}_{k-1}})\sigma' \tag{9.20}$$

thus showing that a proper sequence of length $k-1$ had a solution.

If $r_{n_{k-1}}|_{p_{k-1}}$ does not contain position q, then one must remove the next rule as well. In this case, there must have been some variable $x \in \bar{x}_{k-1}$ such that

$$l_{n_1}\sigma'\mu'' = (x\sigma')|_{q'} \tag{9.21}$$

where q' is the position of some proper subterm.

Now consider the equation

$$l_{n_{k-1}}\sigma'\mu' = (r_{n_{k-2}}|_{p_{k-2}})\sigma' . \tag{9.22}$$

There must be a occurrence of the $(x\sigma')|'_q$ at some proper subterm q_2 of $l_{n_{k-1}}\sigma'\mu'$. Extracting that subterm, one arrives at

$$l_{n_1}\sigma'\mu''\mu' = ((r_{n_{k-2}}|_{p_{k-2}})\sigma')|_{q_2} . \tag{9.23}$$

As before, one can construct a new matching substitution. One is also faced, once again, with the question of whether q_2 is in $r_{n_{k-2}}|_{p_{k-2}}$ or not. If not, one continues the process of removing rules. Eventually, one must either find a proper subterm of the right hand side of a rule which works, or one is left with a single rule which must satisfy the equation

$$l_{n_1}\sigma'\mu^k = (x'\sigma')|_{q''} \tag{9.24}$$

where $x' \in \bar{x}_{n_1}$, μ^k is the substitution μ' applied k times, and q'' is some non-trivial position. But this is clearly unsatisfiable, since $|l_{n_1}\sigma'| > |x'\sigma'|$, and therefore, the process must have halted at one of the previous rule deletions leaving a proper subsequence. □

Corollary 50. *A rewrite system $\mathcal{R}$ (each rule satisfies the constraint that any variable on the right-hand side is also on the 1eft-hand side) with a rewrite rule $l_i \to r_i$ where the right-hand side is equal to a proper subterm of the left-hand side can be shown non-terminating by MSP if and only if the rewrite system $\mathcal{R} - \{l_i \to r_i\}$ can be shown non-terminating by MSP.*

Proof. Since the right-hand side matches the left-hand side, one knows that this will hold true for any instantiation of the rule with any subterm of the left-hand side. By Theorem 49, if the rule is part of a non-terminating sequence for MSP, it can be removed and there is still

a non-terminating sequence. Hence, there is a non-terminating sequence for MSP with the smaller rewrite system. □

One particular class of rules which is covered by Corollary 50 are collapsing rules, e.g. $hgx \to x$. So if a rewrite system has a collapsing rule, it may be removed when testing for non-termination with MSP. Other simple examples of rules which may be removed are $fgx \to gx$ and $ga \to a$.

9.3.3.4 Example

As an example of rule removal, consider the following rewrite system:

$$\begin{aligned} fx &\to hhfx \\ hy &\to ghy \\ hhfgz &\to fgz\ . \end{aligned} \tag{9.25}$$

Substitutions which allow one to show non-termination are

$$\sigma' = \left\{ \begin{array}{lcl} x & \mapsto & gx' \\ y & \mapsto & hfgy'' \\ z & \mapsto & z''' \end{array} \right\} \tag{9.26}$$

and

$$\mu' = \left\{ \begin{array}{lcl} x' & \mapsto & z''' \\ y' & \mapsto & x''' \\ z' & \mapsto & y''' \\ x'' & \mapsto & z' \\ y'' & \mapsto & x' \\ z'' & \mapsto & y' \\ x''' & \mapsto & z'' \\ y''' & \mapsto & x'' \\ z''' & \mapsto & y'' \end{array} \right\} \tag{9.27}$$

which satisfy the equations

$$\begin{aligned} fx\sigma'\mu' &= fgz''' = fgz\sigma' \\ hy\sigma'\mu' &= hhfgx' = hhfx\sigma' \\ hhfgz\sigma'\mu' &= hhfgy'' = hy\sigma' \ . \end{aligned} \tag{9.28}$$

By MSP one knows that there is an non-terminating derivation. Notice that the subterm hy of ghy is being used for the right hand side of the third equation.

According to Corollary 50, since fgz is identical to a proper subterm of $hhfgz$, one should be able to remove the third rule. There must be a subterm of $hhfgy''$ (from the right-hand side of the second rule) which is identical to $fgz'''\mu' = fgy''$. There is. Unfortunately, it does not correspond to any position of ghy, but is in the substitution for y.

This leaves us with the new set of equations

$$\begin{aligned} fx\sigma'\mu'\mu' &= fgy'' = (y\sigma')|_1 \\ hy\sigma'\mu' &= hhfgx' = hhfx\sigma' \ . \end{aligned} \tag{9.29}$$

Now one must remove the second rule as well. There must be an occurrence of the term $fx\sigma'\mu'\mu'\mu' = fgx'$ in $hy\sigma'\mu' = hhfgx'$ and therefore, it is a subterm of $hhfx\sigma'$ as well. This time, however, the position of fgx' is in the context of the right-hand side of the first rule. This leaves us with the single equation

$$fx\sigma'\mu'\mu'\mu' = fgx' = fx\sigma' \ . \tag{9.30}$$

The final substitutions (ignoring irrelevant mappings) are $\sigma = \{x \mapsto gx'\}$ and $\mu = \{x' \mapsto x'\}$. So the third rule was successfully removed from the non-terminating derivation. While this is not the most general substitution which will show non-termination, it does show that a solution exists.

If one were using MSP to detect non-termination with the rules in rewrite system 9.25, one could ignore the third rule entirely. For the first and second rules the only subterms that Theorem 49 applies to are the variable subterms x and y. One can not ignore fx for the first rule, for example, because it is not a proper subterm of the left hand side.

9.4 Using Ground Approximation to Detect Non-Termination

Before giving a test for non-termination using forward closures, another approach is briefly considered. One tempting idea for determining if a rewrite system is non-terminating is to construct a series of ground rewrite systems, each of which is better approximation to the original rewrite system.

Definition 13. Given a rewrite system $\mathcal{R}$, the *ith finite ground approximation* $\mathcal{R}^i$ of $\mathcal{R}$ is given by applying to each rule all substitutions mapping to ground terms of height i or less.

For example, given the following rewrite system:

$$\begin{aligned} a &\rightarrow b \\ f(x,y) &\rightarrow g(x), \end{aligned} \tag{9.31}$$

the initial approximation consists of all the ground rules in the rewrite system. In this case:

$$a \rightarrow b\ . \tag{9.32}$$

The next approximation is the union of the previous approximation with ground instances of the rules where the variables are replaced by ground terms of height one. In this case:

$$\begin{aligned} a &\rightarrow b \\ f(a,a) &\rightarrow g(a) \\ f(a,b) &\rightarrow g(a) \\ f(b,a) &\rightarrow g(b) \\ f(b,b) &\rightarrow g(b)\ . \end{aligned} \tag{9.33}$$

The next approximation is:

$$\begin{aligned} a &\rightarrow b \\ f(a,a) &\rightarrow g(a) \\ f(a,b) &\rightarrow g(a) \\ f(b,a) &\rightarrow g(b) \\ f(b,b) &\rightarrow g(b) \\ f(a,g(a)) &\rightarrow g(a) \\ f(a,g(b)) &\rightarrow g(a) \\ f(a,f(a,a)) &\rightarrow g(a) \\ f(g(a),f(a,a)) &\rightarrow g(g(a)) \\ &\vdots \end{aligned} \tag{9.34}$$

Huet and Lankford showed that termination is decidable for ground rewrite systems [HL78]. More recently, it has been shown that this problem has a polynomial time algorithm [GNP+93]. Alternately, to show decidability, one can employ forward closures [Der81]. Each of the forward closures must start from the left-hand side of one of the rules. Consider the forward closures resulting from some left-hand side l_i. If that left-hand side is ever replicated, the rewrite system is non-terminating. As a derivation step is taken, label the position of the redex with the rule applied. Any labels below the position of the redex are erased. If a redex is ever found for a rule which is below a position with a label for that rule, then the rewrite system is non-terminating. Such a labeling can not be continued forever, however, so the rewrite system either terminates or an instance of non-termination is discovered. As a consequence, the termination properties of each of the finite approximations is decidable. Unfortunately, as the following example shows, there are non-terminating rewrite systems for which all finite approximations terminate.

$$f(x) \rightarrow f(g(x)) \; . \tag{9.35}$$

The first approximation is empty. The second approximation is

$$f(a) \rightarrow f(g(a)) \; . \tag{9.36}$$

Notice that this terminates. The next approximation is given by:

$$\begin{aligned} f(a) &\rightarrow f(g(a)) \\ f(f(a)) &\rightarrow f(g(f(a)) \\ f(f(a)) &\rightarrow f(g(g(a)) \ . \end{aligned} \tag{9.37}$$

Once again, this terminates. If a term is viewed as a string of symbols, it must be some f's or g's with a single a on the right. The second and third rules can only be applied at the end of a term and in both cases do not leave something which can be rewritten again by those rules. Only the first rule could be applied, but it also can only be applied once.

Notice that for this rewrite system the infinite derivations are all instances of:

$$f(x) \rightarrow f(g(x)) \rightarrow f(g(g(x))) \ldots \tag{9.38}$$

In every step of the infinite derivation, the rule is applied at the top of the term and the term grows in height by one after each rewrite step. Therefore, the infinite derivation requires arbitrarily large ground instances on the right hand sides of the rules. Hence, any finite approximation will terminate.

But not all non-terminating rewrite systems pose this difficulty. In fact, if the rewrite system has a cycle, then some finite approximation will uncover a non-terminating derivation.

Theorem 51. *If $\mathcal{R}$ is a rewrite system which cycles, then for some n, the finite ground approximations $\mathcal{R}^i$ with i greater than n are non-terminating.*

Proof. Consider a cycle of $\mathcal{R}$ with m terms t^1, t^2, ... , t^m. If t^1 has variables, then construct an arbitrary ground substitution σ_g. Application of the substitution to the terms in the cycle does not prevent application of the rules. Consider each of the terms in the ground cycle $t^1\sigma_g$, $t^2\sigma_g$, ..., $t^m\sigma_g$. One of them must be of some maximum height given by n. Now since each of the terms is of height less than or equal to n, each of the redexes must be in the nth approximation to $\mathcal{R}$. □

The above suggests a hierarchy of non-termination.

1. There is a non-terminating derivation with a cycle.

2. There is a non-terminating derivation which replicates a ground term.

3. There is a non-terminating derivation which replicates a term with variables.

4. There is a non-terminating derivation which has a homeomorphic embedding.

Finite approximations are sufficient for the first two categories only. Unfortunately, the method of detecting non-termination with forward closures given in the next section does not fit neatly into this hierarchy. There are rewrite systems with cycles it will not work with, but it will work for System 9.35.

9.5 Using Forward Closures to Detect Non-termination

In this section, the use of forward closures to detect non-terminating rewrite sequences is examined.

Definition 14. [Forward Closure Test (FCT)] A rewrite system is non-terminating if

1. the initial term of a forward closure matches a subterm of the last term in a forward closure,

2. there is a rule which has a variable on the right-hand side which is not on the left-hand side, or

3. there is a rule of the form $x \rightarrow x$.

The proof that this finds a non-terminating derivation is trivial.

9.5.1 Comparison to Purdom

There are definite similarities between Purdom's method and using forward closures. Notice that by Lemma 49, in MSP one can not choose the subterm of the right-hand side of a rule to be just a variable. As a consequence, MSP will only apply rules in context created by a previous rule application (as is the case for forward closures). The difference, however, is that MSP requires the rule to be applied in the context created by just the previous rule, whereas forward closures can use context created by any previous rule.

Theorem 52. *FCT can show non-termination of any rewrite system $\mathcal{R}$ for which MSP can show non-termination.*

Proof. If there is a rule with a variable on the right-hand side which isn't on the left-hand side, then FCT will find it. Therefore, one only needs to consider rewrite systems where none of the rules have a variable on the right-hand side that isn't on the left. Given a sequence of such rules for which MSP can show non-termination, by Theorem 49 one can find a subsequence (of some length k) which also show non-termination which does not use any variable subterms on the right-hand sides (the trivial case of non-termination from the rule $x \rightarrow x$ is found by explicit checking of FCT and can be ignored). There will be some forward closure which corresponds to the subsequence of the rules. One final application of the first rule in the sequence gives one a forward closure of length $k+2$ where the initial term is replicated in the $k+1$ term. Since the $k-1$ remaining rules in the sequence can be applied without changing the initial term, the resulting final term will have a subterm which the initial term matches and the FCT detects a non-terminating derivation. □

9.5.2 Advantages of Using Forward Closures

One advantage of using forward closures is that with partial information one can get some information about the innermost termination properties of the rewrite system.

Lemma 53. *If all of the forward closures of a rewrite system $\mathcal{R}$ whose initial terms are of size less than or equal to k terminate, then a term t of size less than or equal to k whose proper subterms are in normal form is innermost terminating.*

Proof. Suppose, on the contrary, that there is an infinite innermost derivation from some such t. Since the subterms are in normal formal form, the first rule must be applied at the top and subsequent rules must be applied in the created context. Therefore, the infinite innermost derivation must be an instance of a forward closure, but all forward closures of that size terminate, hence the term must be terminating too. □

Typically, however the set of forward closures of a rewrite system is infinite. As a consequence, one can only check a finite number of the forward closures, and innermost termination of the rewrite system can only rarely be guaranteed.

When doing partial completion with FCT, one wants to be able to use the rewrite system while attempting to join critical pairs. Avoiding non-termination is not a problem in regular completion since the set of rules is always terminating. But in partial completion this is no longer guaranteed. But notice that as one generates forward closures, the size of the initial term never decreases. This means that the forward closures can be enumerated in increasing size and partial information about the innermost termination properties of the rewrite system can be extracted. In particular, knowing that the system is innermost terminating for terms (whose subterms are in normal form) up to a given size can relieve one of the need to guard against non-termination in some cases.

In addition, if one finds a final rewrite system, one can use the results from the forward closures to safely compute for terms of limited size.

The following counter-example shows that one can not conclude that all terms less than a given size k are innermost terminating even if the forward closures starting from terms of size less than k terminate. Consider the rewrite system:

$$\begin{aligned} hx &\rightarrow qx \\ qfx &\rightarrow hrx \\ rfx &\rightarrow fgx \\ ga &\rightarrow fa \\ b &\rightarrow ffa\ . \end{aligned} \tag{9.39}$$

It has the innermost derivation $hb \rightarrow hffa \rightarrow qffa \rightarrow hrfa \rightarrow hfga \rightarrow hffa \rightarrow \ldots$, which is non-terminating. Yet all of the forward closures from terms of size two or less terminate. It is not until initial terms of size four are encountered, that a non-terminating forward closure is generated. (Since system 9.39 is right-linear, forward closures are guaranteed to be terminating if and only if the rewrite system is terminating by Proposition 36.)

9.5.3 Heuristic Value

When one extends a forward closure, one of three cases can arise.

1. The initial term of the forward closure remains the same (up to a renaming of the variables).

2. The initial term of the forward closure remains the same except that one or more different variables are renamed to the same variable.

3. The initial term of the forward closure is extended in size.

Given the following rewrite system:

$$\begin{aligned} f(x,y) &\rightarrow g(x,y) \\ g(x,y) &\rightarrow a \\ g(x,x) &\rightarrow b \\ g(a,y) &\rightarrow c\ . \end{aligned} \tag{9.40}$$

The forward closure $f(x,y) \rightarrow g(x,y)$ can be extended in each of the three different ways.

1. With the rule $g(x,y) \rightarrow a$, one gets the new forward closure $f(x,y) \rightarrow g(x,y) \rightarrow a$.

2. With the rule $g(x,x) \rightarrow b$, one gets the new forward closure $f(x,x) \rightarrow g(x,x) \rightarrow b$.

3. With the rule $g(a,y) \rightarrow c$, one gets the new forward closure $f(a,y) \rightarrow g(a,y) \rightarrow c$.

In particular, if the right hand side of a forward closure leads to non-termination, there must be extensions of the first and second kinds only (each of the unifications is also a match). In addition, there can only be a finite number of applications of the second kind.

The approaches (heuristics) given by Plaisted and Purdom are all or nothing. Either the set of rules passes the test or it doesn't. The forward closures method (FCT) offers a potential way to grade different sets of rule orientations by comparing of the length of forward closures with the size of the initial starting term. In general, rewrite systems with shorter derivations would be preferred. For example, a rewrite system whose derivations were linear with respect to the size of a term would be preferred to one which was quadratic.

9.5.4 Computational Issues

One potential problem with computing the forward closures of a rewrite system is that it can be computationally intensive. It is not difficult to create rewrite systems that will generate an exponential number of forward closures with respect to the derivation length. For example,

consider the following rewrite system:

$$fa \rightarrow h(fa, fa) \tag{9.41}$$

Its forward closures of length i all end in binary trees with spines consisting of h's along with $i+1$ a's as leaves. It is well known that the number of such trees, T_i, is

$$T_i = \frac{1}{i+1}\begin{pmatrix} 2i \\ i \end{pmatrix} = \frac{4^i}{\sqrt{\pi i^3}}(1 + O(1/i))$$

which is clearly exponential. This just represents a lower bound on the number of forward closures. Each of the two following derivations is a forward closure of length three:

$$\begin{aligned} fa &\rightarrow h(fa, fa) \rightarrow h(h(fa, fa), fa)) \rightarrow h(h(fa, fa), h(fa, fa)) \\ fa &\rightarrow h(fa, fa) \rightarrow h(fa, h(fa, fa)) \rightarrow h(h(fa, fa), h(fa, fa)) \end{aligned}$$

But, they both end in the same term.

Analyzing the above system, one finds that the total number of derivations is given by the recurrence relation

$$D_i = iD_{i-1}$$

with initial condition $D_1 = 1$, which has the

$$D_i = i! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right) \ .$$

Clearly, if one is not careful about discarding alternate derivations which end in the same term, one can do worse than exponential.

Let's make a very rough estimate of the number of derivations. Suppose that there is a rule $l_i \rightarrow r_i$, and that there are L different pairs of positions p and rules $l_j \rightarrow r_j$ such that $l_i|_p$ unifies with l_j. Each of these represents, more or less, the loss of a position at which a unifier can be found that will extend the forward closure. Similarly, one can compute R for the right hand side of rule i, which represents a gain in positions. If Δ is the average difference in $R - L$ for all the rules, one might expect that the number of derivations of length i is approximately $N\Delta^i i!$, where N is the number of rules in the rewrite system.

Notice that this rough analysis did not take into account the possibility that the number of ways in which a forward closure can be extended

1. increases due to replication of a subterm because of repeated variables on the right-hand side of a rule,

2. increases due to the substitution increasing the size of the forward closure because of repeated variables on the left-hand side of a rule (or the last term of the forward closure), or

3. decreases due to the fact that the application of the rule at a position p affects a possible extension of the forward closure at a position above p.

9.6 Summary

The application of forward closures provides a method for detecting non-termination of rewrite systems. In comparison with Purdom's method it does not require the use of semi-unification, while being strictly more powerful in the sense that it will detect non-termination in all cases that Purdom's method will. Another advantage is the ability to extract partial information about the termination properties of terms of limited size. A disadvantage is the need to keep a potentially large set of terms representing forward closures of the rewrite system. It would be expected that any implementation of forward closures must have a means of avoiding multiple (forward closures) derivations ending in the same term.

10 RESTRICTING REWRITING TO INNERMOST DERIVATIONS

Theorem 38 guarantees that the termination of (innermost) forward closures, *is sufficient* to guarantee innermost termination of a rewrite system. This can be exploited in a number of cases. For example, many programming languages are applicative, and hence innermost derivations may be all that one is interested in when proving termination.

Another advantage is seen for showing the termination of modular rewrite systems. Toyama, Klop and Barendregt [TKB89] showed that termination is a modular property of left-linear confluent terminating rewrite systems. Unfortunately, confluence and termination alone are not sufficient to show termination of the combined system. But innermost termination is modular. (Innermost rewriting can be used to show that weak normalization is modular. See [Mid90] for more information on this and other modularity topics.)

A final advantage is that restricting completion to innermost derivations severely limits the number of critical pairs which need to be considered.

10.1 Modularity

Using forward closures, it is easy to show the modularity of innermost termination and even extend it to rewrite systems with shared constructor symbols. For a different proof of this see Gramlich [Gra93].

Theorem 54. *If R_1 and R_2 are rewrite systems which only share constructors and are innermost terminating, then their union is also innermost terminating*

Proof. Since both R_1 and R_2 are innermost terminating their innermost forward closures are also innermost terminating. But since one may only extend a forward closure by rewriting in produced context, none of the rules in R_1 can extend a forward closure from R_2 and vice-versa. Hence, the set of innermost forward closures for the combined rewrite system is just the union of the innermost forward closures of the individual systems, and must also be terminating. □

For example, consider the following rewrite system:

$$\begin{aligned} f(0,1,x) &\rightarrow f(x,x,x) \\ h &\rightarrow 0 \\ h &\rightarrow 1\ . \end{aligned} \tag{10.1}$$

It can be decomposed into the sum of two rewrite systems, one consisting of the first rule, and the other of the remaining two rules. Each of those systems is terminating, and therefore, the combined system must be innermost terminating. If one actually computes the forward closures for the combined system, the only forward closures are the rules themselves!

10.2 Critical Pairs and Local Confluence

When computing critical pairs, only overlaps at the top position need to be considered [Pla93b]. Overlaps at proper subterms need not be considered since one of the redexes will be below the other, and hence only one rule can be applied. Disjoint innermost rule applications can be applied independently in any order and local confluence holds. (For outermost derivations, however, disjoint outermost rule applications may not be applicable in any order, since the application of one rule can create a new outermost redex.)

For example, consider the following rewrite systems:

$$\begin{aligned} f(a) &\rightarrow f(a) \\ a &\rightarrow b\ , \end{aligned} \tag{10.2}$$

and

$$\begin{aligned} f(a) &\rightarrow f(c) \\ a &\rightarrow b\ . \end{aligned} \tag{10.3}$$

Both of these rewrite systems have no overlaps at the top and must be innermost locally-confluent. To show innermost termination, one can examine the innermost forward closures. For both rewrite systems, the *only* innermost forward closure is $a \rightarrow b$. Since this is terminating, the two systems must be innermost terminating as well. Innermost confluence then follows from Newman's lemma [New42].

10.3 Innermost Confluence without Termination

One reason that orthogonal rewrite systems are studied in depth is that they are confluent (regardless of whether they terminate or not.) This, in turn, guarantees that they have the unique normal form property and thus compute partial functions. In addition, if a term has a normal form, an outermost strategy for rewriting will derive the normal form.

A similar situation exists when restricted to innermost derivations.

Lemma 55. *A rewrite system with no overlaps at the top is confluent for innermost derivations.*

Proof. The only possible critical pairs arise from disjoint innermost redexes. These critical pairs are all strongly locally confluent (are joinable in a single step), and hence innermost rewriting is confluent. □

So for rewrite systems with no overlaps at the top, innermost rewriting has unique normal forms. Of course, innermost rewriting will find that normal form. Note, that the rewrite system may have other normal forms if non-innermost derivations are considered, e.g. System 10.3.

10.4 Implementing Innermost Forward Closures

For a rewrite system with top overlaps, one would want to complete it and show innermost termination. Innermost forward closures provide a method for showing innermost termination. Unfortunately, as the following example shows, forward closures may not terminate when the rewrite system is innermost terminating.

$$\begin{aligned} f(a,x) &\rightarrow g(x,x) \\ g(a,b(y)) &\rightarrow f(a,b(y)) \\ b(a) &\rightarrow a\ . \end{aligned} \tag{10.4}$$

The forward closures for the above include

$$\begin{aligned} g(a,b(y)) &\rightarrow f(a,b(y)) \rightarrow g(b(y),b(y)) \\ g(a,b(a)) &\rightarrow f(a,b(a)) \rightarrow f(a,a) \\ g(a,b(a)) &\rightarrow f(a,b(a)) \rightarrow g(b(a),b(a))\ . \end{aligned} \tag{10.5}$$

The third of these leads to an infinite forward closure, but when restricted to innermost derivations, both the second and third are disallowed. None of the other forward closures lead to non-termination; and the above rewrite system is innermost terminating. (It is also an example in which shared rewriting terminates.)

Thus, if one wants to show innermost termination, restricting forward closures to innermost derivations may be necessary.

10.4.1 Extending Innermost Forward Closures

One needs to know how to extend innermost forward closures. In particular, can an innermost forward closure ever arise from a forward closure which isn't innermost? Additionally, are there syntactic conditions which allow us to extend innermost forward closures without checking to make sure that previous derivations steps remain innermost?

While outermost forward closures have a more limited range of applicability, they will be considered as well for completeness.

Theorem 56. *If $s \to \cdots \to t$ is a forward closure of a rewrite system, but is not an innermost (outermost) forward closure then any extension $s\sigma \to \cdots \to t\sigma \to u$ is also not innermost (outermost).*

Proof. Consider the rewrite step which is not innermost (outermost). There must be a second redex below (above) which is innermost (outermost). Since extension of the forward closure can only substitute terms for variables uniformly, the viability of the second redex can not be affected. □

Thus one only needs to consider innermost (outermost) forward closures as candidates for extension when generating innermost (outermost) forward closures. Unfortunately, as the following rewrite system shows, the extension of an innermost forward closure need not be innermost, even if the rule application is innermost.

$$\begin{aligned} qx &\to fgx \\ fx &\to hx \\ g(a) &\to b \end{aligned} \tag{10.6}$$

As one can see, this rewrite system is overlaying, non-overlapping, left-linear, right-linear, non-erasing, and has unary function symbols. This virtually exhausts the standard syntactic categories for rewrite systems. It has the innermost forward closure $qx \to fgx \to hgx$. Its extension $qa \to fga \to hga \to hb$ is not innermost because of the redex ga in the second term.

On the other hand, for outermost forward closures the situation is not quite as bad.

Theorem 57. *If $s \to \cdots \to t$ is an outermost forward closure of an overlaying rewrite system with unary function symbols, then any extension $s\sigma \to \cdots \to t\sigma \to u$ is also outermost provided that $t\sigma \to u$ is an outermost rewrite.*

Proof. Suppose there is some rewrite step that is outermost in the original forward closure, but is no longer outermost after the extension. Its context must be included in the new outermost redex as a proper subterm. But this contradicts the assumption that the rewrite system is overlaying. □

In particular, string rewriting systems are unary, and non-erasing. By Proposition 45 non-overlapping string rewrite systems are terminating if their outermost forward closures are terminating. By the previous theorem, such systems have the desirable characteristic that when one is computing outermost forward closures, only the last term in the forward closures need to be remembered. An example of this is given in Section 8.7.

10.5 Computing Innermost Forward Closures

To compute innermost forward closures, one needs to check that when a forward closure is extended, each of the previous innermost rule applications remains innermost. This back checking can potentially lead to a lot of extra work. In general, one wants to avoid as much of the extra work as possible.

When extending an innermost forward closure $t_0(\bar{x}) \ldots \to \ldots t_n(\bar{x})$ via the substitution σ, the following check must be made:

- If rule $l \to r$ is applied at position p of $t_i(\bar{x})$, then no rule can match $t_i(\bar{x})\sigma$ at a position below p.

The following is an algorithm for extending the innermost forward closure $t_0(\bar{x}) \to \ldots \to \ldots t_n(\bar{x})$, by a set of rules $l_j \to r_j$:

1. For each subterm v of $t_n(\bar{x})$ starting at the bottom, check to see if l_j unifies with v resulting in the substitution σ. (One must rename the variables in l_j first.)

   (a) If some subterm of $v\sigma$ matches a rule, discard σ.

   (b) If σ is a match, do not check above v for extensions.

2. Check each of the previous terms t_i to see if a redex has been added below the rule application for $t_i\sigma$. If so, discard σ.

3. Extend the forward closure to $t_0\sigma \rightarrow \ldots \rightarrow t_n[v]\sigma \rightarrow t_n[r]\sigma$.

Notice that the unifications done before previous extensions of the forward closure are at positions that one needs to check for a match. So there are two things one wants to be able to do. First, one wants to be able to determine if a unifier is a match. Second, one wants to use the work done in previous unifications to check for matches. Both requirements can be met easily with a slight modification (denoted as *r-unification* for restricted unification) of the standard unification algorithm which employs a conditional rewrite system [MMR86].

Definition 15. When r-unifying a term $v(\hat{x})$ with the left-hand side of a rule $l(\hat{y})$, do not add a variable in $\hat{x}$ to the substitution if there is any other unification rule applicable.

A restricted unification is a match for $l(\hat{y})$ if there is a substitution for each variable in $\hat{y}$ and nothing else.

Lemma 58. *If σ is a unifier of a term $v(\hat{x})$ with the left-hand side of a rule $l(\hat{y})$ and μ is a substitution applied to $v(\hat{x})$, then $l(\hat{y})$ matches $v(\hat{x})\mu$ if $E(\sigma')\mu$ r-unifies to a match for the remaining variables of $\hat{y}$ in σ', where σ' contains the substitutions from σ for variables in $\hat{x}$, and $E(\sigma)$ is the set of equations obtained by converting all of the substitutions in σ into equations.*

Proof. When r-unifying $v(\hat{x})\mu$ with $l(\hat{y})$ the same rule applications can be used to produce the substitutions for the variables in $\hat{y}$ as before. But if one considers these substitutions as equations, they can never lead to a failure of restricted unification due to the substitution μ. There are only two ways that restricted unification can fail;

1. There is an occur check. But for each substitution $y_i \mapsto t$ there must be only one occurrence of y_i (otherwise it wouldn't have been part of the substitution). Since μ will not use y_i, there can be no occur check for $y_i = t$.

2. There is a mismatch of function symbols $f(\ldots) = g(\ldots)$. But since no substitution for y_i will be made in $y_i = t$, this can not happen either.

Therefore, there is a match if σ' with the substitution μ applied to it r-unifies to a substitution for the variables of $\hat{y}$ remaining in σ'. □

For example, consider the following rewrite system:

$$\begin{aligned} q(x) &\rightarrow r(f(h(x), a)) \\ h(a) &\rightarrow c \\ r(x) &\rightarrow b \\ f(x, y) &\rightarrow h(x) \ . \end{aligned} \tag{10.7}$$

The derivation $q(x) \rightarrow r(f(h(x), a))$ is an innermost forward closure. To extend this, one would r-unify first with the terms $h(x)$ and a. $h(x)$ r-unifies with the second rule giving the restricted unifier $\sigma_1 = \{x \mapsto a\}$. This can then be used to extend the forward closure. Notice that σ_1 is not a match, so one would also attempt to r-unify with $f(h(x), a)$. This r-unifies with the last rule giving $\sigma_2 = \{x_2 \mapsto h(x), y_2 \mapsto a\}$, where x_2 and y_2 are the renamed versions of the variables in the rule. Since this is a match, no extensions can result from unifications above this term.

The unifier σ_2 gives the extension $q(x) \rightarrow r(f(h(x), a)) \rightarrow r(h(x))$. To extend this forward closure, one would first r-unify with $h(x)$. This gives the restricted unifier $\sigma_3 = \{x \mapsto a\}$. To check that this is a valid extension, one computes the substitution $E(\sigma_1) = \{x = a\}$. Applying σ_3 to this results in $\{a = a\}$. This r-unifies to the empty substitution. Therefore, by Lemma 58, one has a match and σ_3 does not result in a valid extension.

10.6 Modifying Completion for Innermost Derivations

The completion process needs to be modified a little to handle innermost derivations. First, of course, only critical pairs at the top of rules need to be calculated.

Another change is illustrated by the following rewrite system:

$$\begin{aligned} hfx &\rightarrow ghfx \\ fx &\rightarrow a \ . \end{aligned} \tag{10.8}$$

The first rule in the rewrite system can never be an innermost redex since the second rule matches a subterm. Therefore, the first rule should be discarded.

A more serious change is illustrated by the rewrite system:

$$\begin{aligned} f(ha, ga) &\rightarrow b \\ q &\rightarrow ha \\ q &\rightarrow ga\ . \end{aligned} \tag{10.9}$$

The only critical pair is $ha = ga$. No matter which direction one orients the equation, the first rule will be affected. In the original rewrite system there was an innermost derivation $f(ha, ga) \rightarrow b$, but in the extended rewrite system with the rule $ha \rightarrow ga$ added, the derivation is no longer innermost. One must rewrite the left-hand side of the first rule to preserve the derivation, resulting in the new rewrite system:

$$\begin{aligned} f(ga, ga) &\rightarrow b \\ q &\rightarrow ha \\ q &\rightarrow ga \\ ha &\rightarrow ga\ . \end{aligned} \tag{10.10}$$

A slightly more complicated example is the rewrite system:

$$\begin{aligned} fgx &\rightarrow hx \\ c &\rightarrow ga \\ c &\rightarrow a\ . \end{aligned} \tag{10.11}$$

The only critical pair is $ga = a$, which must be oriented as $ga \rightarrow a$ to ensure termination. In the original system there was an innermost derivation $fga \rightarrow ha$, so to preserve this derivation

one must add the rule $fa \rightarrow ha$, giving:

$$\begin{aligned} fgx &\rightarrow hx \\ fa &\rightarrow ha \\ ga &\rightarrow a \\ c &\rightarrow ga \\ c &\rightarrow a\ . \end{aligned} \tag{10.12}$$

Notice that the rewrite rule $fgx \rightarrow hx$ needs to be kept to maintain the innermost derivation $fgb \rightarrow hb$. The original innermost derivation $fga \rightarrow ha$, is now $fga \rightarrow fa \rightarrow ha$, and is completely handled by the new rewrite rules without any interference from the original rule.

Given a set of rewrite rules $\{l_i \rightarrow r_i\}$, the steps to be followed in innermost completion are:

1. Discard any rule $l_i \rightarrow r_i$, if there is some other rule $l_j \rightarrow r_j$ which matches a proper subterm of l_i.
2. Compute all of the critical pairs for overlaps at the tops of rules.
3. Repeat the following until there are no critical pairs:
   (a) Choose a critical pair.
   (b) Orient the critical pair to a rewrite rule $l_k \rightarrow r_k$.
   (c) Revise the rewrite system with the new rule.

To revise a rewrite system with a new rule $l_k \rightarrow r_k$ do the following:

1. Check to see if l_k unifies with a non-variable subterm s of the left-hand side of each rule $l_i = t[s]$. The unifier is σ.
2. If so construct a new rule, $n \rightarrow r_i\sigma$, where $l_i\sigma$ has an innermost derivation of n in the original rewrite system with the new rule added.
3. If unifier σ corresponded to a match, remove the rule $l_i \rightarrow r_i$.
4. Revise the new rewrite system with each of the constructed rules $n \rightarrow r_i\sigma$.

Consider the following rewrite system:

$$\begin{aligned} tx &\rightarrow gx \\ tx &\rightarrow fx \\ fa &\rightarrow gb \\ gb &\rightarrow ga \\ ga &\rightarrow fb\ . \end{aligned} \tag{10.13}$$

It demonstrates that it is not always possible to construct an innermost complete system for a rewrite system which preserves the original innermost derivations. It has the critical pair $gx = fx$. Orienting this equation in either direction will lead to non-termination. Effectively, this is like fixing an ordering in the original Knuth-Bendix completion, since one is preserving innermost derivations of a given rewrite system. Of course, if one does not care if the original innermost derivations of a rewrite system are preserved, then just use the standard Knuth-Bendix completion to find a canonical rewrite system. Since that system is terminating, it is innermost terminating as well.

11 SUMMARY AND FUTURE WORK

The general path ordering is a powerful general purpose tool for demonstrating termination of rewrite systems. It can be applied in situations in which the more familiar simplification orderings cannot, as when the rewrite system is self-embedding. It encompasses virtually all popular methods, including polynomial (and other) interpretations, the Knuth-Bendix ordering and its extensions, and the recursive path orderings and its variants. Geser [Ges94] has suggested a weakening of the subterm conditions, thereby strengthening the general path ordering.

Two versions of the general path ordering were presented in this thesis. One requires that the component orderings be well-founded quasi-orderings. It was shown that this definition has the advantage that it satisfies an incrementality property (it is stable under extensions of its component orderings). The other ordering uses well-quasi orderings for the component orderings. This version does not require that all the subterms of a term be examined by the lexicographical portion of the ordering. Unfortunately it is not incremental.

One question left to be answered is whether the well-founded general path ordering requires the addition of the subterm condition as detailed in Section 7.4 for Theorem 20 to be valid.

Several examples, including 6.1, were mechanically verified by the general path ordering termination code (GPOTC). The implementation supports termination functions for precedence, term extraction (given and maximum), and homomorphisms. Interpretations involving addition, multiplication, negation, and exponentiation are expressible. Currently, the burden of proving that functions are either value-preserving or monotonic is placed on the user. As is usual for such functions, one often ends up needing to know if a given function is positive over some range. When the functions are rational polynomials, this is decidable, but time consuming. The code does not attempt a full solution, but merely applies some quick and dirty heuristics, such as testing the function at endpoints and checking coefficients of polynomials. In cases where the code cannot make a determination, it will query the user for an authoritative answer. The part of the code that does this testing could be upgraded to provide heuristics such as those described in Lankford [Lan79], Ben Cherifa and Lescanne [CL87], or Steinbach and Zehnter [SZ90].

Forward closures provide a more specialized method for showing termination, applicable to locally-confluent overlaying or right-linear systems. Special cases of interest are orthogonal and string rewrite systems which are terminating whenever their forward closures are. In addition, when the rewrite system is non-erasing (as for string systems) the set of forward closures can be restricted to just the innermost forward closures, easing proof of termination. Furthermore, if the system is non-overlapping, any rewrite strategy will suffice to restrict the set of forward closures. One important result of this work was the demonstration of the connection between forward closures and the termination of the innermost derivations of a rewrite system. If one can find some condition such that innermost termination of a rewrite systems implies general termination, then the termination of forward closures implies general termination of the rewrite system.

An interesting question is whether there are analogous theorems that can be shown for rewrite systems with the addition of an equational theory, such as Associativity and/or Commutativity. In the presence of such an equational theory, the notion of an innermost derivation is no longer a simple matter. In addition, if an analog to forward closures is found, there are serious implementation issues to consider. For example, AC-unification is very time consuming and while it is guaranteed to produce a finite set of unifiers, in general, the number of such unifiers can be exponential in the number of variables. One technique for dealing with AC is to delay solving for the unifier and keep a set of constraints. But standard forward closures will unify rules with the non-variable parts of a term, which may have been produced as part of the unification, so one probably can not completely delay the unification.

Both of the methods for showing termination presented in this thesis can often lead to more natural proofs, using arguments similar to those used for recursive definitions. Often with forward closures, some property of the derivations (for example, all forward closures have exactly one occurrence of a given function symbol) allows one to use a simpler argument to complete the proof of termination.

Also presented in this thesis was an application of forward closures for detecting non-termination of a rewrite system. A previous method MSP due to Purdom was analyzed and certain restrictions on the ability of MSP to detect non-terminating derivations were presented. The main result showed that certain subterms of the right-hand sides of rules could not contribute to a non-terminating sequence. An important instance of this showed that any vari-

able subterm does not contribute. Using this result, one can show that any non-terminating derivation that MSP detects will also be found by the test using forward closures. Similarly, one can show that there are non-terminating rewrite systems MSP will not discover a non-terminating derivation, but the forward closures does. A simple example is the rewrite system $\{fffx \to fgfx, gx \to x\}$. The second rule can not be used an any derivation for MSP and since the first rule by itself is terminating, MSP will not find a non-terminating derivation. On the other hand, the almost trivial forward closure $fffx \to fgfx \to fffx$ demonstrates non-termination for this rewrite system.

It is also shown in the thesis that the forward closures method (FCT) has other advantages as well in the context of completion. In particular, the forward closures can give some information about the innermost termination properties of terms of limited height where the subterms are all in normal form. This potentially allows one to not check for termination when computing the joinability of critical pairs during completion.

Innermost and outermost forward closures are also useful, but require some thought to implement correctly. It is shown that an extension of an innermost/outermost forward closure may not itself be an innermost/outermost derivation. A method is presented for avoiding extra work in checking back along a forward closure for such violations. In general, the number of forward closures may be exponential in the length of the derivation, potentially limiting the length of derivations that one could check automatically. One can avoid some of the problems by discarding forward closures that duplicate the last term in the derivation. Even that checking can be costly and methods for reducing it should be considered.

Finally, some results are presented which show the usefulness of forward closures when you restrict your rewrite system to innermost derivations only. In particular, generating a rewrite system that is innermost locally confluent and preserves innermost derivations is examined and a method similar to completion is presented. If the rewrite system is innermost terminating then it will be innermost confluent as well. Forward closures are a natural method for showing innermost termination. This has potential application for use with most programming languages, since innermost computation strategies are common.

A DESCRIPTION OF GPO CODE

There is code available which implements parts of the well-quasi general path ordering described in this thesis. The most recent version can be obtained by e-mailing a request the author. The following gives a brief description of the functions available in the interface.

The code was developed in Macintosh Common Lisp and just uses functions/special forms as described in Steele [Ste90]. The code uses packages for modularity and information hiding.

A.1 General Description of the System for GPO

Before running the general path ordering, you will need to adjust the path names for the files in the file `Start.Lisp`. Once this has been done, the system can be started by loading the file `Start.Lisp`. It will load the source files in an appropriate order. Once this is done, the user should define a set of rules and an ordering. The system is then ready.

`(init-genord)` sets the value of all global variables to their default values. (`Start.Lisp` invokes this function automatically.) This version of the general path ordering uses the Well-Quasi definition. You must guarantee that all of the component orderings used are well-quasi orderings.

`(print-rule rule stream)` prints a single rule in a readable format.

`(print-list-of-rules rules stream)` prints a list of rules using `(print-rule)`.

`(term-cond productions comp)` applies the ordering comp to each of the rules in the list productions. The result is a list of values, one for each rule, indicating whether the right-hand side was greater than the left-hand side. If all the results are `GR` and the components are acceptable, the rewriting system terminates.

`(back-cond productions comp)` similar to the previous, but the left-hand side is compared to the right-hand side, effectively examining a system in which all the rules are reversed.

`(toggle-show-causes)` switches the global variable `*genord-show-partial-results*` between `t` and `nil`. This variable controls whether the results to each call of the general ordering and the lexical component are displayed.

A.2 Defining a Rule

Each rule is stored in a production struct which has two components; a left hand side (`lhs`) and a right hand side (`rhs`). These are constructed from constant symbols, function symbols and variables. A constant symbol is indicated to the system with a `!`. A function is constructed using parenthesis and uses prefix notation. The function symbol itself is indicated with a `!`. A variable is indicated with a `?`.

Function and constant symbols can be nearly any string of contiguous non-space characters. The only caveat is that upper and lower case are indistinguishable. `!ConsT` and `!cONSt` are the same symbol. The same criteria applies to variables.

To make a production the built-in structure construction should be used. For example the rule

$$f(a, h(x)) \rightarrow 0 \tag{A.1}$$

would be created by

```
(make-production :lhs '(!f !a (!h ?X)) :rhs '!0).
```

A.3 Defining an Ordering

An ordering function accepts two terms as arguments and returns one of three values: `GR`, `EQ`, or `UN`. `GR` indicates that the first argument was greater in the ordering than the second. `EQ` indicates that the two argument were equal. `UN` indicates that either the first argument was less than the second or that the two arguments were incomparable.

`(makeorder name complist)` is a macro which creates an ordering function with the given name using the list of component orderings. A component ordering is a function from terms to elements of a well-founded set. The components are evaluated lexicographically from first to last in the list.

A.3.1 Precedence Component

`(make_prec_comp q-ordering)` is a macro which creates a precedence ordering on symbols. `q-ordering` is a list of symbols with the highest precedence first. A sublist is used to make two or more symbols with equal precedence. The following makes an ordering with $+ > g = h > *$.

```
(make_prec_comp '(!+ (!g !h) !*) )
```

A.3.2 Subterm Extraction Component

`(make_subterm_comp num-of-term gen)` is a macro which creates a component to extract a given subterm. Terms are numbered from 1 going left to right. `gen` is an ordering to be applied to the extracted subterms. It is intended that this be a recursive call to the ordering that this component is being used in. For other orderings termination is not guaranteed and the user is responsible for verifying that the appropriate conditions are met. An example which first uses the precedence ordering given above then extracts leftmost subterms (on `g` and `h`) is:

```
(makeorder AnOrder (list
                    (make_prec_comp '(!+ (!g !h) !*) )
                    (make_subterm_comp 1 AnOrder)))
```

A.3.3 Maximum Subterm Extraction Component

`(make_maxsubterm_comp gen)` is a macro which creates a comp to extract the maximum subterm using the ordering `gen`. It is intended that this be a recursive call to the ordering that this component is being used in. For other orderings termination is not guaranteed and the user is responsible for verifying that the appropriate conditions are met. Since this just extracts a single term, the user should verify that the general path ordering is total first.

A.3.4 Homomorphism Components

Homomorphism and monotonic functions are supported with a single macro. The functions which are supported are constants, addition, multiplication, exponentiation and evaluation of subterms. The user is responsible for verifying that the particular function he defines meets conditions for termination.

`(make_function_comp FNdefs)` is a macro which creates a component to evaluates subterms using the definitions in the list `FNdefs`. Each definition is a pair associating a symbol of the rewriting system with a function.

- constant: any value such as -1, 0, 2, etc.

- addition: `(+ ...)`
- multiplication: `(* ...)`
- exponentiation: `(^ base exponent)`
- subterm: `(arg number)`

For example, the following associates $f[X,Y]$ with the function $X \times Y + 2$, $g[X]$ with the function 2^X, and $c[]$ with the constant function 1.

```
(make_function_comp (
                     (!f (+ (* (arg 1) (arg 2)) 2))
                     (!g (^ 2 (arg 1)))
                     (!c 1)))
```

When the macro is evaluated the user is queried for the minimum value which a term can attain. In the previous example this is `1`, the value of the constant `c`.

A.3.5 Determining Positiveness

In evaluating function components, the question of whether a particular term is positive arises. In particular, the question is "Is the term strictly greater than zero when evaluated for all values of the variables greater than the minimum?"

There are a number of heuristics incorporated in the system for determining if a polynomial is positive. Since, few of the heuristics apply to exponentiation, one should only use it if it is required. For example, use `(* (arg 1) (arg 1))` instead of `(^ (arg 1) 2)`. In those cases, where the system is unable to make a determination, the user is prompted for a response. The allowed responses are `P` for strictly positive over the domain of interest, `Z` for identically zero over the domain, and `N` for negative or zero at some point in the domain.

The system stores away such determinations for future use. All determinations are added to the list held by the variable `*genord-all-positiveness-results*` while only those supplied by the user are added to `*genord-user-provided-positiveness-results*`. Since `cons` is used to add to the list, items will be in reverse order with respect to when they were encountered by the system.

`(init-positiveness-checking)` resets the global variables containing the positiveness results to `nil`.

`(show-user-positiveness-results)` displays all of the terms for which the user was queried along with response the user gave.

`(show-all-positiveness-results)` displays all of the terms for which a determination was made. The result which was obtained is given along with a reason.

A.4 Examples Files

The example files are all in the subdirectory `Examples`. There are a number of functions which are defined in the example files for your convenience in running them.

`(load-example aFile &optional quiet)` attempts to load the file in the sources directory with the name `".example"` appended on the string given by `aFile`. Normally, the rules and the ordering will be displayed with `(show-rules)` and `(show-order)` as the file is loaded. If the optional argument quiet is `t` this will be suppressed.

If there is a homomorphism component defined in the file, you will be queried for a minimum value at this point.

`(go-forward)` Is redefined for each of the examples to invoke the function `(term-cond productions comp)` with the set of productions and comparison ordering defined in the example file.

`(go-backward)` Is redefined for each of the examples to invoke the function (back-cond productions comp) with the set of productions (rules) and comparison ordering defined in the example file.

`(show-rules)` will print the productions defined in the example file with a brief description/comment.

`(show-order)` will print a pretty printed version of the code defining the ordering used for the example.

B USING GPOTC TO SHOW TERMINATION OF INSERTION SORT

The following is a Lisp Session in which the insertion sort example given in 6.1 is mechanically examined with the well-quasi general path ordering code.

```
? (load "Internal Mirror:Lisp Files:GPO:Start-GenOrd.lisp")
Loading files for general path ordering
Initialization for General Ordering Termination Code complete
For information and help open the file README
#P"Internal Mirror:Lisp Files:GPO:Start-GenOrd.lisp"
? (load-example "insertionSort")

Rules for insertion sort

RULE 1:  sort(nil)  -->  nil
RULE 2:  sort(cons(X, Y))  -->  insert(X, sort(Y))
RULE 3:  insert(X, nil)  -->  cons(X, nil)
RULE 4:  insert(X, cons(V, W))  -->  choose(X, cons(V, W), X, V)
RULE 5:  choose(X, cons(V, W), 0, 0)  -->  cons(X, cons(V, W))
RULE 6:  choose(X, cons(V, W), s(P), 0)  -->  cons(X, cons(V, W))
RULE 7:  choose(X, cons(V, W), 0, s(Q))  -->  cons(V, insert(X, W))
RULE 8:  choose(X, cons(V, W), s(P), s(Q))  -->  choose(X, cons(V, W), P, Q)

Rule listing completed
The lisp code for the ordering is given by:

(MAKEORDER ORD1
           (LIST (MAKE_PREC_TAU SYMORD2)
                 (MAKE_SUBTERM_TAU ((sort 1) (choose 2) (insert 2)) ORD1)
                 (MAKE_PREC_TAU SYMORD1)
                 (MAKE_SUBTERM_TAU ((sort 1) (choose 3) (insert 2)) ORD1)))
```

```
The symbol ordering SymOrd1 is defined by:

(SETF SYMORD1 '(sort insert choose cons))

The symbol ordering SymOrd2 is defined by:

(SETF SYMORD2 '(sort (insert choose) cons))
;Compiler warnings :
;   Undeclared free variable SYMORD1 (2 references), in an anonymous lambda
form inside GENORD::WQO-COMPARE inside ORD1.
;   Undeclared free variable SYMORD2 (2 references), in an anonymous lambda
form inside GENORD::WQO-COMPARE inside ORD1.
T
? (go-forward)
(:GR :GR :GR :GR :GR :GR :GR :GR)
? (toggle-show-causes)
T
? (go-forward)

 sort(nil)  -->  nil
 sort(nil) > nil by case (1)
 | nil is syntactically equal to term nil

 sort(cons(X, Y))  -->  insert(X, sort(Y))
 sort(cons(X, Y)) > insert(X, sort(Y)) by case (2)
 Case 2a: Check that the LHS > all subterms of the RHS:
 | sort(cons(X, Y)) > X by case (1)
 | | cons(X, Y) >= X by case (1)
 | | | X is syntactically equal to term X
 | |
 | sort(cons(X, Y)) > sort(Y) by case (2)
 | Case 2a: Check that the LHS > all subterms of the RHS:
 | | sort(cons(X, Y)) > Y by case (1)
 | | | cons(X, Y) >= Y by case (1)
 | | | | Y is syntactically equal to term Y
```

```
| Case 2b: Check that the LHS > RHS via lexicographic comparison:
| | 1:sort(cons(X, Y)) >= sort(Y) by basic ordering of a precedence
| | |
| | 2:immediate subterms sort|1 with sort|1: cons(X, Y) > Y
| | | cons(X, Y) > Y by case (1)
| | | | Y is syntactically equal to term Y
Case 2b: Check that the LHS > RHS via lexicographic comparison:
| 1:sort(cons(X, Y)) > insert(X, sort(Y)) by basic ordering of a precedence

insert(X, nil)  -->  cons(X, nil)
insert(X, nil) > cons(X, nil) by case (2)
Case 2a: Check that the LHS > all subterms of the RHS:
| insert(X, nil) > X by case (1)
| | X is syntactically equal to term X
| |
| insert(X, nil) > nil by case (1)
| | nil is syntactically equal to term nil
Case 2b: Check that the LHS > RHS via lexicographic comparison:
| 1:insert(X, nil) > cons(X, nil) by basic ordering of a precedence

insert(X, cons(V, W))  -->  choose(X, cons(V, W), X, V)
insert(X, cons(V, W)) > choose(X, cons(V, W), X, V) by case (2)
Case 2a: Check that the LHS > all subterms of the RHS:
| insert(X, cons(V, W)) > X by case (1)
| | X is syntactically equal to term X
| |
| insert(X, cons(V, W)) > cons(V, W) by case (1)
| | cons(V, W) is syntactically equal to term cons(V, W)
| |
| insert(X, cons(V, W)) > X by case (1)
| | X is syntactically equal to term X
| |
| insert(X, cons(V, W)) > V by case (1)
| | cons(V, W) >= V by case (1)
| | | V is syntactically equal to term V
```

```
 Case 2b: Check that the LHS > RHS via lexicographic comparison:
 | 1:insert(X, cons(V, W)) >= choose(X, cons(V, W), X, V) by basic ordering of a
precedence
 | |
 | 2:immediate subterms insert|2 with choose|2: cons(V, W) >= cons(V, W)
 | | cons(V, W) is syntactically equal to term cons(V, W)
 | |
 | 3:insert(X, cons(V, W)) > choose(X, cons(V, W), X, V) by basic ordering of a
precedence

 choose(X, cons(V, W), 0, 0)  -->  cons(X, cons(V, W))
 choose(X, cons(V, W), 0, 0) > cons(X, cons(V, W)) by case (2)
 Case 2a: Check that the LHS > all subterms of the RHS:
 | choose(X, cons(V, W), 0, 0) > X by case (1)
 | | X is syntactically equal to term X
 | |
 | choose(X, cons(V, W), 0, 0) > cons(V, W) by case (1)
 | | cons(V, W) is syntactically equal to term cons(V, W)
 Case 2b: Check that the LHS > RHS via lexicographic comparison:
 | 1:choose(X, cons(V, W), 0, 0) > cons(X, cons(V, W)) by basic ordering of a
precedence

 choose(X, cons(V, W), s(P), 0)  -->  cons(X, cons(V, W))
 choose(X, cons(V, W), s(P), 0) > cons(X, cons(V, W)) by case (2)
 Case 2a: Check that the LHS > all subterms of the RHS:
 | choose(X, cons(V, W), s(P), 0) > X by case (1)
 | | X is syntactically equal to term X
 | |
 | choose(X, cons(V, W), s(P), 0) > cons(V, W) by case (1)
 | | cons(V, W) is syntactically equal to term cons(V, W)
 Case 2b: Check that the LHS > RHS via lexicographic comparison:
 | 1:choose(X, cons(V, W), s(P), 0) > cons(X, cons(V, W)) by basic ordering of a
precedence

 choose(X, cons(V, W), 0, s(Q))  -->  cons(V, insert(X, W))
```

```
 choose(X, cons(V, W), 0, s(Q)) > cons(V, insert(X, W)) by case (2)
 Case 2a: Check that the LHS > all subterms of the RHS:
 | choose(X, cons(V, W), 0, s(Q)) > V by case (1)
 | | cons(V, W) >= V by case (1)
 | | | V is syntactically equal to term V
 | |
 | choose(X, cons(V, W), 0, s(Q)) > insert(X, W) by case (2)
 | Case 2a: Check that the LHS > all subterms of the RHS:
 | | choose(X, cons(V, W), 0, s(Q)) > X by case (1)
 | | | X is syntactically equal to term X
 | | |
 | | choose(X, cons(V, W), 0, s(Q)) > W by case (1)
 | | | cons(V, W) >= W by case (1)
 | | | | W is syntactically equal to term W
 | Case 2b: Check that the LHS > RHS via lexicographic comparison:
 | | 1:choose(X, cons(V, W), 0, s(Q)) >= insert(X, W) by basic ordering of a
precedence
 | | |
 | | 2:immediate subterms choose|2 with insert|2: cons(V, W) > W
 | | | cons(V, W) > W by case (1)
 | | | | W is syntactically equal to term W
 Case 2b: Check that the LHS > RHS via lexicographic comparison:
 | 1:choose(X, cons(V, W), 0, s(Q)) > cons(V, insert(X, W)) by basic ordering
of a precedence

 choose(X, cons(V, W), s(P), s(Q))  -->  choose(X, cons(V, W), P, Q)
 choose(X, cons(V, W), s(P), s(Q)) > choose(X, cons(V, W), P, Q) by case (2)
 Case 2a: Check that the LHS > all subterms of the RHS:
 | choose(X, cons(V, W), s(P), s(Q)) > X by case (1)
 | | X is syntactically equal to term X
 | |
 | choose(X, cons(V, W), s(P), s(Q)) > cons(V, W) by case (1)
 | | cons(V, W) is syntactically equal to term cons(V, W)
 | |
 | choose(X, cons(V, W), s(P), s(Q)) > P by case (1)
```

```
 | | s(P) >= P by case (1)
 | | | P is syntactically equal to term P
 | |
 | choose(X, cons(V, W), s(P), s(Q)) > Q by case (1)
 | | s(Q) >= Q by case (1)
 | | | Q is syntactically equal to term Q
 Case 2b: Check that the LHS > RHS via lexicographic comparison:
 | 1:choose(X, cons(V, W), s(P), s(Q)) >= choose(X, cons(V, W), P, Q) by basic
ordering of a precedence
 | |
 | 2:immediate subterms choose|2 with choose|2: cons(V, W) >= cons(V, W)
 | | cons(V, W) is syntactically equal to term cons(V, W)
 | |
 | 3:choose(X, cons(V, W), s(P), s(Q)) >= choose(X, cons(V, W), P, Q) by basic
ordering of a precedence
 | |
 | 4:immediate subterms choose|3 with choose|3: s(P) > P
 | | s(P) > P by case (1)
 | | | P is syntactically equal to term P
(:GR :GR :GR :GR :GR :GR :GR :GR)
?
```

BIBLIOGRAPHY

[BD94] Leo Bachmair and Nachum Dershowitz. Equational inference, canonical proofs, and proof orderings. *J. of the Association for Computing Machinery*, 41(2):236–276, March 1994.

[Bit93] Elias Tahhan Bittar. Non erasing, right linear, orthogonal term rewrite systems application to Zantema's problem. Technical Report RR 2202, INRIA, 1993.

[CL87] Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, October 1987.

[Dau89] M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications (Chapel Hill, NC)*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120, Berlin, April 1989. Springer-Verlag.

[Der81] Nachum Dershowitz. Termination of linear rewriting systems. In *Proceedings of the Eighth International Colloquium on Automata, Languages and Programming (Acre, Israel)*, volume 115 of *Lecture Notes in Computer Science*, pages 448–458, Berlin, July 1981. European Association of Theoretical Computer Science, Springer-Verlag.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.

[Der87] Nachum Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3(1&2):69–115, February/April 1987. Corrigendum: *4*, 3 (December 1987), 409–410; reprinted in *Rewriting Techniques and Applications*, J.-P. Jouannaud, ed., pp. 69—115, Academic Press, 1987.

[Der95] Nachum Dershowitz. 33 examples of termination. In H. Comon and J.-P. Jouannaud, editors, *French Spring School of Theoretical Computer Science Advanced Course on*

Term Rewriting (Font Romeux, France, May 1993), volume 909 of *Lecture Notes in Computer Science*, pages 16–26, Berlin, 1995. Springer-Verlag.

[DH95] Nachum Dershowitz and Charles Hoot. Natural termination. *Theoretical Computer Science*, 142:170–207, May 1995.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.

[Ges93] Alfons Geser. A solution to zantema's problem. Technical Report MIP-9314, Universität Passau, Passau, Germany, December 1993.

[Ges94] Alfons Geser. An improved general path order. Technical Report MIP-9407, Universität Passau, Passau, Germany, June 1994.

[Geu89] Oliver Geupel. Overlap closures and termination of term rewriting systems. Report MIP-8922, Universität Passau, Passau, West Germany, July 1989.

[GKM83] John V. Guttag, Deepak Kapur, and David R. Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM J. on Computing*, 12(1):189–214, February 1983.

[GNP+93] Jean Gallier, Paliath Narendran, David Plaisted, Stan Raatz, and Wayne Snyder. An algorithm for finding canonical sets of ground rewrite rules in polynomial time. *Journal of the Association for Computing Machinery*, 40(1):1–16, 1993.

[Gra92] Bernhard Gramlich. Relating innermost, weak, uniform and modular termination of term rewriting systems. In A. Voronkov, editor, *Proceedings of the Conference on Logic Programming and Automated Reasoning (St. Petersburg, Russia)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 285–296, Berlin, July 1992. Springer-Verlag.

[Gra93] Bernhard Gramlich. On termination, innermost termination and completeness of heirarchically structured rewrite systems. Technical Report SR-93-09, SEKI, Universitat Kaiserslautern, 1993.

[Gra94] Bernhard Gramlich. personal communication, 1994.

[Hig52] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society (3)*, 2(7):326–336, September 1952.

[HL78] Gérard Huet and Dallas S. Lankford. On the uniform halting problem for term rewriting systems. Rapport laboria 283, Institut de Recherche en Informatique et en Automatique, Le Chesnay, France, March 1978.

[HR86] Jieh Hsiang and Michaël Rusinowitch. A new method for establishing refutational completeness in theorem proving. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction (Oxford, England)*, volume 230 of *Lecture Notes in Computer Science*, pages 141–152, Berlin, July 1986. Springer-Verlag.

[KB70] Donald E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, U. K., 1970. Reprinted in *Automation of Reasoning 2*, Springer-Verlag, Berlin, pp. 342–376 (1983).

[KL80] Sam Kamin and Jean-Jacques Lévy. Two generalizations of the recursive path ordering. Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL, February 1980.

[Klo92] Jan Willem Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, Oxford, 1992.

[KMNS88] Deepak Kapur, David Musser, Paliath Narendran, and Jonathan Stillman. Semi-unification. In G. Goos and J. Harmanis, editors, *Foundation of Software Technology and Theoretical Computer Science, 8th Conference, (Pune, India)*, Lecture Notes in Computer Science, Berlin, December 1988. Springer-Verlag.

[Kru54] Joseph B. Kruskal. *The theory of well-partially-ordered sets.* PhD thesis, Princeton, NJ, June 1954. Ph.D. thesis.

[Kru60] Joseph B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, May 1960.

[Kru72] Joseph B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *J. Combinatorial Theory Ser. A*, 13(3):297–305, November 1972.

[Lan79] Dallas S. Lankford. On proving term rewriting systems are Noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, LA, May 1979. Revised October 1979.

[Les90] Pierre Lescanne. On the recursive decomposition ordering with lexicographical status and other related orderings. *J. Automated Reasoning*, 6:39–49, 1990.

[LM78] Dallas S. Lankford and David R. Musser. A finite termination criterion. May 1978.

[LS77] R. Lipton and L. Snyder. On the halting of tree replacement systems. In *Proceedings of the Conference on Theoretical Computer Science*, pages 43–46, Waterloo, Canada, August 1977.

[Mar89] Ursula Martin. A geometrical approach to multiset orderings. *Information Processing Letters*, 67:37–54, May 1989.

[McN94] Robert McNaughton. The uniform halting problem for one-rule semi-thue systems. Technical Report 94-18, Rensselaer, Department of Computer Science, Rensselaer Polytechnic Institute,Troy, New York 12180-3590, August 1994.

[Mid90] Aart Middeldorp. *Modular Properties of Term Rewriting Systems.* PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1990.

[Mid94] Aart Middeldorp. A simple proof to a result of Bernhard Gramlich. Unpublished note, February 1994.

[MMR86] A. Martelli, C. Moiso, and G. F. Rossi. An algorithm for unification in equational theories. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 180–186, Salt Lake City, UT, September 1986.

[MN70] Zohar Manna and Steven Ness. On the termination of Markov algorithms. In *Proceedings of the Third Hawaii International Conference on System Science*, pages 789–792, Honolulu, HI, January 1970.

[New42] M. H. A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[O'D77] Michael J. O'Donnell. *Computing in systems described by equations*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1977.

[Par78] Rohit Parikh. Effectiveness. Technical Report Report, July 1978.

[Pla78] David A. Plaisted. Well-founded orderings for proving termination of systems of rewrite rules. Report R-78-932, Department of Computer Science, University of Illinois, Urbana, IL, July 1978.

[Pla79] David A. Plaisted. Personal communication, 1979. Department of Computer Science, University of Illinois.

[Pla86] David A. Plaisted. A simple non-termination test for the Knuth-Bendix method. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction (Oxford, England)*, volume 230 of *Lecture Notes in Computer Science*, pages 79–88, Berlin, July 1986. Springer-Verlag.

[Pla93a] David A. Plaisted. Polynomial time termination and constraint satisfaction tests. In Claude Kirchner, editor, *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications*, pages 405–420, Montreal, Canada, June 1993. Vol. 690 of *Lecture Notes in Computer Science*, Springer , Berlin.

[Pla93b] David A. Plaisted. Term rewriting systems. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, chapter 2. Oxford University Press, Oxford, 1993. To appear.

[Pur87] Paul Purdom. Detecting loop simplifications. In P. Lescanne, editor, *Proceedings of the Second International Conference on Rewriting Techniques and Applications (Bordeaux, France)*, volume 256 of *Lecture Notes in Computer Science*, pages 54–61, Berlin, May 1987. Springer-Verlag.

[Ste89] Joachim Steinbach. Extensions and comparison of simplification orderings. In Nachum Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications (Chapel Hill, NC)*, volume 355 of *Lecture Notes in Computer Science*, pages 434–448, Berlin, April 1989. Springer-Verlag.

[Ste90] Guy L. Steele, Jr. *Common Lisp:The Language, Second Edition.* Digital Press, Bedford, Massachusetts, 1990.

[SZ90] Joachim Steinbach and Michael Zehnter. Vade-mecum of polynomial orderings. Report SR-90-03, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, West Germany, 1990.

[TKB89] Yoshihito Toyama, Jan Willem Klop, and Hendrik Pieter Barendregt. Termination for the direct sum of left-linear term rewriting systems. In Nachum Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications (Chapel Hill, NC)*, volume 355 of *Lecture Notes in Computer Science*, pages 477–491, Berlin, April 1989. Springer-Verlag.

[Zan92a] Hans Zantema. personal communication, 1992.

[Zan92b] Hans Zantema. Termination of term rewriting by semantic labelling. Technical Report RUU-CS-92-38, Utecht University, Utrecht, the Netherlands, December 1992.

VITA

Charles Glen Hoot was born on May 23, 1960 in Boulder, Colorado. He attended the University of California at San Diego and graduated cum laude with a Bachelor of Arts degree in Physics with a minor in Applied Mechanics, in June 1982.

He then attended graduate school at Princeton University and received a Master of Arts degree in Astrophysics, in January 1985. Subsequently, he worked at the Aerospace Corporation in El Segundo, California as a Member of the Technical Staff from April 1985 to July 1989. He went to the University of Illinois at Urbana-Champaign for further graduate studies culminating in this doctoral thesis.