Equational Logic Programming

Michael J. O'Donnell

Contents

This text is a preprint of Chapter 2 in the *Handbook of Logic in Artificial Intelligence and Logic Programming,* volume 5 on *Logic Programming,* Dov Gabbay editor, Oxford University Press. Donald W. Loveland, Gopalan Nadathur, and David J. Sherman served as official readers, and provided useful feedback on drafts of the chapter.

1 Introduction to Equational Logic Programming

1.1 Survey of Prerequisites

Sections 2.3.4 and 2.3.5 of the chapter 'Introduction: Logic and Logic Programming Languages' are crucial prerequisites to this chapter. I summarize their relevance below, but do not repeat their content.

Logic programming languages in general are those that compute by deriving semantic consequences of given formulae in order to answer questions. In equational logic programming languages, the formulae are all equations expressing postulated properties of certain functions, and the questions ask for equivalent normal forms for given terms. Section 2.3.4 of the 'Introduction . . . ' chapter gives definitions of the models of equational logic, the semantic consequence relation

$$\mathbf{T} \models_{\doteq} (t_1 \doteq t_2)$$

($t_1 \doteq t_2$ is a semantic consequence of the set $\mathbf{T}$ of equations, see Definition 2.3.14), and the question answering relation

$$(\mathbf{norm}\, t_1, \ldots, t_i : t) \mathrel{?\!\!-}_{\doteq} (t \doteq s)$$

($t \doteq s$ asserts the equality of t to the normal form s, which contains no instances of $t_1, \ldots, t_i$, see Definition 2.3.16). Since this chapter is entirely about Equational Logic, we drop the subscripts and write $\models$ for $\models_{\doteq}$ and $?\!\!-$ for $?\!\!-_{\doteq}$. The composed relation

$$(\mathbf{norm}\, t_1, \ldots, t_i : t) \mathrel{?\!\!-} \mathbf{T} \models (t \doteq s)$$

($t \doteq s$ is a semantically correct answer to the question $(\mathbf{norm}\, t_1, \ldots, t_i : t)$ for knowledge $\mathbf{T}$, see Definition 2.2.2) means that s is a normal form—a term containing no instances of $t_1, \ldots, t_i$—whose equality to t is a semantic consequence of the equations in $\mathbf{T}$. Equational logic programming languages in use today all take sets $\mathbf{T}$ of equations, prohibited forms $t_1, \ldots, t_i$, and terms t to normalize, and they compute normal forms s satisfying the relation above.

Section 2.3.5 of the 'Introduction ...' chapter explains how different equational languages variously determine $\mathbf{T}$, $t_1, \ldots, t_i$, and t from the language design, the program being executed, and the input. An alternate style of equational logic programming, using questions of the form $(\mathbf{solve}\, x_1, \ldots, x_i : t_1 \doteq t_2)$ that ask for substitutions for $x_1, \ldots, x_i$ solving the equation $(t_1 \doteq t_2)$, is very attractive for its expressive power, but much harder to implement efficiently (see Section 7.2).

There is a lot of terminological confusion about Equational Logic Programming. First, many in the Prolog community use 'logic' to mean the First-Order Predicate Calculus (FOPC), while I stick closer to the dictionary meaning of logic, in which FOPC is one of an infinity of possible logical systems. Those who identify logic with FOPC often use the phrase 'equational logic programming' to mean some sort of extension of Prolog using equations, such as logic programming in FOPC with equality. In this chapter, 'equational logic programming' means the logic programming of pure equational logic, as described in the chapter 'Introduction: Logic and Logic Programming Languages.'

A second source of confusion is that many equational logic programming languages have been invented under different labels. Lisp [McC60], APL [Ive62], Red languages [Bac74], Functional Programming languages [Bac78, Hud92], many dataflow languages [AW85, PA85, PA86], and languages for algebraic specification of abstract datatypes [FGJM85, GH78, Wan76] are all forms of equational logic programming languages, although they are seldom referred to as such. This chapter focusses on a generic notion of equational logic programming, rather than surveying particular languages.

1.2 Motivation for Programming with Equations

From a programmer's point of view, an equational logic programming language is the same thing as a functional programming language[Bac78]. The advantages of functional programming languages are discussed in [Hud89, BW88, FH88]—equational logic programming languages offer essentially the same advantages to the programmer. Functional programming and equational logic programming are different views of programming, which provide different ways of designing and describing a language, but they yield essentially the same class of possible languages. The different styles of design and description, while they allow the same range of possibilities, influence the sense of naturalness of different languages, and therefore the relative importance of certain features to the designer and implementer. The most important impact of the equational logic programming view on language design is the strong motivation that it gives to implement *lazy,* or *demand-driven,* computation.

In the conventional view of functional programming, computation is the *evaluation* of an input term in a *unique* model associated with the programming language. This view makes it very natural to evaluate a term of the form $f(s_1,\ldots,s_n)$ by first evaluating all of the arguments s_i, and then applying the function denoted by f to the values of the arguments. If the attempt to evaluate one of the arguments leads to infinite computation, then the value of that argument in the model is said to be an object called 'undefined' (the word is used here as a *noun*, although the dictionary recognizes it only as an adjective), and typically denoted by the symbol $\perp$. But, since the value $\perp$ is indicated by the behavior of infinite computation, there is no chance to actually apply the function denoted by f to it, so that every function is forced to map $\perp$ to $\perp$. Such functions are called *strict* functions.

Early functional programming languages required all primitive functions to be strict, except for the conditional function $cond$. The normal way to evaluate a term of the form $cond(s,t,u)$ is to evaluate s, then use its value to determine which of t or u to evaluate, omitting the other. The function denoted by $cond$ is thus not strict, since for example the value of $cond(true, 0, \perp)$ is 0 rather than $\perp$. Only Backus seems to have been annoyed by the inconsistency between the nonstrictness of the conditional function and the strictness of all other primitives. He proposed a strict conditional, recovering the selective behavior of the nonstrict conditional through a higher-order coding trick [Bac78]. In effect, he took advantage of the nearly universal unconscious acceptance of a nonstrict interpretation of function *application,* even when the function to be applied is strict.

In the equational logic programming view, computation is the derivation of an equivalent normal form for an input term using the information given by a set of equations describing the symbols of the programming language. The equivalence of input to output holds in *all* of the infinitely many models of those equations. This view makes it very natural to apply equations involving f to derive an equivalent form for $f(s_1,\ldots,s_n)$ at any time, possibly before all possible derivation has been performed on the arguments s_i. The natural desire for *completeness* of an implementation requires that infinite computation be avoided whenever possible. Notice that the equational logic programming view does not *assign* a value $\perp$ denoting 'undefined' (the noun) to a term with infinite computational behavior. In fact, in each individual model all functions are total. Rather, we might observe that a term is *undefined* (the word is now an adjective, as approved by the dictionary) if there is no equivalent term suitable for output, although each model of the given equations assigns it some value. So, equational logic programming leads naturally to computational behavior that is not strict—in fact, a logically complete implementation of equational logic programming must make functions as unstrict as possible. The preference for nonstrictness comes from regarding undefinedness as our inability to discover the value of a function, rather than the inherent lack of a semantic value.

The contrast between strict and nonstrict treatments of functions is best understood by comparing the conventional implementation of *cond*, *true* and *false* to that of *cons*, *car* and *cdr* in Lisp.

Example 1.2.1. The following equations define the relationship between *cond*, *true*, and *false*:

$$\mathbf{T}_{cond} = \{(cond(true, x, y) \doteq x),\ (cond(false, x, y) \doteq y)\}$$

Similarly the following equations define the relationship between *cons*, *car*, and *cdr*:

$$\mathbf{T}_{cons} = \{(car(cons(x, y)) \doteq x),\ (cdr(cons(x, y)) \doteq y)\}$$

These equations were given, without explicit restriction, in the earliest published definition of Lisp [McC60].

Notice the formal similarity between $\mathbf{T}_{cond}$ and $\mathbf{T}_{cons}$ in Example 1.2.1 above. In both cases, two equations provide a way to select one of the two subterms denoted by the variables x and y. In $\mathbf{T}_{cond}$, the selection is determined by the first argument to *cond*, in $\mathbf{T}_{cons}$ it is determined by the function symbol applied to the term headed by *cons*. Yet in all early Lisp implementations *cons* is evaluated strictly, while *cond* is not. The equation $(car(cons(0, s)) \doteq 0)$ is a logical consequence of $\mathbf{T}_{cons}$, even when s leads to infinite computation, so a complete implementation of equational logic programming must *not* treat *cons* strictly.

In the Lisp and functional programming communities, nonstrict evaluation of functions other than the conditional is called *lazy evaluation*. The power of lazy evaluation as a programming tool is discussed in [FW76, HM76, Hen80, Hud89, BW88, FH88, O'D85]. Lazy evaluation is *demand-driven*—computation is performed only as required to satisfy demands for output. So, the programmer may define large, and even infinite, data structures as intermediate values, and depend on the language implementation to compute only the relevant parts of those structures. In particular, lazily computed lists behave as *streams* [Kar81, HS88], allowing a straightforward encoding of pipelined coroutines in a functional style.

Many modern implementations of functional programming languages offer some degree of lazy evaluation, and a few are now uniformly lazy. But, in the functional programming view, lazy evaluation is an optional added feature to make programming languages more powerful. The basic denotational semantic approach to functional programming makes strictness very natural to describe, while denotational semantics for lazy evaluation seems to require rather sophisticated use of domain theory to construct models with special values representing all of the relevant nonterminating and partially-terminating behaviors of terms [Win93]. In the equational logic programming view, lazy evaluation is required for logical completeness, and strict evaluation is an arbitrary restriction on derivations that prevents certain answers from being found.

The functional and equational views also diverge in their treatments of certain terms that are viewed as pathological. From the functional programming view, pathological terms seem to require specialized logical techniques treating errors as values, and even new types of models called *error algebras* [Gog77]. For example, in a language with stacks, the term $pop(empty)$ is generally given a value which is a token denoting the erroneous attempt to pop an empty stack. Given a set of equations, equational logic programming provides a conceptual framework, based on well-understood traditional concepts from mathematical logic, for prescribing completely the computational behavior of terms. The judgement that a particular term is pathological is left to the consumer of that answer, which might be a human reader or another program. For example, the term $pop(empty)$ need not be evaluated to an error token: it may be output as a normal form, and easily recognized as a pathological case by the consumer. Or, an explicit equation $pop(empty) \doteq e$ may be added to the program, where e gives as much or as little detailed information about the particular error as desired.

So, for the programmer there is nothing to choose between lazy functional programming and equational logic programming—these are two styles for describing the same programming languages, rather than two different classes of programming languages. To the language designer or implementor, the functional programming view provides a connection to a large body of previous work, and offers some sophisticated tools for the thorough description of the processing of erroneous programs and the use of varying degrees of strictness or laziness. The equational logic programming view offers a deeper explanation of the logical content of computations, a way of defining correctness of the computation of answers independently of the classification of programs as correct or erroneous, and a strong motivation for uniformly lazy evaluation. It also connects equational/functional programming to other sorts of logic programming in a coherent way, which may prove useful to future designs that integrate equational/functional programming with other styles.

1.3 Outline of the Chapter

The next part of this chapter is primarily concerned with problems in the implementation of equational logic programming and some interesting variants of it. Those problems arise at four very different levels of abstraction—logic, strategy, algorithm, and code. At the level of pure logic, Section 2 discusses two different formal systems of proof for Equational Logic—inferential proof and term rewriting proof—and argues that the latter is logically weaker in general, but more likely to provide efficient computation for typical equational programs. The *confluence property* of sets of equations is introduced, and shown to be a useful way of guaranteeing that term rewriting proof can succeed. Next, Section 3 treats high-level strategic questions in the efficient search for a term rewriting proof to answer a given question. The crucial problem is to choose the next rewriting step out of a number of possibilities, so as to guarantee that all correct answers are found, and to avoid unnecessary steps. Then, Section 4 discusses the design of efficient algorithms and data structures for finding and choosing rewriting steps, and for representing the results of rewriting. Section 5 contains a brief description of the conventional machine code that a compiler can generate based on these algorithms and data structures. Section 6 discusses briefly some of the problems involved in parallel implementation of equational logic programming. Finally, Section 7 treats several possible extensions to the functionality of equational logic programming and the problems that arise in their semantics and implementation.

2 Proof Systems for Equational Logic

The basic idea in implementations of equational logic programming is to search for a proof that provides a correct answer to a given question. The basic idea behind proofs in Equational Logic is that the equation $t_1 \doteq t_2$ allows t_1 and t_2 to be used interchangeably in other formulae. As in Definition 3.1.1, of the chapter 'Introduction: Logic and Logic Programming Languages,' $\mathbf{T} \mid D - F$ means that D is a correct proof of the formula F from hypotheses in $\mathbf{T}$. $\mathbf{T} \vdash F$ means that there exists a proof of F from hypotheses in $\mathbf{T}$. In this chapter, subscripts on the generic symbols $\mid -$ and $\vdash$ are omitted whenever the particular proof system is clear from the context.

In Sections 2.1 and 2.2, we consider two different styles of equational proof. Inferential proofs derive equations step by step from other equations. Term rewriting proofs use equations to transform a given term into a provably equivalent term by substituting equals for equals.

2.1 Inferential Proofs

In order to explore a variety of approaches to proving equations, we first define generic concepts of rules of inference and proofs using rules, and then consider the power of various sets of rules.

Definition 2.1.1. Let the set $\mathbf{V}$ of variables, the sets $\mathbf{Fun}_i$ of i-ary function symbols, and the set $\mathbf{T}_P$ of terms, be the same as in Definition 2.3.1 of the 'Introduction …' chapter Section 2.3, and let the set of *equational formulae,* or simply *equations,* be $\mathbf{F}_{\doteq} = \{t_1 \doteq t_2 : t_1, t_2 \in \mathbf{T}_P\}$, as in Definition 2.3.13 in Section 2.3.4 of the chapter 'Introduction: Logic and Logic Programming Languages.'

An *equational rule of inference* is a binary relation

$$\mathcal{R} \subseteq \mathbf{2}^{\mathbf{F}_{\doteq}} \times \mathbf{F}_{\doteq}$$

When $\mathbf{T}\ \mathcal{R}\ F$, we say that F *follows from* $\mathbf{T}$ by rule $\mathcal{R}$. Members of $\mathbf{T}$ are called *hypotheses* to the application of the rule, and F is the *conclusion.* When $\emptyset\ \mathcal{R}\ F$, we call F a *postulate.* (It is popular now to call a postulated formula F an *axiom,* although the dictionary says that an axiom must be self-evident, not just postulated.) Rules of inference are usually presented in the form

$$\frac{H_1, \ldots, H_m}{C}$$

Where $H_1 \ldots, H_m$ are schematic descriptions of the hypotheses, and C is a schematic description of the conclusion of an arbitrary application of the rule. Notice that the union of rules of inference is itself a rule.

The set of *inferential equational proofs* is $\mathbf{P}_{\doteq} = \mathbf{F}_{\doteq}^{+}$, the set of nonempty finite sequences of equations. Given a rule of inference $\mathcal{R}$, the proof relation

$$\mid\!-_{\mathcal{R}} \subseteq \mathbf{2}^{\mathbf{F}_{\doteq}} \times \mathbf{P}_{\doteq} \times \mathbf{F}_{\doteq}$$

is defined by

$$\begin{gathered}\mathbf{T} \mid \langle F_0, \ldots, F_m \rangle -_{\mathcal{R}} F \text{ if and only if } F_m = F \text{ and,}\\ \text{for all } i \leq m, \text{ one of the following cases holds:}\end{gathered}$$

1. $F_i \in \mathbf{T}$
2. There exist $j_1, \ldots, j_n < i$ such that $\{F_{j_1}, \ldots, F_{j_n}\}\ \mathcal{R}\ F_i$

So, a proof of F from hypotheses in $\mathbf{T}$ using rule $\mathcal{R}$ is a sequence of equations, each one of which is either a hypothesis, or it follows from previous equations by the rule $\mathcal{R}$. The following are popular rules of inference for proofs in equational logic.

Definition 2.1.2.

$$\textit{Reflexive}\ \frac{}{s \doteq s}$$

$$\textit{Symmetric}\ \frac{s \doteq t}{t \doteq s}$$

$$\textit{Transitive}\ \frac{r \doteq s,\ s \doteq t}{r \doteq t}$$

$$\textit{Instantiation}\ \frac{s \doteq t}{s[r/x] \doteq t[r/x]}$$

$$\textit{Substitution}\ \frac{s \doteq t}{r[s/x] \doteq r[t/x]}$$

$$\textit{Congruence}\ \frac{s_1 \doteq t_1, \ldots, s_m \doteq t_m}{f(s_1, \ldots, s_m) \doteq f(t_1, \ldots, t_m)}$$

Now, when $\mathcal{R}$ is the union of any of the rules presented above, $\langle \mathbf{F}_{\doteq}, \mathbf{P}_{\doteq}, \mid\!-_{\mathcal{R}} \rangle$ is a compact proof system (Definition 2.1.3, Section 2.1).

The rules above are somewhat redundant. Every proof system using a subset of these rules is sound, and those using the Reflexive, Symmetric, Transitive and Instantiation rules, and at least one of Substitution and Congruence, are also complete.

Proposition 2.1.3. *Let $\mathcal{R}$ be the union of any of the rules in Definition 2.1.2. Then $\langle \boldsymbol{F}_{\doteq}, \boldsymbol{P}_{\doteq}, \mid\!-_{\mathcal{R}} \rangle$ is a sound proof system for the standard semantic system of Definition 2.3.14 Section 2.3.4 of the chapter 'Introduction: Logic and Logic Programming Languages.' That is, $\boldsymbol{T} \vdash_{\mathcal{R}} (t_1 \doteq t_2)$ implies $\boldsymbol{T} \models (t_1 \doteq t_2)$.*

The proof of soundness is an elementary induction on the number of steps in a formal equational proof, using the fact that each of the rules of inference proposed above preserves truth.

Proposition 2.1.4. *Let $\mathcal{R}$ be the union of the Reflexive, Symmetric, Transitive, and Instantiation rules, and at least one of the Substitution and Congruence rules. Then $\langle \boldsymbol{F}_{\doteq}, \boldsymbol{P}_{\doteq}, \mid\!-_{\mathcal{R}} \rangle$ is a complete proof system. That is, $\boldsymbol{T} \models (t_1 \doteq t_2)$ implies $\boldsymbol{T} \vdash_{\mathcal{R}} (t_1 \doteq t_2)$.*

To prove completeness, we construct for each set $\boldsymbol{T}$ of equations, a term model $M_{\boldsymbol{T}}$ such that $\mathbf{Theory}(\{M_{\boldsymbol{T}}\})$ contains exactly the semantic consequences of $\boldsymbol{T}$. For each term $t \in \boldsymbol{T}_P$,

$$|t|_{\boldsymbol{T}} = \{s : \boldsymbol{T} \vdash_{\mathcal{R}} (s \doteq t)\}$$

Because $\mathcal{R}$ includes the Reflexive, Symmetric, and Transitive rules, provable equality is an equivalence relation on terms, and $|t|_{\boldsymbol{T}}$ is the equivalence class containing t. Now, construct the model

$$M_{\boldsymbol{T}} = \langle U_{\boldsymbol{T}}, \tau_{\boldsymbol{T}} \rangle$$

whose universe is

$$U_{\boldsymbol{T}} = \{|t|_{\boldsymbol{T}} : t \in \boldsymbol{T}_P\}$$

and whose function assignment is defined by

$$\tau_{\boldsymbol{T}}(f)(|t_1|_{\boldsymbol{T}}, \ldots, |t_i|_{\boldsymbol{T}}) = |f(t_1, \ldots, t_i)|_{\boldsymbol{T}}$$

Either of the rules Substitution and Congruence is sufficient to guarantee that $\tau_{\boldsymbol{T}}$ is well defined. Finally, the Instantiation rule guarantees that $\boldsymbol{T} \vdash (s \doteq t)$ if and only if $\tau_{\boldsymbol{T}\nu}(s) = \tau_{\boldsymbol{T}\nu}(t)$ for all variable assignments ν, which by Definition 2.3.14 is equivalent to $\boldsymbol{T} \models (s \doteq t)$.

Notice that each inference by the Congruence rule is derivable by k applications of the Substitution rule, combined by the Transitive rule. In effect, Congruence is just a special form of multiple simultaneous substitution. Similarly, each inference by the Substitution rule is derivable by repeated applications of the Congruence rule and additional instances of the Reflexive rule (this can be proved easily by induction on the structure of the term r on which substitution is performed in the Substitution rule).

In the rest of this chapter, the symbols $|\!-_{inf}$ and $\vdash_{inf}$ refer to a sound and complete system of inferential equational proof, when the precise rules of inference are not important.

2.2 Term Rewriting Proofs

The most commonly used methods for answering normal form questions ($\mathbf{norm}\, t_1, \ldots, t_i : t$) all involve replacing subterms by equal subterms, using the Substitution rule, to transform the term t into an equivalent normal form. Substitution of subterms according to given rules is called *term rewriting*, and is an interesting topic even when the rewriting rules are not given by equations (see the chapter 'Equational Reasoning and Term Rewriting Systems' in Volume 1). In this chapter, we are concerned only with the use of term rewriting to generate equational proofs—this technique is also called *demodulation* [Lov78] in the automated deduction literature.

Definition 2.2.1. Let $\mathbf{T} = \{l_1 \doteq r_1, \ldots, l_n \doteq r_n\}$ be a set of equations. Recall that an *instance* of a formula or term is the result of substituting terms for variables (Definition 2.3.5 in Section 2.3.1 of the chapter 'Introduction: Logic and Logic Programming Languages').

A term s_1 *rewrites to* s_2 *by* $\mathbf{T}$ (written $s_1 \xrightarrow{\mathbf{T}} s_2$) if and only if there is a term t, a variable x with exactly one occurrence in t, and an instance $l_i' \doteq r_i'$ of an equation $l_i \doteq r_i$ in $\mathbf{T}$, such that $s_1 = t[l_i'/x]$ and $s_2 = t[r_i'/x]$. That is, s_2 results from finding exactly one instance of a left-hand side of an equation in $\mathbf{T}$ occurring as a subterm of s_1, and replacing it with the corresponding right-hand side instance.

A *term rewriting sequence* for $\mathbf{T}$ is a nonempty finite or infinite sequence $\langle u_0, u_1, \ldots \rangle$ such that, for each i, $u_i \xrightarrow{\mathbf{T}} u_{i+1}$.

Term rewriting sequences formalize the natural intuitive process of replacing equals by equals to transform a term. A term rewriting sequence may be viewed as a somewhat terse proof.

Definition 2.2.2. Let $\mathbf{T}_P^+$ be the set of nonempty finite sequences of terms in $\mathbf{T}_P$. The proof relation $\mid\!-_{tr}$ is defined by $\mathbf{T} \mid \langle u_0, \ldots, u_m \rangle -_{tr} (s \doteq t)$ if and only if $u_0 = s$, $u_m = t$, and for each $i < m$, $u_i \xrightarrow{\mathbf{T}} u_{i+1}$.

Then $\langle \mathbf{F}_{\doteq}, \mathbf{T}_P^+, \mid\!-_{tr} \rangle$, is a compact proof system, representing the *term rewriting* style of equational proof.

A term rewriting proof for $\mathbf{T}$ represents an inferential proof from hypotheses in $\mathbf{T}$ in a natural way.

Proposition 2.2.3. *If $\boldsymbol{T} \vdash_{tr} (s \doteq t)$, then $\boldsymbol{T} \vdash_{inf} (s \doteq t)$.*

Let $\langle u_0, \ldots, u_n \rangle$ be the term rewriting sequence such that

$$\boldsymbol{T} \mid \langle u_0, \ldots, u_n \rangle -_{tr} (s \doteq t)$$

In particular, $u_0 = s$ and $u_n = t$. The proof of the proposition is an elementary induction on n.

Basis: *For $n = 0$, $s = u_0 = u_n = t$, so $\boldsymbol{T} \mid \langle u_0 \doteq u_0 \rangle -_{inf} (s \doteq t)$, by the Reflexive rule.*

Induction: *For $n > 0$, since a nonempty prefix of a term rewriting proof is also a term rewriting proof, we have $\boldsymbol{T} \mid \langle u_0, \ldots, u_{n-1} \rangle -_{tr} (s \doteq u_{n-1})$. By the induction hypothesis, there is a D such that $\boldsymbol{T} \mid D -_{inf} (s \doteq u_{n-1})$. It is easy to extend D to D' so that $\boldsymbol{T} \mid D' - (s \doteq t)$, by adding the following steps:*

- *the appropriate equation from $\boldsymbol{T}$;*
- *a sequence of applications of the Instantiation rule to produce the appropriate instance of the equation above;*
- *one application of the Substitution rule to produce $u_{n-1} \doteq t$;*
- *one application of the Transitive rule to produce $s \doteq t$.*

Since inferential proof is sound, it follows that term rewriting proof is also sound.

Example 2.2.4. Let $\mathbf{T} = \{f(a, f(x, y)) \doteq f(y, x),\ g(x) \doteq x\}$.

$$\langle f(g(a), f(g(b), c)),\ f(a, f(g(b), c)),\ f(c, g(b)),\ f(c, b)\rangle$$

is a term rewriting proof of

$$f(g(a), f(g(b), c)) \doteq f(c, b)$$

from $\mathbf{T}$. The corresponding inferential proof from the induction in Proposition 2.2.3 is given below. Line numbers are added on the left, and rules cited on the right, for clarity: formally the proof is just the sequence of equations. The key occurrences of the terms in the term rewriting sequence above are boxed to show the correspondence.

$$\begin{array}{rrcll r}
& \langle & & & & \\
1 & f(g(a), f(g(b), c)) & \doteq & \boxed{f(g(a), f(g(b), c))} & , & \mathit{Reflexive} \\
2 & g(x) & \doteq & x & , & \mathit{Hypothesis} \\
3 & g(a) & \doteq & a & , & \mathit{Instantiation}, 2 \\
4 & f(g(a), f(g(b), c)) & \doteq & f(a, f(g(b), c)) & , & \mathit{Substitution}, 3 \\
5 & f(g(a), f(g(b), c)) & \doteq & \boxed{f(a, f(g(b), c))} & , & \mathit{Transitive}, 1, 3 \\
7 & f(a, f(x, y)) & \doteq & f(y, x) & , & \mathit{Hypothesis} \\
8 & f(a, f(g(b), y)) & \doteq & f(y, g(b)) & , & \mathit{Instantiation}, 7 \\
9 & f(a, f(g(b), c)) & \doteq & f(c, g(b)) & , & \mathit{Instantiation}, 8 \\
10 & f(a, f(g(b), c)) & \doteq & f(c, g(b)) & , & \mathit{Substitution}, 9 \\
11 & f(g(a), f(g(b), c)) & \doteq & \boxed{f(c, g(b))} & , & \mathit{Transitive}, 1, 10 \\
12 & g(x) & \doteq & x & , & \mathit{Hypothesis} \\
13 & g(b) & \doteq & b & , & \mathit{Instantiation}, 12 \\
14 & f(c, g(b)) & \doteq & f(c, b) & , & \mathit{Substitution}, 13 \\
15 & f(g(a), f(g(b), c)) & \doteq & \boxed{f(c, b)} & & \mathit{Transitive}, 1, 14 \\
& & & & \rangle &
\end{array}$$

Steps 5, 10, and 12 above are redundant (they reproduce the results already obtained in steps 4, 9, 2), but the systematic procedure in the induction of Proposition 2.2.3 includes them for uniformity.

So, a term rewriting proof is a convenient and natural shorthand for an inferential proof.

Not every inferential proof corresponds to a term rewriting proof. First, the proofs corresponding to term rewriting sequences do not use the Symmetric rule. This represents a serious incompleteness in term rewriting proof. Section 2.3 shows how restrictions on equational hypotheses can avoid the need for the Symmetric rule, and render term rewriting complete for answering certain normal form questions.

Example 2.2.5. Let $\mathbf{T} = \{a \doteq b,\ c \doteq b,\ c \doteq d\}$. $\mathbf{T} \models (a \doteq d)$, and $\mathbf{T} \vdash_{inf} (a \doteq d)$, by one application of the Symmetric rule and two applications of the Transitive rule. But, there is no term rewriting sequence from a to d, nor from d to a, nor from a and d to a common form equal to both.

Second, term rewriting proofs limit the order in which the Instantiation, Substitution, and Transitive rules are applied. This second limitation does not affect the deductive power of the proof system.

Proposition 2.2.6. *Let $\boldsymbol{T} = \{l_1 \doteq r_1, \ldots, l_n \doteq r_n\}$ be a set of equations. Let $\boldsymbol{T}^R = \{r_1 \doteq l_1, \ldots, r_n \doteq l_n\}$. $\boldsymbol{T}^R$ is the same as $\boldsymbol{T}$ except that the left and right sides of equations are interchanged—equivalently, $\boldsymbol{T}^R$ contains the results of applying the Symmetric rule to the equations in $\boldsymbol{T}$.*

For all equations $(s \doteq t)$, if $\boldsymbol{T} \vdash_{inf} (s \doteq t)$ (equivalently, if $\boldsymbol{T} \models (s \doteq t)$) then $\boldsymbol{T} \cup \boldsymbol{T}^R \vdash_{tr} (s \doteq t)$.

The proof of the proposition, given in more detail in [O'D77], works by permuting the steps in an arbitrary inferential proof of $s \doteq t$ into the form:

1. *hypotheses;*
2. *applications of the Symmetric rule;*
3. *applications of the Instantiation rule;*
4. *applications of the Substitution rule;*
5. *applications of the Transitive rule.*

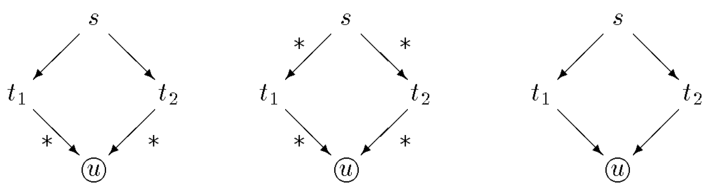
The reflexive rule is only needed in the degenerate case when $s = t$ (s and t are the same term). In this form, it is easy to represent each of the applications of the Transitive rule as concatenating two term rewriting sequences. The crucial quality of the permuted form of the proof is that all uses of the Instantiation rule come before any use of the Transitive and Substitution rules.

The implementor of a logic programming system often faces a trade-off between the cost of an individual proof, and the cost of the search for that proof. The discipline of term rewriting can be very advantageous in reducing the number of possible steps to consider in the search for a proof to answer a question, but it increases the lengths of proofs in some cases. Section 4.3.3 shows how clever uses of Instantiation sometimes reduce the length of a proof substantially compared to term rewriting proofs. Efficient implementations of programming languages have not yet succeeded in controlling the costs of search for a proof with the more sophisticated approaches to Instantiation, so term rewriting is the basis for almost all implementations.

2.3 The Confluence Property and the Completeness of Term Rewriting

Term rewriting is often much more efficient than an undisciplined search for an equational proof. But, for general sets $\mathbf{T}$ of equational hypotheses, term rewriting is not complete, due to its failure to apply the Symmetric rule. It is tempting, then, to use each equation in both directions, and take advantage of the completeness result of Proposition 2.2.6. Unfortunately, known techniques for efficient term rewriting typically fail or become inefficient when presented with the reversed forms of equations. So, we find special restrictions on equations that imply the completeness of term rewriting for the answering of particular normal form questions. The *confluence property*, also called the *Church-Rosser property*, provides the key to such restrictions.

Definition 2.3.1. Let $\rightarrow$ be a binary relation, and $\rightarrow^*$ be its reflexive-transitive closure. $\rightarrow$ is *confluent* if and only if, for all s, t_1, t_2 in its domain such that $s \rightarrow^* t_1$ and $s \rightarrow^* t_2$, there exists a u such that $t_1 \rightarrow^* u$ and $t_2 \rightarrow^* u$ (see Figure 1 B)



A. Local confluence B. Confluence C. One-step confluence

The circle around u indicates that it is existentially quantified, the uncircled s, t_1, t_2 are universally quantified.

Fig. 1. Confluence and related properties.

Two similar properties that are very important in the literature are *local confluence*, which is weaker than confluence, and *one-step confluence*, which is stronger than confluence.

Definition 2.3.2 ([New42]). Let $\rightarrow$ be a binary relation, and $\rightarrow^*$ be its reflexive-transitive closure. $\rightarrow$ is *locally confluent* if and only if, for all s, t_1, t_2 in its domain such that $s \rightarrow t_1$ and $s \rightarrow t_2$, there exists a u such that $t_1 \rightarrow^* u$ and $t_2 \rightarrow^* u$ (see Figure 1 A).

While confluence guarantees that divergent term rewriting sequences may always be rewritten further to a common form, local confluence guarantees this only for single step term rewriting sequences.

Definition 2.3.3 ([New42]). Let $\rightarrow$ be a binary relation. $\rightarrow$ is *locally confluent* if and only if, for all s, t_1, t_2 in its domain such that $s \rightarrow t_1$ and $s \rightarrow t_2$, there exists a u such that $t_1 \rightarrow u$ and $t_2 \rightarrow u$ (see Figure 1 C).

While confluence guarantees that divergent term rewriting sequences may always be rewritten further to a common form, one-step confluence guarantees that for single step divergences, there is a single-step convergence.

Proposition 2.3.4. *One-step confluence implies confluence implies local confluence.*

The first implication is a straightforward induction on the number of steps in the diverging rewrite sequences. The second is trivial.

2.3.1 *Consequences of Confluence.*

When the term rewriting relation $\xrightarrow{\mathbf{T}}$ for a set $\mathbf{T}$ of equations has the confluence property, term rewriting is sufficient for deriving all logical consequences of $\mathbf{T}$, in the sense that $\mathbf{T} \models (s \doteq t)$ implies that s and t rewrite to some common form u.

Proposition 2.3.5 ([CF58]). *Let $\boldsymbol{T}$ be a set of equations, and let $\xrightarrow{\boldsymbol{T}}$ be the term rewriting relation for $\boldsymbol{T}$ (Definition 2.2.1). If $\xrightarrow{\boldsymbol{T}}$ is confluent, then for all terms s and t such that $\boldsymbol{T} \models (s \doteq t)$, there is a term u such that $\boldsymbol{T} \vdash_{tr} (s \doteq u)$ and $\boldsymbol{T} \vdash_{tr} (t \doteq u)$.*

The proof of the proposition is an elementary induction on the length of an inferential proof D such that $\boldsymbol{T} \mid D -_{inf} (s \doteq t)$.

So, confluent term rewriting is nearly complete, in the sense that every logical consequence $s \doteq t$ of a set of equations $\mathbf{T}$ may be derived by choosing an appropriate term u, and finding two term rewriting proofs and a trivial inferential proof as follows:

1. $\mathbf{T} \vdash_{tr} (s \doteq u)$
2. $\mathbf{T} \vdash_{tr} (t \doteq u)$
3. $\{s \doteq u,\ t \doteq u\} \vdash_{inf} (s \doteq t)$ trivially, by one application of Symmetry and one application of Transitivity.

The near-completeness of confluent term rewriting leads to its use in theorem proving [KB70, Lov78]. For equational logic programming, term rewriting can answer all normal form queries in a confluent system, when the prohibited terms in normal forms are all the left-hand sides of equations.

Proposition 2.3.6. *Let $\boldsymbol{T} = \{l_1 \doteq r_1, \ldots, l_m \doteq r_m\}$ be a set of equations, with confluent term rewriting relation $\stackrel{\boldsymbol{T}}{\rightarrow}$, and let t be any term. If*

$$(\mathbf{norm}\, l_1, \ldots, l_m : t) \mathrel{?\!\!-} \boldsymbol{T} \models (t \doteq s)$$

then

$$(\mathbf{norm}\, l_1, \ldots, l_m : t) \mathrel{?\!\!-} \boldsymbol{T} \vdash_{tr} (t \doteq s)$$

The proof is elementary. By confluence, t and s rewrite to a common form u. Since s is a normal form, it is not rewritable, and must be the same as u.

So, for equations $\mathbf{T}$ with confluent rewriting relation, term rewriting based on $\mathbf{T}$ is sufficient for answering all queries requesting normal forms that prohibit left-hand sides of equations in $\mathbf{T}$. From now on, a *normal form* will mean a normal form for the left-hand sides of whatever set of equations we are discussing (see Definition 2.3.16 in the chapter 'Introduction: Logic and Logic Programming Languages' for the general concept of *normal form*).

The most famous consequence of the confluence property is uniqueness of normal forms.

Proposition 2.3.7. *Let $\boldsymbol{T} = \{l_1 \doteq r_1, \ldots, l_m \doteq r_m\}$ be a set of equations, with confluent term rewriting relation $\stackrel{\boldsymbol{T}}{\rightarrow}$. If*

$$(\mathbf{norm}\, l_1, \ldots, l_m : t) \mathrel{?\!\!-} \boldsymbol{T} \models (t \doteq s_1)$$

and

$$(\mathbf{norm}\, l_1, \ldots, l_m : t) \mathrel{?\!\!-} \boldsymbol{T} \models (t \doteq s_2)$$

then $s_1 = s_2$ (s_1 and s_2 are the same term).

The proof is elementary. By confluence, s_1 and s_2 rewrite to a common form u. Since s_1 and s_2 are normal forms, they are not rewritable, and must be the same as u.

So, equational logic programs using confluent systems of equations have uniquely defined outputs. This is an interesting property to note, but it is not essential to the logic programming enterprise—logic programs in FOPC are allowed to have indeterminate answers (Section 2.3.1 of the 'Introduction' chapter), and this freedom is often seen as an advantage. In efficient equational logic programming, confluence is required for the completeness of term rewriting, and uniqueness of answers is an accidental side-effect that may be considered beneficial or annoying in different applications. Confluence, in effect, guarantees that the order of applying rewrite steps cannot affect the normal form. In Section 3 we see that the order of application of rewrite rules *can* affect the efficiency with which a normal form is found, and in some cases whether or not the unique normal form is found at all.

2.3.2 Testing for Confluence.

Proposition 2.3.8. *Confluence is an undecidable property of finite sets of equations.*

The proof is straightforward. Given an arbitrary Turing Machine $\mathcal{M}$, modify $\mathcal{M}$ so that, if it halts, it does so in the special configuration I_f. Encode configurations (instantaneous descriptions) of $\mathcal{M}$ as terms (just let the tape and state symbols be unary function symbols), and provide rewriting rules to simulate the computation of $\mathcal{M}$. So far, we have a system of equations in which an arbitrary encoding of an initial configuration I_0 rewrites to I_f if and only if $\mathcal{M}$ halts on I_0. Choose a new symbol a not used in encoded configurations, and add two more equations: $I_0 \doteq a$ and $I_f \doteq a$. The extended system is confluent if and only if $\mathcal{M}$ halts on I_0.

For practical purposes in programming language implementations, we need a *sufficient* condition for confluence that is efficient to test.

Orthogonality. A particularly useful sort of condition for guaranteeing confluence is *orthogonality,* also called *regularity* (but not connected in any sensible way to the regular languages). Orthogonality is a set of restrictions on rewrite rules insuring that they do not interfere with one another in certain pathological ways. We consider three versions of orthogonality. *Rewrite-orthogonality* insures that the rewrites performed by the rules do not interfere, while the stronger condition of *rule-orthogonality* prohibits even the appearance of interference based on an inspection of the left-hand sides of the rules, and ignoring the right-hand sides. *Constructor-orthogonality* is an even stronger and simpler syntactic condition that guarantees rule-orthogonality. In other literature on term rewriting, 'orthogonality' and 'regularity' refer to the stronger form, rule-orthogonality.

Definition 2.3.9. Let $\mathbf{T} = \{l_1 \doteq r_1, \ldots, l_n \doteq r_n\}$ be a set of equations. $\mathbf{T}$ is *rewrite-orthogonal* if and only if the following conditions hold:

1. (Nontrivial) No left-hand side l_i of an equation $l_i \doteq r_i$ in $\mathbf{T}$ consists entirely of a variable.
2. (Rule-like) Every variable in the right-hand side r_i of an equation $l_i \doteq r_i$ in $\mathbf{T}$ occurs in the left-hand side l_i as well.
3. (Left-linear) No variable occurs more than once in the left-hand side l_i of an equation $l_i \doteq r_i$ in $\mathbf{T}$.
4. (Rewrite-Nonambiguous) Let l_i and l_j be left-hand sides of equations in $\mathbf{T}$, and let s be a term with a single occurrence of a new variable y (not occurring in any equation of $\mathbf{T}$). If

$$s[l_i[t_1, \ldots, t_m/x_1, \ldots, x_m]/y] = l_j[t'_1, \ldots, t'_n/x'_1, \ldots, x'_n]$$

then either s is an instance of l_j, or

$$s[r_i[t_1,\ldots,t_m/x_1,\ldots,x_m]/y] = r_j[t'_1,\ldots,t'_n/x'_1,\ldots,x'_n]$$

In clause 4 the nested substitution may be hard to read.

$$s[r_i[t_1,\ldots,t_m/x_1,\ldots,x_m]/y]$$

is the result of substituting $t_1,\ldots,t_m$ for $x_1,\ldots,x_m$ in r_i, to produce $r'_i = r_i[t_1,\ldots,t_m/x_1,\ldots,x_m]$, then substituting r'_i for y in s. Clause 4 is best understood by considering an example where it fails. The set of equations $\{f(g(v,w),x) \doteq a,\ g(h(y),z) \doteq b\}$ is rewrite-ambiguous because, in the term $f(g(h(c),d),e)$, there is an instance of $f(g(w,x))$ and an instance of $g(h(y),z)$, and the two instances share the symbol g. Furthermore, $f(g(h(c),d),e)$ rewrites to a using the first equation, and to a different result, $f(b,e)$, using the second equation.

Nontriviality and the Rule-like property are required in order for the interpretation of the equations as term rewriting rules to make much sense. Left-linearity is of practical importance because the application of a rule with repeated variables on the left-hand side requires a test for equality. Non-left-linear systems also fail to be confluent in rather subtle ways, as shown in Example 2.3.16 below. Rewrite-nonambiguity says that if two rewriting steps may be applied to the same term, then they are either completely independent (they apply to disjoint sets of symbols), or they are equivalent (they produce the same result). Example 2.3.16 below shows more cases of rewrite-ambiguity and its consequences.

One simple way to insure rewrite-nonambiguity is to prohibit all interference between left-hand-sides of rules.

Definition 2.3.10 ([Klo91],[Klo80]). Let $\mathbf{T} = \{l_1 \doteq r_1,\ldots,l_n \doteq r_n\}$ be a set of equations. $\mathbf{T}$ is *rule-orthogonal* if and only if $\mathbf{T}$ satisfies conditions 1–3 of Definition 2.3.9 above, and also

$4'$ (Rule-Nonambiguous) Let l_i and l_j be left-hand sides of equations in $\mathbf{T}$, and let s be a term with a single occurrence of a new variable y (not occurring in any equation of $\mathbf{T}$). If

$$s[l_i[t_1,\ldots,t_m/x_1,\ldots,x_m]/y] = l_j[t'_1,\ldots,t'_n/x'_1,\ldots,x'_n]$$

then either s is an instance of l_j, or $s = y$ and $i = j$.

Rule-nonambiguity says that if two rewriting steps may be applied to the same term, then they are either completely independent, or they are identical (the same rule applied at the same place). Notice that rule nonambiguity depends only on the left-hand sides of equations, not the right-hand sides. In fact, only the Rule-like condition of rule-orthogonality depends on right-hand sides.

Definition 2.3.11. Two systems of equations are *left-similar* if the multisets of left-hand sides of equations are the same, except for renaming of variables.

Proposition 2.3.12. *A set $\boldsymbol{T}$ of equations is rule-orthogonal if and only if*

- *$\boldsymbol{T}$ satisfies the rule-like restriction, and*
- *every rule-like set of equations left-similar to $\boldsymbol{T}$ is rewrite-orthogonal.*

That is, rule-orthogonality holds precisely when rewrite-orthogonality can be guaranteed by the forms of the left-hand sides alone, independently of the right-hand sides.

An even simpler way to insure rule-nonambiguity is to use a *constructor* system, in which symbols appearing leftmost in rules are not allowed to appear at other locations in left-hand sides.

Definition 2.3.13. Let $\mathbf{T} = \{l_1 \doteq r_1, \ldots, l_n \doteq r_n\}$ be a set of equations. $\mathbf{T}$ is *constructor-orthogonal* if and only if $\mathbf{T}$ satisfies conditions 1–3 of Definition 2.3.9 above, and the symbols of the system partition into two disjoint sets—the set $\mathbf{C}$ of *constructor symbols*, and the set $\mathbf{D}$ of *defined symbols*, satisfying

$4''$ (Symbol-Nonambiguous)

- Every left-hand side of an equation in $\mathbf{T}$ has the form $f(t_1, \ldots, t_m)$, where $f \in \mathbf{D}$ is a defined symbol, and $t_1, \ldots, t_m$ contain only variables and constructor symbols in $\mathbf{C}$.
- Let l_i and l_j be left-hand sides of equations in $\mathbf{T}$. If there exists a common instance s of l_i and l_j, then $i = j$.

In most of the term-rewriting literature, 'orthogonal' and 'regular' both mean *rule-orthogonal*. It is easy to see that constructor orthogonality implies rule-orthogonality, which implies rewrite-orthogonality. Most functional programming languages have restrictions equivalent or very similar to constructor-orthogonality.

Orthogonal systems of all varieties are confluent.

Proposition 2.3.14. *Let $\boldsymbol{T}$ be a constructor-, rule- or rewrite-orthogonal set of equations. Then the term rewriting relation $\xrightarrow{\boldsymbol{T}}$ is confluent.*

Let $\rightarrow$ be the rewrite relation that is to be proved confluent. The essential idea of these, and many other, proofs of confluence is to choose another *relation $\rightarrow'$ with the one-step confluence property (Definition 2.3.3, whose transitive closure is the same as the transitive closure of $\rightarrow$. Since confluence is defined entirely in terms of the transitive closure, $\rightarrow$ is confluent if and only if $\rightarrow'$ is confluent. $\rightarrow'$ is confluent because one-step confluence implies confluence. To prove confluence of orthogonal systems of equations, the appropriate $\rightarrow'$ allows simultaneous rewriting of any number of disjoint subterms.*

Theorem 10.1.3 of the chapter 'Equational Reasoning and Term-Rewriting Systems' in Section 10.1 of Volume 1 of this handbook is the rewrite-orthogonal portion of this proposition, which is also proved in [HL91, Klo91]. The proof for rewrite-orthogonal systems has never been published, but it is a straightforward generalization. [O'D77] proves a version intermediate between rule-orthogonality and rewrite-orthogonality.

In fact, for nontrivial, rule-like, left-linear systems, rule-nonambiguity captures precisely the cases of confluence that depend only on the left-hand sides of equations.

Proposition 2.3.15. *A nontrivial, rule-like, left-linear set $\boldsymbol{T}$ of equations is rule-nonambiguous if and only if, for every set of equations $\boldsymbol{T}'$ left-similar to $\boldsymbol{T}$, $\stackrel{\boldsymbol{T}'}{\rightarrow}$ is confluent.*

($\Rightarrow$) is a direct consequence of Propositions 2.3.14 and 2.3.12. ($\Leftarrow$) is straightforward. In a rule-ambiguous system, simply fill in each right-hand side with a different constant symbol, not appearing on any left-hand side, to get a nonconfluent system.

In the rest of this chapter, we use the term 'orthogonal' in assertions that hold for both rewrite- and rule-orthogonality. To get a general understanding of orthogonality, and its connection to confluence, it is best to consider examples of nonorthogonal systems and investigate why they are not confluent, as well as a few examples of systems that are not rule orthogonal, but are rewrite orthogonal, and therefore confluent.

Example 2.3.16. The first example, due to Klop [Klo80], shows the subtle way in which non-left-linear systems may fail to be confluent. Let

$$\mathbf{T}_1 = \{eq(x,x) \doteq true,\ f(x) \doteq eq(x, f(x)),\ a \doteq f(a)\}$$

eq represents an equality test, a very useful operation to define with a non-left-linear equation. Now

$$f(a) \stackrel{\mathbf{T}_1}{\rightarrow} eq(a, f(a)) \stackrel{\mathbf{T}_1}{\rightarrow} eq(f(a), f(a)) \stackrel{\mathbf{T}_1}{\rightarrow} true$$

and also

$$f(a) \stackrel{\mathbf{T}_1}{\rightarrow} f(f(a)) \stackrel{\mathbf{T}_1}{\rightarrow} f(eq(a, f(a))) \stackrel{\mathbf{T}_1}{\rightarrow} f(eq(f(a), f(a))) \stackrel{\mathbf{T}_1}{\rightarrow} f(true)$$

true is in normal form, and $f(true)$ rewrites infinitely as

$$f(true) \xrightarrow{\mathbf{T}_1} eq(true, f(true)) \xrightarrow{\mathbf{T}_1} eq(true, eq(true, f(true))) \xrightarrow{\mathbf{T}_1} \cdots$$

The system is not confluent, because the attempt to rewrite $f(true)$ to *true* yields an infinite regress with $f(true) \xrightarrow{\mathbf{T}_1} eq(true, f(true))$. Notice that $\xrightarrow{\mathbf{T}_1}$ has unique normal forms. The failure of confluence involves a term with a normal form, and an infinite term rewriting sequence from which that normal form cannot be reached. Non-left-linear systems that satisfy the other requirements of rule-orthogonality always have unique normal forms, even when they fail to be confluent [Che81]. I conjecture that this holds for rewrite-orthogonality as well.

A typical rewrite-ambiguous set of equations is

$$\mathbf{T}_2 = \{c(a, y) \doteq a,\ c(x, b) \doteq b\}$$

c represents a primitive sort of *nondeterministic choice* operator. $\mathbf{T}_2$ violates condition $(4')$ because

$$w[c(a, y)[b/y]/w] = c(a, b) = c(x, b)[a/x]$$

but

$$w[a[b/y]/w] = a \neq b = b[a/x]$$

$\xrightarrow{\mathbf{T}_2}$ is not confluent, as $c(a, b) \xrightarrow{\mathbf{T}_2} a$ by the first equation, and $c(a, b) \xrightarrow{\mathbf{T}_2} b$ by the second equation, but a and b are in normal form.

By contrast, consider the set

$$\mathbf{T}_{or+} = \{or(true, y) \doteq true,\ or(x, true) \doteq true\}$$

of equations defining the *positive parallel or* operator. Although $\mathbf{T}_{or+}$ is rule-ambiguous, it is *rewrite*-nonambiguous:

$$w[or(true, y)[true/y]/w] = or(true, true) = or(x, true)[true/x]$$

and w is not an instance of $or(x, true)$, but the corresponding right-hand sides are both *true*:

$$w[true[true/y]/w] = true = true[true/x]$$

$\mathbf{T}_{or+}$ is rewrite-orthogonal, so $\xrightarrow{\mathbf{T}_{or+}}$ is confluent.

A more subtle example of a rewrite-orthogonal set of equations that is rule-ambiguous is the *negative parallel or:*

$$\mathbf{T}_{or-} = \{or(\mathit{false}, y) \doteq y,\ or(x, \mathit{false}) \doteq x\}$$

Although

$$w[or(\mathit{false}, y)[\mathit{false}/y]/w] = or(\mathit{false}, \mathit{false}) = or(x, \mathit{false})[\mathit{false}/x]$$

and w is not an instance of $or(x, \mathit{false})$, the substitution above unifies the corresponding right-hand sides as well:

$$y[\mathit{false}/y] = \mathit{false} = x[\mathit{false}/x]$$

$\mathbf{T}_{or-}$ is rewrite-orthogonal, so $\xrightarrow{\mathbf{T}_{or-}}$ is confluent.

Another type of rewrite-ambiguous set of equations is

$$\mathbf{T}_3 = \{f(g(x, y)) \doteq g(f(x), f(y)),\ g(i, z) \doteq z\}$$

These equations express the fact that f is a *homomorphism* for g (i.e., f *distributes* over g), and that i is a *left identity* for g. The left-hand sides of the two equations overlap in $f(g(i, z))$, with the symbol g participating in instances of the left-hand sides of both equations. Condition (4) is violated, because

$$f(w)[g(i, z)[y/z]/w] = f(g(i, y)) = f(g(x, y))[i/x]$$

but $f(w)$ is not an instance of $f(g(x, y))$. $\xrightarrow{\mathbf{T}_3}$ is not confluent, as $f(g(i, i)) \xrightarrow{\mathbf{T}_3} g(f(i), f(i))$ by the first equation, and $f(g(i, i)) \xrightarrow{\mathbf{T}_3} f(i)$ by the second equation, but both $g(f(i), f(i))$ and $f(i)$ are in normal form. While the previous examples of ambiguity involved two rules applying to precisely the same term, the ambiguity in $\mathbf{T}_3$ comes from two overlapping applications of rules to a term and one of its subterms. Some definitions of orthogonality/regularity treat these two forms of ambiguity separately.

By contrast, consider the set

$$\mathbf{T}_4 = \{f(g(x, y)) \doteq f(y),\ g(i, z) \doteq z\}$$

Although $\mathbf{T}_4$ is rule-ambiguous, it is *rewrite*-nonambiguous:

$$f(w)[g(i, z)[y/z]/w] = f(g(i, y)) = f(g(x, y))[i/x]$$

and $f(w)$ is not an instance of $f(g(x, y))$, but the corresponding right-hand sides yield

$$f(w)[z[z/z]/w] = f(z) = f(y)[i, z/x, y]$$

$\mathbf{T}_4$ is rewrite-orthogonal, so $\xrightarrow{\mathbf{T}_4}$ is confluent.

Condition (4) may also be violated by a single self-overlapping equation, such as

$$\mathbf{T}_5 = \{f(f(x)) \doteq g(x)\}$$

The left-hand side $f(f(x))$ overlaps itself in $f(f(f(x)))$, with the second instance of the symbol f participating in two different instances of $f(f(x))$. Condition (4) is violated, because

$$f(y)[f(f(x))/y] = f(f(f(x))) = f(f(x))[f(x)/x]$$

but $f(y)$ is not an instance of $f(f(x))$. $\stackrel{\mathbf{T}_5}{\to}$ is not confluent, as $f(f(f(a))) \stackrel{\mathbf{T}_5}{\to} g(f(a))$ and $f(f(f(a))) \stackrel{\mathbf{T}_5}{\to} f(g(a))$, but both $g(f(a))$ and $f(g(a))$ are in normal form.

A final example of overlapping left-hand sides is

$$\mathbf{T}_6 = \{f(g(a,x),y) \doteq a,\ g(z,b) \doteq b\}$$

The left-hand sides of the two equations overlap in $f(g(a,b),y)$, with the symbol g participating in instances of the left-hand sides of both equations. Condition (4) is violated, because

$$f(w,y)[g(z,b)[a/z]/w] = f(g(a,b),y) = f(g(a,x),y)[b/x]$$

but $f(w,y)$ is not an instance of $f(g(a,x),y)$. $\stackrel{\mathbf{T}_6}{\to}$ is not confluent, as $f(g(a,b),c) \stackrel{\mathbf{T}_6}{\to} a$ by the first equation, and $f(g(a,b),c) \stackrel{\mathbf{T}_6}{\to} f(b,c)$ by the second equation, but both a and $f(b,c)$ are in normal form.

The equations for combinatory logic

$$\mathbf{T}_{SK} = \{@(@(@(S,x),y),z) \doteq @(@(x,z),@(y,z)),\ @(@(K,x),y) \doteq x\}$$

are rule-orthogonal, but not constructor-orthogonal, since the symbol @ (standing for application of a function to an argument) appears leftmost and also in the interior of left-hand sides. In more familiar notation, $@(\alpha,\beta)$ is written $(\alpha\beta)$, and leftward parentheses are omitted, so the equations look like

$$Sxyz \doteq xz(yz),\ Kxy \doteq x$$

Many functional programming languages vary the definition of constructor-orthogonality to allow pure applicative systems (the only symbol of arity greater than zero is the apply symbol @) in which the *zeroary* symbols (S and K in the example above) are partitioned into defined symbols and constructors.

The equations for addition of Horner-rule form polynomials in the symbolic variable V (V is a variable in the polynomials, but is treated formally as a constant symbol in the equations) are

$$\mathbf{T}_{poly} = \quad \{ \quad \begin{array}{r} +(+(w,*(V,x)),+(y,*(V,z))) \doteq +(+(w,y),*(V,+(x,z))), \\ +(w,+(y,*(V,z))) \doteq +(+(w,y),*(V,z)), \\ +(+(w,*(V,x)),y) \doteq +(+(w,y),*(V,x)) \end{array} \quad \}$$

This system is rule-orthogonal, but not constructor-orthogonal, because the symbols $+$ and $*$ appear leftmost and also in the interior of left-hand sides. In the more familiar infix form for $+$ and $*$, the equations look like

$$\begin{array}{r} (w+V*x)+(y+V*z) \doteq (w+y)+V*(x+z), \\ w+(y+V*z) \doteq (w+y)+V*z, \\ (w+V*x)+y \doteq (w+y)+V*x \end{array}$$

No natural variation on the definition of constructor-orthogonality seems to allow these equations. The only obvious way to simulate their behavior with a constructor-orthogonal system is to use two different symbols for addition, and two different symbols for multiplication, depending on whether the operation is active in adding two polynomials, or is merely part of the representation of a polynomial in Horner-rule form.

Although the polynomial example above shows that some natural sets of equations are rule-orthogonal but not constructor-orthogonal, Thatte has an automatic translation from rule-orthogonal to constructor-orthogonal systems [Tha85] showing that in some sense the programming power of the two classes of systems is the same. I still prefer to focus attention on the more general forms of orthogonality, because they deal more directly with the intuitive forms of equations, and because I believe that improved equational logic programming languages of the future will deal with even more general sets of equations, so I prefer to discourage dependence on the special properties of constructor systems.

Knuth-Bendix Methods. Although overlapping left-hand sides of equations *may* destroy the confluence property, there are many useful equational programs that are confluent in spite of overlaps. In particular, the equation expressing the associative property has a self-overlap, and equations expressing distributive or homomorphic properties often overlap with those expressing identity, idempotence, cancellation, or other properties that collapse a term. These overlaps are usually benign, and many useful equational programs containing similar overlaps are in fact confluent.

Example 2.3.17. Consider the singleton set

$$\mathbf{T}_7 = \{g(x, g(y, z)) \doteq g(g(x, y), z)\}$$

expressing the *associative* law for the operator g. This equation has a self-overlap, violating condition (4) of rewrite-orthogonality (Definition 2.3.9) because

$$g(w, u)[g(x, g(y, z))/u] = g(w, g(x, g(y, z))) = g(x, g(y, z))[w, x, g(y, z)/x, y, z]$$

but the corresponding right-hand sides disagree:

$$\begin{array}{c} g(w, u)[g(g(x, y), z)/u] = g(w, g(g(x, y), z)) \\ \neq \\ g(g(w, x), g(y, z)) = g(g(x, y), z)[w, x, g(y, z)/x, y, z] \end{array}$$

Nonetheless, $\overset{\mathbf{T}_7}{\to}$ is confluent. For example, while

$$g(a, g(b, g(c, d))) \overset{\mathbf{T}_7}{\to} g(a, g(g(b, c), d))$$

and

$$g(a, g(b, g(c, d))) \overset{\mathbf{T}_7}{\to} g(g(a, b), g(c, d))$$

by different applications of the equation, the two results rewrite to a common normal form by

$$g(a, g(g(b, c), d)) \overset{\mathbf{T}_7}{\to} g(g(a, g(b, c)), d) \overset{\mathbf{T}_7}{\to} g(g(g(a, b), c), d)$$

and

$$g(g(a, g(b, c)), d) \overset{\mathbf{T}_7}{\to} g(g(g(a, b), c), d)$$

Consider also the set

$$\mathbf{T}_8 = \{f(g(x, y)) \doteq g(f(x), f(y), \ g(i, z) \doteq z, \ f(i) \doteq i\}$$

expressing the distribution of f over g, and the fact that i is a left identity for g and a fixed point for f. The first and second equations overlap, violating condition (4) of rewrite-orthogonality, because

$$f(w)[g(i, z)/w] = f(g(i, z)) = f(g(x, y))[i, z/x, y]$$

but the corresponding right-hand sides disagree:

$$f(w)[z/w] = f(z) \neq g(f(i), f(z)) = g(f(x), f(y))[i, z/x, y]$$

Nonetheless, $\xrightarrow{\mathbf{T}_8}$ is confluent. For example, while

$$f(g(i,a)) \xrightarrow{\mathbf{T}_8} g(f(i),f(a))$$

by the first equation and

$$f(g(i,a)) \xrightarrow{\mathbf{T}_8} f(a)$$

by the second equation, the first result rewrites to the second, which is in normal form, by

$$g(f(i),f(a)) \xrightarrow{\mathbf{T}_8} g(i,f(a)) \xrightarrow{\mathbf{T}_8} f(a)$$

Notice that $\mathbf{T}_8 = \mathbf{T}_3 \cup \{f(i) \doteq i\}$, and that confluence failed for $\mathbf{T}_3$ (Example 2.3.16).

Experience with equational logic programming suggests that most naively written programs contain a small number of benign overlaps, which are almost always similar to the examples above. An efficient test for confluence in the presence of such overlaps would be extremely valuable.

The only known approach to proving confluence in spite of overlaps is based on the Knuth-Bendix procedure [KB70]. This procedure relies on the fact that local confluence (Definition 2.3.2) is often easier to verify than confluence, and that local confluence plus termination imply confluence.

Proposition 2.3.18 ([New42]). *If $\rightarrow$ is locally confluent, and there is no infinite sequence $s_0 \rightarrow s_1 \rightarrow \cdots$, then $\rightarrow$ is confluent.*

The proof is a simple induction on the number of steps to normal form.

Unfortunately, a system with nonterminating rewriting sequences may be locally confluent, but not confluent.

Example 2.3.19. $\mathbf{T}_1$ of Example 2.3.16 is locally confluent, but not confluent.

Consider also the set of equations

$$\mathbf{T}_9 = \{a \doteq b,\ b \doteq a,\ a \doteq c,\ b \doteq d\}$$

$\xrightarrow{\mathbf{T}_9}$ is locally confluent, but not confluent. Notice how confluence fails due to the two-step rewritings $a \xrightarrow{\mathbf{T}_9} b \xrightarrow{\mathbf{T}_9} d$ and $b \xrightarrow{\mathbf{T}_9} a \xrightarrow{\mathbf{T}_9} c$ (see Figure 2).

Another example, without a rewriting loop, is the set of equations

$$\mathbf{T}_{10} = \{f(x) \doteq g(h(x)),\ g(x) \doteq g(h(x)),\ f(x) \doteq c,\ g(x) \doteq d\}$$

$\xrightarrow{\mathbf{T}_{10}}$ is locally confluent, but not confluent. Again, confluence fails due to the two-step rewritings $f(x) \xrightarrow{\mathbf{T}_{10}} g(h(x)) \xrightarrow{\mathbf{T}_{10}} d$ and $g(x) \xrightarrow{\mathbf{T}_{10}} f(h(x)) \xrightarrow{\mathbf{T}_{10}} c$ (see Figure 3).

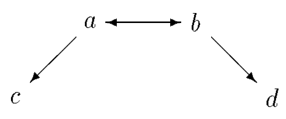


Fig. 2. $\mathbf{T}_9$ is locally confluent, but not confluent.

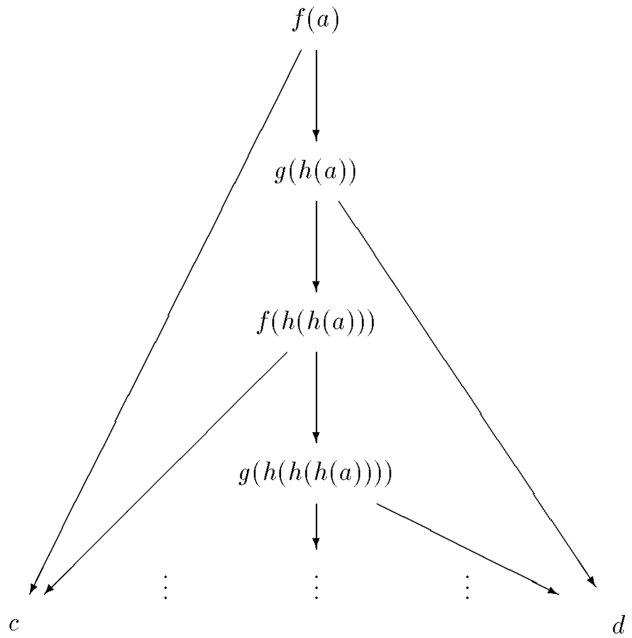


Fig. 3. $\mathbf{T}_{10}$ is locally confluent, but not confluent.

The Knuth-Bendix procedure examines overlaps one at a time to see whether they destroy the local confluence property. Given a pair of equations $l_1 \doteq r_1$ and $l_2 \doteq r_2$ whose left-hand sides overlap—i.e., there is a term $s \neq y$ with one occurrence of y such that

$$s[l_1[t_1, \ldots, t_m/x_1, \ldots, x_m]/y] = l_2[t'_1, \ldots, t'_n/x'_1, \ldots, x'_n]$$

but s is not an instance of l_2. For each s, l_1 and l_2, use the smallest $t_1, \ldots, t_m$ and $t'_1, \ldots, t'_n$ that satisfy this equation. The results of rewriting the instance of s above in two different ways, according to the overlapping instances of equations, are $c_1 = s[r_1[t_1, \ldots, t_m/x_1, \ldots, x_m]/y]$ and $c_2 = r_2[t'_1, \ldots, t'_n/x'_1, \ldots, x'_n]$. The pair $\langle c_1, c_2 \rangle$ is called a *critical pair*. A finite set of equations generates a finite set of critical pairs, since only a finite number of ss can be compatible with some l_2, but not an instance of l_2. The procedure checks all critical pairs to see if they rewrite to a common normal form. If so, the system is locally confluent.

Proposition 2.3.20 ([Hue80]). *Let* $\boldsymbol{T}$ *be a set of equations. If for every critical pair* $\langle c_1, c_2 \rangle$ *of* $\boldsymbol{T}$ *there is a term* d *such that* $c_1 \xrightarrow{\boldsymbol{T}}{}^{*} d$ *and* $c_2 \xrightarrow{\boldsymbol{T}}{}^{*} d$*, then* $\xrightarrow{\boldsymbol{T}}$ *is locally confluent.*

This proposition, and the Knuth-Bendix method, apply even to non-left-linear sets of equations. For example, the local confluence of $\mathbf{T}_1$ in Example 2.3.16 may be proved by inspecting all critical pairs.

When some critical pair cannot be rewritten to a common form, the Knuth-Bendix procedure tries to add an equation to repair that failure of local confluence. For equational logic programming, we would like to use just the part of the procedure that checks local confluence, and leave it to the programmer to decide how to repair a failure. Although, in principle, the search for a common form for a critical pair might go on forever, in practice a very shallow search suffices. I have never observed a natural case in which more than two rewriting steps were involved. Unfortunately, many useful equational programs have nonterminating term rewriting sequences, so local confluence is not enough. The design of a variant of the Knuth-Bendix procedure that is practically useful for equational logic programming is an open topic of research—some exploratory steps are described in [CO91]. A number of methods for proving termination are known [Der87, GKM83], which might be applied to portions of an equational program even if the whole program is not terminating, but we have no experience with the practical applicability of these methods. If the rewriting of the terms c_1 and c_2 in a critical pair to a common form d (see Proposition 2.3.20) takes no more than one rewriting step (this is *one-step confluence,* Definition 2.3.3), then we get confluence and not just local confluence. Rewrite-orthogonal systems are those whose critical pairs are all trivial—the members of the pair are equal, and so the reduction to a common form takes zero steps. Unfortunately, all of the important examples so far of confluent but not rewrite-orthogonal equational programs have the basic structure of associativity or distributivity (see Example 2.3.17) and require two rewriting steps to resolve their critical pairs.

The sets of equations in Example 2.3.17 pass the Knuth-Bendix test for local confluence, and a number of well-known techniques can be used to prove that there is no infinite term rewriting sequence in these systems. But, we need to recognize many variations on these example systems, when they are embedded in much larger sets of equations which generate some infinite term rewriting sequences, and no completely automated method has yet shown practical success at that problem (although there are special treatments of commutativity and associativity [BPW89, DHJP83]). On the other hand, in practice naturally constructed systems of equations that are locally confluent are almost always confluent. Surely someone will find a useful and efficient formal criterion to distinguish the natural constructions from the pathological ones of Example 2.3.19.

3 Term Rewriting Proof Strategies

Given an orthogonal set of equations $\mathbf{T} = \{l_1 \doteq r_1, \ldots, l_m \doteq r_m\}$, or any set with confluent term rewriting relation $\xrightarrow{\mathbf{T}}$, we may now answer all questions of the form $(\mathbf{norm}\, l_1, \ldots, l_m : t)$ by exploring term rewriting sequences starting with t. Confluence guarantees that if there is an answer, some term rewriting sequence will find it (Proposition 2.3.6). Furthermore, confluence guarantees that no finite number of term rewriting steps can be catastrophic, in the sense that if $s \xrightarrow{\mathbf{T}}{}^{*} t$ and if s rewrites to a normal form, then t rewrites to the same normal form. Confluence, however, does not guarantee that no infinite term rewriting sequence can be catastrophic.

Example 3.0.1. Consider the set of equations

$$\mathbf{T}_{11} = \{car(cons(x, y)) \doteq x,\ a \doteq f(a)\}$$

The first equation is the usual definition of the *car* (left projection) function of Lisp, the second is a silly example of an equation leading to infinite term rewriting. $\mathbf{T}_{11}$ is orthogonal, so $\xrightarrow{\mathbf{T}_{11}}$ is confluent. But $car(cons(b, a))$ rewrites to the normal form b, and also in the infinite rewriting sequence $car(cons(b, a)) \xrightarrow{\mathbf{T}_{11}} car(cons(b, f(a))) \xrightarrow{\mathbf{T}_{11}} \cdots$.

Notice that, due to confluence, no matter how far we go down the infinite term rewriting sequence $car(cons(b, a)) \xrightarrow{\mathbf{T}_{11}} car(cons(b, f(a))) \xrightarrow{\mathbf{T}_{11}} \cdots$, one application of the first equation leads to the normal form b. Nonetheless, a naive strategy might fail to find that normal form by making an infinite number of unfortunate rewrites. In fact, the usual recursive evaluation techniques used in Lisp and other term-evaluating languages correspond to term rewriting strategies that choose infinite sequences whenever possible. A breadth-first search of all possible rewriting sequences is guaranteed to find all normal forms, but at the cost of a lot of unnecessary work.

For efficient implementation of equational logic programming, we need strategies for choosing term rewriting steps so that

- a small number of term rewriting sequences is explored, preferably only one;
- if there is a normal form, it is found, preferably by the shortest or cheapest sequence possible.

Some theoretical work on sequencing in the lambda calculus has already been explored under the title of *one-step strategies* [Bar84].

3.1 Complete and Outermost Complete Rewriting Sequences

In orthogonal systems of equations, there are two useful results on strategies that are guaranteed to find normal forms. The formal notation for stating these results precisely is somewhat involved (see the chapter 'Equational Reasoning and Term Rewriting Systems in Volume 1), so I only give rough definitions here. The concepts in this section can be extended to nonorthogonal systems, but in some cases there are very subtle problems in the extensions, and they have never been treated in the literature.

Definition 3.1.1 ([HL91]). A *redex* is an occurrence of an instance of a left-hand side of an equation in a term. An *outermost* redex is one that is not nested inside any other redex. When α is a redex in s, and $s \xrightarrow{\mathbf{T}}{}^* t$, the *residuals* of α in t are the redexes in t that correspond in the obvious way to α in s—they are essentially explicit copies of α, except that some rewriting step may have rewritten a subterm of α, so that some copies may be modified. All residuals of α are occurrences of instances of the same left-hand side as α.

Example 3.1.2. Consider the rule-orthogonal set of equations

$$\mathbf{T}_{12} = \{f(x) \doteq g(x,x),\ g(h(x),y) \doteq h(x),\ a \doteq b\}$$

The term $g(f(f(h(a))), f(h(a)))$ has five redexes: two occurrences each of the terms a and $f(h(a))$, and one occurrence of $f(f(h(a)))$. The latter two are both instances of the left-hand side $f(x)$ of the first equation. The leftmost occurrence of $f(h(a))$ is nested inside $f(f(h(a)))$, so it is not outermost. Each occurrence of a is nested inside an occurrence of $f(h(a))$, so neither is outermost. The rightmost occurrence of $f(h(a))$, and the sole occurrence of $f(f(h(a)))$, are both outermost redexes. In the rewriting sequence below, the leftmost occurrence of $f(h(a))$, and its residuals in each succeeding term, are boxed.

$$
\begin{aligned}
g(f(\boxed{f(h(a))}), f(h(a))) &\overset{\mathbf{T}_{12}}{\to} g(g(\boxed{f(h(a))}, \boxed{f(h(a))}), f(h(a))) \\
&\overset{\mathbf{T}_{12}}{\to} g(g(\boxed{f(h(b))}, \boxed{f(h(a))}), f(h(a))) \\
&\overset{\mathbf{T}_{12}}{\to} g(g(g(h(b), h(b)), \boxed{f(h(a))}), f(h(a))) \\
&\overset{\mathbf{T}_{12}}{\to} g(g(h(b), \boxed{f(h(a))}), f(h(a))) \\
&\overset{\mathbf{T}_{12}}{\to} g(h(b), f(h(a))) \\
&\overset{\mathbf{T}_{12}}{\to} h(b)
\end{aligned}
$$

Notice how the leftmost occurrence of $f(h(a))$ in the first term of the sequence is copied into two occurrences in the second, due to the rewriting of a redex in which it is nested. The first of these is changed to $f(h(b))$ in the third term of the sequence, but it is still a residual of the original leftmost $f(h(a))$. In the fourth term of the sequence, $f(h(b))$ is rewritten, eliminating one of the residuals. In the sixth term, the remaining residual, still in the form $f(h(a))$, is eliminated due to rewriting of a redex in which it is nested. Another occurrence of $f(h(a))$ remains, but it is a residual of the rightmost occurrence of that term in the first term of the sequence.

In general, a residual α of a redex is eliminated when α is rewritten (or, in rewrite-orthogonal systems, when a redex overlapping α is rewritten). α is copied zero, one, or more times (zero times eliminates the residual) when another redex in which α is nested is rewritten. α remains the same when another redex disjoint from α is rewritten. Finally, α is modified in form, but remains an instance of the same left-hand side, when another redex nested inside α is rewritten.

In orthogonal systems, the nontrivial, rule-like, and nonambiguous qualities of equations (restrictions 1, 2, 4 or $4'$ of Definition 2.3.9 or 2.3.10) guarantee that a given redex may be rewritten in precisely one way. So, a term rewriting strategy need only choose a redex to rewrite at each step. The most obvious way to insure that all normal forms are found is to rewrite every redex fairly.

Definition 3.1.3 ([O'D77]). A finite or infinite term rewriting sequence $t_0 \to t_1 \to \cdots$ is *complete* (also called *fair*) if and only if, for every i and every redex α in t_i, there exists a $j > i$ such that t_j contains no residual of α.

A complete term rewriting sequence is fair to all redexes, in the sense that every redex α (or its residuals, which are essentially the later versions of the redex) eventually gets eliminated, either by rewriting α (with rewrite-orthogonality, a redex overlapping α), or by making zero copies of α while rewriting another redex in which α is nested. Complete term rewriting sequences are maximal, in the sense that they produce terms that are rewritten further than every other sequence. Since nothing is rewritten further than a normal form, complete sequences produce a normal form whenever there is one.

Proposition 3.1.4 ([O'D77]). *Let $\boldsymbol{T}$ be an orthogonal set of equations, let $t_0 \xrightarrow{\boldsymbol{T}} t_1 \xrightarrow{\boldsymbol{T}} \cdots$ be a complete rewriting sequence, and let s be any term such that $t_0 \xrightarrow{\boldsymbol{T}}{}^* s$. There exists an i such that $s \xrightarrow{\boldsymbol{T}}{}^* t_i$. In particular, if s is in normal form, then $t_i = s$.*

Computing a single complete term rewriting sequence is generally cheaper than searching breadth-first among a number of sequences, but fair rewriting strategies (such as the strategy of adding new redexes to a queue, and rewriting all residuals of the head redex in the queue) typically perform a substantial number of superfluous rewriting steps, and can easily waste an exponentially growing amount of work in some cases. Since a residual α of a redex may only be eliminated by rewriting α, or some redex inside which α is nested, we need only be fair to the outermost redexes in order to be sure of finding normal forms.

Definition 3.1.5 ([O'D77]). A finite or infinite term rewriting sequence $t_0 \to t_1 \to \cdots$ is *outermost complete* or *outermost fair* (called *eventually outermost* in [O'D77]) if and only if, for every i and every outermost redex α in t_i, there exists a $j > i$ such that the unique residual of α in t_{j-1} is either eliminated by rewriting in t_j, or is no longer outermost in t_j (equivalently, no residual of α is outermost in t_j).

Since, for the least j satisfying the definition above, α remains outermost from t_i through t_{j-1} and cannot be copied, there is no loss of generality in requiring the residual of α in t_{j-1} to be unique.

Proposition 3.1.6 ([O'D77]). *Let $\boldsymbol{T}$ be a rule-orthogonal set of equations, let $t_0 \xrightarrow{\boldsymbol{T}} t_1 \xrightarrow{\boldsymbol{T}} \cdots$ be an outermost complete rewriting sequence, and let s be a (unique) normal form for t_0. There exists an i such that $t_i = s$.*

[O'D77] proves this proposition for a form of orthogonality intermediate between rule- and rewrite-orthogonality. I conjecture that the proof generalizes to rewrite-orthogonality.

The requirement that **T** be orthogonal, and not just confluent, is essential to Proposition 3.1.6.

Example 3.1.7. Consider the set of equations

$$\mathbf{T}_{13} = \{f(g(x,b)) \doteq b,\ g(x,y) \doteq g(f(x),y),\ a \doteq b\}$$

These equations are confluent, but not rewrite-orthogonal, since the left-hand sides of the first and second equations overlap in $f(g(x,b))$, but the corresponding right-hand sides yield $b \neq f(g(f(x),b))$. The natural outermost complete rewriting sequence starting with $f(g(b,a))$ is the infinite one

$$f(g(b,a)) \stackrel{\mathbf{T}_{13}}{\rightarrow} f(g(f(b),a)) \stackrel{\mathbf{T}_{13}}{\rightarrow} f(g(f(f(b)),a)) \stackrel{\mathbf{T}_{13}}{\rightarrow} \cdots$$

But $f(g(b,a))$ rewrites to normal form by

$$f(g(b,a)) \stackrel{\mathbf{T}_{13}}{\rightarrow} f(g(b,b)) \stackrel{\mathbf{T}_{13}}{\rightarrow} b$$

The problem is that rewriting the nonoutermost redex a to b creates a new outermost redex for the first equation *above* the previously outermost one for the second equation. This leapfrogging effect allows an inner redex to kill an outer one indirectly, by creating another redex even closer to the root. There should be some interesting conditions, weaker than rewrite-orthogonality, that prohibit this leapfrogging effect and guarantee outermost termination for confluent systems.

The obvious way to generate outermost-complete rewriting sequences is to alternate between finding all outermost redexes, and rewriting them all. The order in which the outermost redexes are rewritten is irrelevant since they are all disjoint and cannot cause copying or modification of one another. Unfortunately, this strategy often generates a lot of wasted work. For example, consider a system containing the equations $\mathbf{T}_{cond}$ for the conditional function from Example 1.2.1

$$\mathbf{T}_{cond} = \{cond(true,x,y) \doteq x,\ cond(false,x,y) \doteq y\}$$

In a term of the form $cond(r,s,t)$, there will usually be outermost redexes in all three of r s and t. But, once r rewrites to either *true* or *false*, one of s and t will be thrown away, and any rewriting in the discarded subterm will be wasted. The ad hoc optimization of noticing when rewriting of one outermost redex immediately causes another to be nonoutermost sounds tempting, but it probably introduces more overhead in detecting such cases than it saves in avoiding unnecessary steps. Notice that it will help the *cond* example only when r rewrites to *true* or *false* in one step. So, we need some further analysis to choose which among several outermost redexes to rewrite.

3.2 Sequentiality Analysis and Optimal Rewriting

For rewrite-orthogonal systems of equations in general, it is impossible to choose reliably a redex that *must* be rewritten in order to reach normal form, so that there is no risk of wasted work.

Example 3.2.1. Let $\mathbf{T}_c$ be equations defining some general-purpose programming system, such as Lisp. The forms of the particular equations in $\mathbf{T}_c$ are not important to this example, merely the fact that they are powerful enough for general-purpose programming. Assume that in the system $\mathbf{T}_c$ there is an effective way to choose a redex that must be rewritten to reach normal form (this is the case for typical definitions of Lisp). Now, add the *positive parallel-or* equations

$$\mathbf{T}_{or+} = \{or(true, x) \doteq true,\ or(x, true) \doteq true\}$$

and consider the system $\mathbf{T}_c \cup \mathbf{T}_{or+}$.

For an arbitrary given term $or(s, t)$, we would like to choose either s or t to rewrite first. If s rewrites to $true$, but t does not, then it is crucial to choose s, else work (possibly infinitely much work) will be wasted. Similarly, if t rewrites to $true$, but s does not, it is crucial to choose t. If *neither* s nor t rewrites to $true$, then both must be rewritten to normal form in order to normalize the whole term, so we may choose either. If *both* s and t rewrite to $true$ then, ideally, we would like to choose the one that is cheapest to rewrite, but suppose that we are satisfied with either choice in this case also.

Suppose that we have an effective way to choose s or t above. Then, we have a *recursive separation* of the terms $or(s, t)$ in which s rewrites to $true$ and t has no normal form from those in which t rewrites to $true$ and s has no normal form. Such a separation is known to be impossible. (It would lead easily to a computable solution of the halting problem. See [MY78] for a discussion of *recursive inseparability*.) So, we cannot decide effectively whether to rewrite redexes in s or in t without risking wasted work.

The case where *both* s and t rewrite to $true$ poses special conceptual problems for sequentiality theory. Although it is necessary to rewrite *one* of s or t in order to reach the normal form $true$, it is neither necessary to rewrite s, nor necessary to rewrite t. The criterion of choosing a redex that *must* be rewritten fails to even *define* a next rewriting step mathematically, and the question of computability does not even arise. Notice that this latter case is problematic for $\mathbf{T}_{or+}$ alone, without the addition of $\mathbf{T}_c$.

The difficulty in Example 3.2.1 above appears to depend on the unifiability of the left-hand sides of the two equations in $\mathbf{T}_{or+}$, which is allowed in rewrite-orthogonal systems, but not rule-orthogonal systems. A more subtle example, due to Huet and Lévy, shows that rule-orthogonality is still not sufficient for effective sequencing.

Example 3.2.2 ([HL91]). Replace the parallel-or equations of Example 3.2.1 by the following:

$$\mathbf{T}_{14} = \{f(x,a,b) \doteq c,\ f(b,x,a) \doteq c,\ f(a,b,x) \doteq c\}$$

and consider the system $\mathbf{T}_c \cup \mathbf{T}_{14}$. Given a term of the form $f(r,s,t)$, we cannot decide whether to rewrite redexes in r, in s, or in t without risking wasted work, because we cannot separate computably the three cases

- $r \to^* a$ and $s \to^* b$
- $r \to^* b$ and $t \to^* a$
- $s \to^* a$ and $t \to^* b$

Unlike the parallel-or example, it is impossible for more than one of these three cases to hold. There is always a mathematically well-defined redex that *must* be rewritten in order to reach a normal form, and the problem is entirely one of choosing such a redex effectively. In fact, for sets $\mathbf{T}$ of equations such that $\mathbf{T} \cup \mathbf{T}_{14}$ is *terminating* (every term has a normal form), the choice of whether to rewrite r, s, or t in $f(r,s,t)$ is effective, but usually unacceptably inefficient.

So, some further analysis of the form of equations beyond checking for orthogonality is required in order to choose a good redex to rewrite next in a term rewriting sequence. Analysis of equations in order to determine a good choice of redex is called *sequentiality analysis*.

3.2.1 Needed Redexes and Weak Sequentiality

The essential ideas for sequentiality analysis in term rewriting are due to Huet and Lévy [HL91], based on a notion of sequential predicate by Kahn [KP78]. A redex that may be rewritten without risk of wasted work is called a *needed redex*.

Definition 3.2.3 ([HL91]). Given an orthogonal set $\mathbf{T}$ of equations and a term t_0, a redex α in t_0 is a *needed redex* if and only if, for every term rewriting sequence $t_0 \xrightarrow{\mathbf{T}} t_1 \xrightarrow{\mathbf{T}} \cdots \xrightarrow{\mathbf{T}} t_m$, either

- there exists an i, $1 \leq i \leq m$ such that a residual of α is rewritten in the step $t_{i-1} \xrightarrow{\mathbf{T}} t_i$, or
- α has at least one residual in t_m.

A needed redex is a redex whose residuals can never be completely eliminated by rewriting other redexes. So, the rewriting of a needed redex is not wasted work, since at least one of its residuals has to be rewritten in order to reach normal form. Huet and Lévy defined needed redexes only for terms with normal forms, but the generalization above is trivial. A system is *weakly sequential* if there is always a needed redex to rewrite.

Definition 3.2.4. A rewrite-orthogonal set of equations is *weakly sequential* if and only if every term that is not in normal form contains at least one needed redex. A set of equations is *effectively weakly sequential* if and only if there is an effective procedure that finds a needed redex in each term not in normal form.

The word 'sequential' above is conventional, but may be misleading to those interested in parallel computation. A weakly sequential system is not *required* to be computed sequentially—typically there is great opportunity for parallel evaluation. Rather, a weakly sequential system *allows* sequential computation without risk of wasted rewriting work. In this respect 'sequentializable' would be a more enlightening word than 'sequential.'

The parallel-or system $\mathbf{T}_{or+}$ of Example 3.2.1 is rewrite-orthogonal, but not weakly sequential, because the term

$$or(or(true, a), or(true, a))$$

has two redexes, neither of which is needed, since either can be eliminated by rewriting the other, then rewriting the whole term to $true$. Rule-orthogonality guarantees weak sequentiality.

Proposition 3.2.5 ([HL91]). *A nontrivial, rule-like, and left-linear set of equations (Definition 2.3.9) is weakly sequential if and only if it is rule-orthogonal.*

The proof of ($\Leftarrow$) is in [HL91]. It involves a search through all rewriting sequences (including infinite ones), and does not yield an effective procedure. ($\Rightarrow$) is straightforward, since when two redexes overlap neither is needed.

Proposition 3.2.5 above shows that no analysis based on weak sequentiality can completely sequentialize systems whose confluence derives from rewrite-orthogonality, or from a Knuth-Bendix analysis. Section 3.2.4 discusses possible extensions of sequentiality beyond rule-orthogonal systems.

The system $\mathbf{T}_c \cup \mathbf{T}_{14}$ of Example 3.2.2 is rule-orthogonal, and therefore weakly sequential. For example, in a term of the form $f(r, s, t)$, where $r \rightarrow^* a$ and $s \rightarrow^* b$, both r and s contain needed redexes. The subsystem $\mathbf{T}_{14}$, without the general-purpose programming system $\mathbf{T}_c$, is *effectively* weakly sequential, but only because it is terminating. I conjecture that effective weak sequentiality is undecidable for rule-orthogonal systems.

3.2.2 Strongly Needed Redexes and Strong Sequentiality

The uncomputability of needed redexes and the weak sequential property are addressed analogously to the uncomputability of confluence: by finding efficiently computable sufficient conditions for a redex to be needed, and for a system to be effectively weakly sequential. A natural approach is to ignore right-hand sides of equations, and detect those cases of needed redexes and effectively weakly sequential systems that are guaranteed by the structure of the left-hand sides. To this end we define ω-rewriting, in which a redex is replaced by an arbitrary term.

Definition 3.2.6 ([HL91]). Let $\mathbf{T} = \{l_1 \doteq r_1, \ldots, l_n \doteq r_n\}$ be a rule-orthogonal set of equations.

A term s_1 *ω-rewrites to s_2 by* $\mathbf{T}$ (written $s_1 \xrightarrow[\omega]{\mathbf{T}} s_2$) if and only if there is a term t, a variable x with exactly one occurrence in t, an instance l'_i of a left-hand side l_i in $\mathbf{T}$, and a term r such that $s_1 = t[l'_i/x]$ and $s_2 = t[r/x]$. That is, s_2 results from finding exactly one instance of a left-hand side of an equation in $\mathbf{T}$ occurring as a subterm of s_1, and replacing it with an arbitrary term. The definition of residual (Definition 3.1.1) generalizes naturally to ω-rewriting.

Now, a *strongly needed redex* is defined analogously to a needed redex, using ω-rewriting instead of rewriting.

Definition 3.2.7 ([HL91]). Given a rule-orthogonal set $\mathbf{T}$ of equations and a term t_0, a redex α in t_0 is *strongly needed* if and only if, for every ω-rewriting sequence $t_0 \xrightarrow[\omega]{\mathbf{T}} t_1 \xrightarrow[\omega]{\mathbf{T}} \cdots \xrightarrow[\omega]{\mathbf{T}} t_m$, either

- there exists an i, $1 \leq i \leq m$ such that a residual of α is rewritten in the step $t_{i-1} \xrightarrow[\omega]{\mathbf{T}} t_i$, or
- α has at least one residual in t_m.

Because of rule-orthogonality, the property of being strongly needed depends only on the location of a redex occurrence, and not on its internal structure. So, we call an arbitrary occurrence in a term *strongly needed* if and only if a redex substituted in at that location is strongly needed. [HL91] defines *strong indexes*, and shows that they determine exactly the strongly needed redexes. It is easy to see that every strongly needed redex is needed, and outermost. And, it is easy to detect whether a given redex is strongly needed (see Section 4 and [HL91]). A system of equations is strongly sequential if there is always a strongly needed redex to be rewritten, except in a normal form term.

Definition 3.2.8 ([HL91]). A rule-orthogonal set of equations is *strongly sequential* if and only if every term that is not in normal form contains at least one strongly needed redex.

It is obvious that every strongly sequential system is effectively weakly sequential, but the converse does not hold.

Example 3.2.9. The system $\mathbf{T}_{14}$ of Example 3.2.2, although it is effectively weakly sequential, is *not* strongly sequential. $f(f(a,b,c), f(a,b,c), f(a,b,c))$ ω-rewrites to $f(f(a,b,c), a, b)$, which is a redex, so the first redex $f(a,b,c)$ is not strongly needed. Similarly, $f(f(a,b,c), f(a,b,c), f(a,b,c))$ ω-rewrites to the redexes $f(b, f(a,b,c), a)$ and $f(a, b, f(a,b,c))$, so the second and third redexes are not strongly needed. All three redexes are weakly needed.

By contrast, consider the strongly sequential system

$$\mathbf{T}_{15} = \{f(g(b,b), h(x,b)) \doteq b,\ f(g(c,x), h(c,c)) \doteq c,\ h(x,d) \doteq d,\ a \doteq b\}$$

In the term $f(g(a,a), h(a,a))$, the first and last occurrences of a are strongly needed, but the second and third are not.

Notice that ω-rewriting allows different redexes that are occurrences of instances of the same left-hand side to be rewritten inconsistently in different ω-rewriting steps. Such inconsistency is critical to the example above, where in one case $f(a,b,c)$ ω-rewrites to a, and in another case it ω-rewrites to b.

Strong sequentiality is independent of the right-hand sides of equations.

Proposition 3.2.10. *If $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ are left-similar (see Definition 2.3.11), and $\boldsymbol{T}_1$ is strongly sequential, then so is $\boldsymbol{T}_2$.*

The proof is straightforward, since $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ clearly have the same ω-rewriting relations (Definition 3.2.6).

It is *not* true, however, that a system is strongly sequential whenever all left-similar systems are weakly sequential. The system of Example 3.2.2 and all left-similar systems are weakly sequential, but not strongly sequential. But in that case, no given redex is needed in all of the left-similar systems. An interesting open question is whether a redex that is needed in all left-similar systems must be strongly needed.

For finite rule-orthogonal sets of equations, strong sequentiality is decidable.

Proposition 3.2.11 ([HL91, KM91]). *Given a finite rule-orthogonal set $\boldsymbol{T}$ of equations, it is decidable whether $\boldsymbol{T}$ is strongly sequential.*

The details of the proof are quite tricky, but the essential idea is that only a finite set of terms, with sizes limited by a function of the sizes of left-hand sides of equations, need to be checked for strongly needed redexes.

In developing the concept of *strongly needed* redexes and connecting it to the concept of *weakly needed* redexes, Huet and Lévy define the intermediately powerful concept of an *index*. Roughly, an index is a needed redex that can be distinguished from other redexes just by their relative positions in a term, without knowing the forms of the redexes themselves [HL91]. Every index is a weakly needed redex, but not vice versa. *Strong* indexes are equivalent to strongly needed redexes. A system in which every term not in normal form has at least one index is called *sequential*. The precise relation between sequentiality in all left-similar systems, and strong sequentiality, is an interesting open problem.

All of the sequentiality theory discussed in this section deals with sequentializing the process of rewriting a term to normal form. Example 7.1.6 in Section 7.1 shows that even strongly sequential systems may require a parallel evaluation strategy for other purposes, such as a complete procedure for rewriting to head-normal form.

3.2.3 Optimal Rewriting

From a naive point of view, the natural strategy of rewriting a strongly needed redex at each step does not lead to minimal-length rewriting sequences ending in normal form. The problem is that the rewriting of a strongly needed redex may cause another needed (but not strongly needed) redex to be copied arbitrarily many times. Since strongly needed redexes are always outermost, they are particularly likely to cause such copying.

Example 3.2.12. In the strongly sequential system of equations

$$\mathbf{T}_{16} = \{f(x) \doteq g(x, x)\}$$

given the initial term $f(f(a))$, both redexes, $f(f(a))$ and $f(a)$, are needed, but only the outermost one is strongly needed. By rewriting a strongly needed redex at each step, we get the 3-step sequence

$$f(f(a)) \overset{\mathbf{T}_{16}}{\to} g(f(a), f(a)) \overset{\mathbf{T}_{16}}{\to} g(g(a, a), f(a)) \overset{\mathbf{T}_{16}}{\to} g(g(a, a), g(a, a))$$

But, there is a 2-step sequence

$$f(f(a)) \overset{\mathbf{T}_{16}}{\to} f(g(a, a)) \overset{\mathbf{T}_{16}}{\to} g(g(a, a), g(a, a))$$

which does not rewrite the unique strongly needed redex in the first step.

It is easy to construct further examples in which the number of steps wasted by rewriting strongly needed redexes is arbitrarily large.

Proposition 3.2.13. *Given an arbitrary strongly sequential system of equations, and a term, there is no effective procedure to choose a redex at each step so as to minimize the length of the rewriting sequence to normal form.*

I am not aware of a treatment of this point in the literature. The basic idea is to use the equations

$$\boldsymbol{T}_{17} = \{f(x,y) \doteq g(x,x,h(y)),\ a \doteq b,\ g(x,x,0) \doteq 0\}$$

similar to $\boldsymbol{T}_{16}$ of Example 3.2.12, and add additional equations to define h as the evaluator for Lisp or some other general-purpose computing system. Now, start with a term of the form $f(a,p)$, where p is an arbitrary program. If evaluation of p halts with value 0 (that is, if $h(p)$ rewrites to 0), then an optimal rewriting sequence rewrites $f(a,p)$ to $g(a,a,h(p))$ in the first step, and never rewrites the as. If evaluation of p halts with any value other than 0, then an optimal sequence rewrites a to b in the first step (else it must be rewritten twice later). An effective method for choosing the first step would yield a recursive separation of the programs that halt with output 0 from those that halt with output 1, which is known to be impossible [MY78].

Notice that, when there is a normal form, breadth-first search over all rewriting sequences yields a very expensive computation of a minimal sequence. But, no effective procedure can choose some redex in all cases (even in the absence of a normal form), and minimize the number of rewriting steps when there is a normal form.

The uncomputability of minimal-length rewriting strategies in Proposition 3.2.13 sounds discouraging. The number of rewriting steps is *not*, however, a good practical measure of the efficiency of a sequencing strategy. Given equations, such as $f(x) \doteq g(x,x)$ in Example 3.2.12, with more than one occurrence of the same variable x on the right-hand side, normal sensible implementations do *not* make multiple copies of the subterm substituted for that variable. Rather, they use multiple pointers to a single copy. Then, only one actual computing step is required to rewrite all of the apparent multiple copies of a redex within that substituted subterm. So, in Example 3.2.12, the strategy of choosing a strongly needed redex actually leads to only two steps, from $f(f(a))$ to $g(f(a), f(a))$, and then directly to $g(g(a,a), g(a,a))$. The normal form is represented in practice with only one copy of the subterm $g(a,a)$, and two pointers to it for the two arguments of the outermost g. If we charge only one step for rewriting a whole set of shared redexes, then rewriting strongly needed redexes is optimal.

Proposition 3.2.14. *Consider multiple-rewriting sequences, in which in one step all of the shared copies of a redex are rewritten simultaneously. Given a strongly sequential set of equations and a term, the strategy of rewriting at each step a strongly needed redex and all of its shared copies leads to normal form in a minimal number of steps.*

This proposition has never been completely proved in print. I claimed a proof [O'D77] but had a fatal error [BL79, O'D79]. The hard part of the proposition—that the rewriting of a strongly needed redex is never a wasted step—was proved by Huet and Lévy [HL91]. The remaining point—that rewriting a strongly needed redex never causes additional rewriting work later in the sequence—seems obvious, but has never been treated formally in general. Lévy [Lév78] treated a similar situation in the lambda calculus, but in that case there is no known efficient implementation technique for the sequences used in the optimality proof. Although the formal literature on optimal rewriting is still incomplete, and extensions of optimality theory to systems (such as the lambda calculus) with bound variables are extremely subtle, for most practical purposes Huet's and Lévy's work justifies the strategy of rewriting all shared copies of a strongly needed redex at each step. Optimality aside, rewriting strategies that always choose a strongly needed redex are examples of *one-step normalizing strategies,* which provide interesting theoretical problems in combinatory logic and the lambda calculus [Bar84].

3.2.4 Extensions to Sequentiality Analysis

Proposition 3.2.5 seems to invalidate rewrite-orthogonal systems for efficient or optimal sequential rewriting. A closer look shows that the definition of weakly needed redexes and weak sequentiality is inappropriate for rewrite-orthogonal systems. When two redexes from a rewrite-orthogonal system overlap, we get the same result by rewriting either one. So, there is no need for a sequential strategy to choose between them, and we might as well allow an arbitrary selection. This observation suggests a more liberal concept of needed redex.

Definition 3.2.15. Given a rewrite-orthogonal set $\mathbf{T}$ of equations and a term t_0, a redex α in t_0 is a *rewrite-needed redex* if and only if, for every term rewriting sequence $t_0 \xrightarrow{\mathbf{T}} t_1 \xrightarrow{\mathbf{T}} \cdots \xrightarrow{\mathbf{T}} t_m$, either

- there exists an i, $1 \leq i \leq m$ such that a residual of α is rewritten in the step $t_{i-1} \xrightarrow{\mathbf{T}} t_i$, or
- there exists an i, $1 \leq i \leq m$ such that a redex overlapping a residual of α is rewritten in the step $t_{i-1} \xrightarrow{\mathbf{T}} t_i$, or
- α has at least one residual in t_m.

This is the same as Definition 3.2.3 of needed redex, except that when one redex is reduced, we give credit to all redexes that overlap it. We generalize Definition 3.2.4 with the new version of needed redexes.

Definition 3.2.16. An orthogonal set of equations is *weakly rewrite-sequential* if and only if every term that is not in normal form contains at least one rewrite-needed redex. A set of equations is *effectively weakly rewrite-sequential* if and only if there is an effective procedure that finds a rewrite-needed redex in each term not in normal form.

Three sorts of overlaps between left-hand sides of equations have different impacts on weak rewrite-sequentiality. Recall (Definition 2.3.9) that the problematic overlaps occur when there is a term s, and left-hand sides l_i and l_j, such that

$$s[l_i[t_1, \ldots, t_m/x_1, \ldots, x_m]/y] = l_j[t'_1, \ldots, t'_n/x'_1, \ldots, x'_n]$$

Rewrite-nonambiguity requires that either s is an instance of l_j, or the corresponding right-hand sides r_i and r_j satisfy

$$s[r_i[t_1, \ldots, t_m/x_1, \ldots, x_m]/y] = r_j[t'_1, \ldots, t'_n/x'_1, \ldots, x'_n]$$

1. Sometimes the structure of the inner term l_i is entirely subsumed by the structure of the outer term l_j—that is, the substituted terms $t'_1, \ldots, t'_n$ are trivial, and

   $$s[l_i[t_1, \ldots, t_m/x_1, \ldots, x_m]/y] = l_j$$

   In this case, the equation $l_j \doteq r_j$ is redundant, since every possible application of it can be accomplished by applying $l_i \doteq r_i$ instead.

2. Sometimes the structure of the inner term l_i extends below the structure of the outer term l_j—that is, the substituted terms $t_1, \ldots, t_m$ are trivial, and

   $$s[l_i/y] = l_j[t'_1, \ldots, t'_n/x'_1, \ldots, x'_n]$$

   Overlaps of this sort appear not to destroy weak rewrite-sequentiality.

3. Otherwise, neither set of substituted terms $t_1, \ldots, t_m$ nor $t'_1, \ldots, t'_n$ is trivial. This is the interesting case. Weak rewrite-sequentiality may hold or not, depending on the extent to which redexes in substituted subterms are copied or eliminated by the right-hand sides.

Figure 4 illustrates the three types of overlap with suggestive pictures.

Example 3.2.17. Consider the set

$$\mathbf{T}_{18} = \{f(g(h(x))) \doteq f(h(x)),\ g(x) \doteq x\}$$

The overlap here is of type 1, since

$$f(y)[g(x)[h(x)/x]/y] = f(g(h(x)))$$

The first equation is redundant since it is essentially a special case of the second.

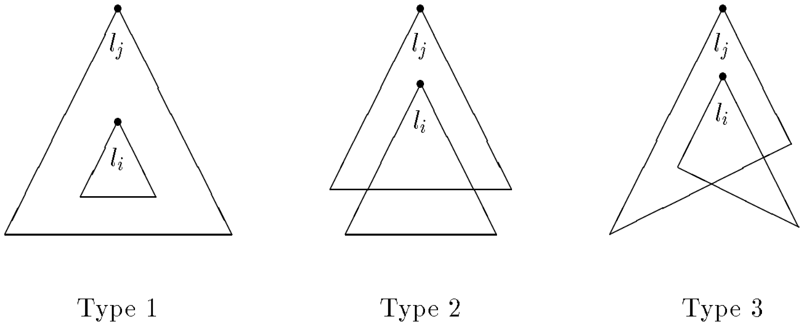


Fig. 4. The three types of left-hand-side overlap.

Next, consider

$$\mathbf{T}_{19} = \{f(g(x), a) \doteq f(x, a),\ g(h(x)) \doteq h(x)\}$$

The overlap here is of type 2, since

$$f(y, a)[g(h(x))/y] = f(g(h(x)), a) = f(g(x), a)[h(x)/x]$$

$\mathbf{T}_{19}$ is weakly rewrite-sequential. In a term of the form $f(s, t)$, in which $s \overset{\mathbf{T}_{19}}{\to} g(h(s'))$ and also $t \overset{\mathbf{T}_{19}}{\to} a$, it is always safe to rewrite t first, since by rewriting s to $g(h(s'))$ and then rewriting this to $h(s')$, we cannot eliminate the redexes in t.

Now, consider

$$\mathbf{T}_{20} = \{f(g(a, x)) \doteq f(a),\ g(x, a) \doteq a\}$$

The overlap here is of type 3, since

$$f(y)[g(x, a)[a/x]/y] = f(g(a, a)) = f(g(a, x))[a/x]$$

is the smallest substitution showing the overlap, and neither substitution is trivial. $\mathbf{T}_{20}$ is not weakly rewrite-sequential, since the term $f(g(g(a, a), g(a, a)))$ has the two rewrite sequences

$$f(g(g(a, a), g(a, a))) \overset{\mathbf{T}_{20}}{\to} f(g(a, g(a, a))) \overset{\mathbf{T}_{20}}{\to} f(a)$$

which shows that the rightmost occurrence of $g(a, a)$ is not needed and

$$f(g(g(a, a), g(a, a))) \overset{\mathbf{T}_{20}}{\to} f(g(g(a, a), a)) \overset{\mathbf{T}_{20}}{\to} f(a)$$

which shows that the leftmost occurrence of $g(a, a)$ is not needed.

Modify the previous example slightly, by changing the right-hand sides:

$$\mathbf{T}_{21} = \{f(g(a,x)) \doteq f(x),\ g(x,a) \doteq x\}$$

The result is still a type 3 overlap, but the system is weakly rewrite-sequential, since the redexes that are not needed immediately in the creation of an outer redex are preserved for later rewriting.

The positive parallel-or equations in $\mathbf{T}_{or+}$ of Examples 2.3.16 and 3.2.1 give another example of a type 3 overlap where weak rewrite-sequentiality fails. On the other hand, the negative parallel-or equations of $\mathbf{T}_{or-}$ in Example 2.3.16 have type 2 overlap, but they are sequential. In a term of the form $or(s,t)$ where $s \overset{\mathbf{T}_{or-}}{\to} \mathit{false}$ and $t \overset{\mathbf{T}_{or-}}{\to} \mathit{false}$, it is safe to rewrite either s or t first, since the redexes in the other, unrewritten, subterm are preserved for later rewriting.

Theories extending sequentiality analysis, through concepts such as weak rewrite-sequentiality, are open topics for research. I conjecture that weak rewrite-sequentiality is an undecidable property of rewrite-orthogonal systems, and that the natural concept of strong rewrite-sequentiality has essentially the same properties as strong sequentiality, except for allowing type 2 overlaps. Optimality is very subtle in these extensions, since the amount of sharing may vary depending on which of two overlapping redexes is reduced. More interesting and powerful extensions of sequentiality will require analysis of right-hand sides to deal with type 3 overlaps. Such analysis should be related in interesting ways to *strictness analysis* in functional programming languages [Myc80, Hug85b] , which detects partial strictness properties of defined functions. *Abstract interpretation* [AH87, CC77] provides a promising approach to sequentiality analysis based on right-hand sides.

Extensions of useful sequentiality analysis to systems whose confluence is established by variations on the Knuth-Bendix procedure will require the concept of residual to be generalized so that the residual of a redex α may be an arbitrarily long rewriting *sequence* used in resolving a critical pair involving α. Variations on sequentiality analysis for incremental and parallel implementations of equational logic programming are discussed in Sections 6 and 7, respectively.

4 Algorithms and Data Structures to Implement Equational Languages

The basic idea of implementing equational logic programming for strongly sequential systems is straightforward. Represent terms as linked structures with sharing, in the time-honored style of Lisp [MAE$^+$65, McC60].

At every step, find a strongly needed redex and rewrite it, halting if and when the sequence ends with a normal form. A lot of work is required to reduce these basic ideas to efficient practice. At the abstract level of algorithm and data-structure design, the problem breaks naturally into three components: a data structure to represent terms, a pattern-matching and sequencing algorithm to find strongly needed redexes, and a driving procedure to invoke the pattern-matcher/sequencer, perform the chosen rewrites, and incorporate the results into the term data structure.

4.1 Data Structures to Represent Terms

The natural data structure for terms is a linked structure in a heap, with sharing allowed. Each occurrence of a symbol f in a term is represented by a node of storage containing f and pointers to its arguments. Sharing is accomplished by allowing several different argument pointers to point to the same node. There are a number of optimizations that coalesce small nodes, or break large nodes into linked sequences, that have been explored in the literature on Lisp compilers [BC79]. In this section, we consider data structures at an abstract level with precisely one symbol per heap node, and assume that such optimizations are applied at a lower level of implementation.

4.1.1 A Conceptual Model for Term Data Structures

Some useful techniques for implementing equational logic programming require more than the linked heap structures representing terms. For example, it is sometimes better to represent the rewriting of s to t by a link from the head node of the representation of s pointing to the head node of t, rather than by an actual replacement of s by t. This representation still uses a heap, but the heap now represents a portion of the infinite rewriting graph for a starting term, rather than just a single term at some intermediate stage in rewriting to normal form. Other techniques involve the memoing of intermediate steps to avoid recomputation—these require more efficient table lookup than may be achieved with a linked heap. Fortunately, there is a single abstract data structure that subsumes all of the major proposals as special cases, and which allows a nice logical interpretation [She90]. This data structure is best understood in terms of three tables representing three special sorts of functions.

Definition 4.1.1. For each $i \geq 0$ let $\mathbf{Fun}_i$ be a countably infinite set of *function symbols of arity* i. The 0-ary function symbols in $\mathbf{Fun}_0$ are called *constant symbols.* $\mathbf{T}_P^0$ is the set of ground terms (terms without variables) constructed from the given function symbols (see Definition 2.3.1 of the chapter 'Introduction: Logic and Logic Programming Languages'). Let $\mathbf{P}$ be a countably infinite set. Members of $\mathbf{P}$ are called *parameters,* and are

Table 1. Parameter valuation representing $f(g(a, f(a, a)), f(a, a))$.

α_0	$\mapsto$	$f(\alpha_1, \alpha_2)$
α_1	$\mapsto$	$g(\alpha_3, \alpha_2)$
α_2	$\mapsto$	$f(\alpha_3, \alpha_4)$
α_3	$\mapsto$	a
α_4	$\mapsto$	a

written $\alpha, \beta, \ldots$, sometimes with subscripts. Formally, parameters behave just like the *variables* of Definition 2.3.1, but their close association with heap addresses later on makes us think of them somewhat differently.

An *i-ary signature* is a member of $\mathbf{Fun}_i \times \mathbf{P}^i$. The signature $\langle f, \langle \alpha_1, \ldots, \alpha_i \rangle \rangle$, is normally denoted by $f(\alpha_1, \ldots, \alpha_i)$. **Sig** denotes the set of signatures of all arities.

Let **nil** be a symbol distinct from all function symbols, parameters, and signatures.

A *parameter valuation* is a function $\mathbf{val} : \mathbf{P} \rightarrow \mathbf{Sig} \cup \{\mathbf{nil}\}$.

A *parameter replacement* is a function $\mathbf{repl} : \mathbf{P} \rightarrow \mathbf{P} \cup \{\mathbf{nil}\}$.

A *signature index* is a function $\mathbf{ind} : \mathbf{Sig} \rightarrow \mathbf{P} \cup \{\mathbf{nil}\}$.

A parameter valuation, parameter replacement, or signature index is *finitely based* if and only if its value is **nil** for all but a finite number of arguments.

The conventional representation of a term by a linked structure in a heap may be understood naturally as a table representing a finitely based parameter valuation. The parameters are the heap addresses, and the signatures are the possible values for data nodes. $\mathbf{val}(\alpha)$ is the signature stored at address α. But, we may also think of parameters as additional 0-ary symbols, of signatures as terms of height 1 built from parameters, and of the function **val** as a set of formulae asserting equalities between parameters and signatures. Each value $\mathbf{val}(\alpha) = f(\beta_1, \ldots, \beta_i) \neq \mathbf{nil}$ of the valuation function represents the formula $\alpha \doteq f(\beta_1, \ldots, \beta_i)$. When **val** represents the contents of a heap with the head symbol of a term t stored at address α, then $\alpha \doteq t$ is a logical consequence of the equations represented by **val**.

Example 4.1.2. Consider the finitely based parameter valuation **val** given by Table 1. All values of **val** not shown in the tables are **nil**. The linked heap structure associated with **val** is shown in Figure 5. The set of equations represented by **val** is

$$\{\alpha_0 \doteq f(\alpha_1, \alpha_2),\ \alpha_1 \doteq g(\alpha_3, \alpha_2),\ \alpha_2 \doteq f(\alpha_3, \alpha_4),\ \alpha_3 \doteq a,\ \alpha_4 \doteq a\}$$

Logical consequences of these equations include $\alpha_2 \doteq f(a, a)$, $\alpha_1 \doteq g(a, f(a, a))$,

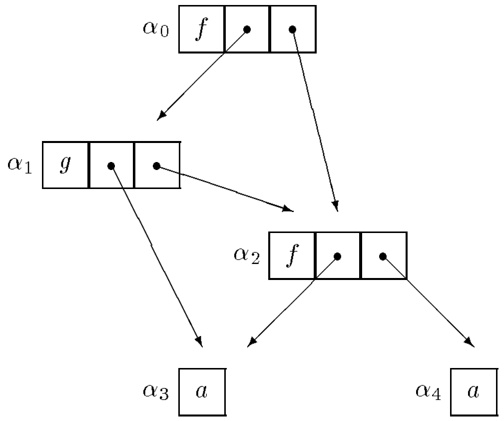


Fig. 5. Linked structure representing $f(g(a, f(a,a)), f(a,a)))$**.**

and $\alpha_0 \doteq f(g(a, f(a,a)), f(a,a))$.

It is useful to have a notation for the term represented explicitly by a parameter valuation at a particular parameter.

Definition 4.1.3. Let **val** be a parameter valuation. The partial function $\mathbf{val}^* : \mathbf{P} \to \mathbf{T}_P^0$ is defined inductively by

$$\mathbf{val}^*(\alpha) = f(\mathbf{val}^*(\beta_1), \ldots, \mathbf{val}^*(\beta_i))$$

when $\mathbf{val}(\alpha) = f(\beta_1, \ldots, \beta_i)$. Notice that the case where f is a constant symbol, and therefore $i = 0$, provides a basis for the inductive definition. If a value of **nil** is encountered anywhere in the induction, or if the induction fails to terminate because of a loop, $\mathbf{val}^*$ is undefined.

When $\mathbf{val}^*(\alpha)$ is well-defined, the equation $\alpha \doteq \mathbf{val}^*(\alpha)$ is always a logical consequence of the equations represented by **val**.

Optimized implementations of equational logic programming languages sometimes find it more efficient to link together nodes representing left- and right-hand sides of equations, rather than to actually perform rewriting steps. Such linking can be implemented by an additional pointer in each node of a heap structure. The information in these links is naturally represented by a parameter replacement function. Given a set **T** of equations, it first appears that we should think of the function **repl** as a set of formulae asserting that one term rewrites to another—that is,

$\mathbf{repl}(\alpha) = \beta \neq \mathbf{nil}$ represents the formula $\mathbf{val}^*(\alpha) \xrightarrow{\mathbf{T}}{}^* \mathbf{val}^*(\beta)$. But, further rewriting steps on subterms of the term represented by α may invalidate such a relation. There are also implementations that make efficient use of data structures for which $\mathbf{val}^*$ is ill defined. So, the proper logical interpretation of $\mathbf{repl}(\alpha) = \beta$ as a formula is merely $\alpha \doteq \beta$. An efficient implementation manipulates **val** and **repl** so that β is in some way a better starting point for further rewriting than α. The precise sense in which it is better varies among different implementations. **val** and **repl** together yield a set of terms for each parameter, all of which are known to be equal. The set $\mathbf{val}^*_{\mathbf{repl}}(\alpha)$ defined below is the set of terms that may be read by starting at α and following links in **val** and **repl**.

Definition 4.1.4. Let **val** be a parameter valuation and let **repl** be a parameter replacement. The function $\mathbf{val}^*_{\mathbf{repl}} : \mathbf{P} \to \mathbf{2}^{\mathbf{T}^0_P}$ is defined so that $\mathbf{val}^*_{\mathbf{repl}}(\alpha)$ is the least set satisfying

1. If $\mathbf{val}(\alpha) = f(\beta_1, \ldots, \beta_i)$, then

$$\mathbf{val}^*_{\mathbf{repl}}(\alpha) \supseteq \{f(t_1, \ldots, t_i) : t_1 \in \mathbf{val}^*_{\mathbf{repl}}(\beta_1) \wedge \cdots \wedge t_i \in \mathbf{val}^*_{\mathbf{repl}}(\beta_i)\}$$

2. If $\mathbf{repl}(\alpha) \neq nil$, then $\mathbf{val}^*_{\mathbf{repl}}(\alpha) \supseteq \mathbf{val}^*_{\mathbf{repl}}(\mathbf{repl}(\alpha))$

In the presence of loops, $\mathbf{val}^*_{\mathbf{repl}}(\alpha)$ may be infinite. Even without loops, its size may be exponential in the size of the data structure representing **val** and **repl**. The power of such data structures derives from this ability to represent large sets of equivalent terms compactly. When $\mathbf{val}^*(\alpha)$ is well defined, $\mathbf{val}^*(\alpha) \in \mathbf{val}^*_{\mathbf{repl}}(\alpha)$. Another special member of $\mathbf{val}^*_{\mathbf{repl}}(\alpha)$ is particularly interesting—the one reached by following **repl** links as much as possible.

Definition 4.1.5. Let **val** be a parameter valuation and let **repl** be a parameter replacement. The partial function $\mathbf{val}^{max}_{\mathbf{repl}} : \mathbf{P} \to \mathbf{T}^0_P$ is defined inductively by

$$\mathbf{val}^{max}_{\mathbf{repl}}(\alpha) = \mathbf{val}^{max}_{\mathbf{repl}}(\mathbf{repl}(\alpha))$$

when $\mathbf{repl}(\alpha) \neq \mathbf{nil}$.

$$\mathbf{val}^{max}_{\mathbf{repl}}(\alpha) = f(\mathbf{val}^{max}_{\mathbf{repl}}(\beta_1), \ldots, \mathbf{val}^{max}_{\mathbf{repl}}(\beta_i))$$

when $\mathbf{repl}(\alpha) = \mathbf{nil}$ and $\mathbf{val}(\alpha) = f(\beta_1, \ldots, \beta_i)$. As with $\mathbf{val}^*$, $\mathbf{val}^{max}_{\mathbf{repl}}$ is undefined if **nil** is encountered as a value of **val**, or if the induction fails to terminate because of a loop.

Table 2. Parameter valuation and replacement.

val			repl		
α_0	$\mapsto$	$f(\alpha_1, \alpha_3)$	α_0	$\mapsto$	α_4
α_1	$\mapsto$	$f(\alpha_2, \alpha_3)$	α_1	$\mapsto$	**nil**
α_2	$\mapsto$	a	α_2	$\mapsto$	**nil**
α_3	$\mapsto$	b	α_3	$\mapsto$	α_5
α_4	$\mapsto$	$g(\alpha_2, \alpha_3)$	α_4	$\mapsto$	**nil**
α_5	$\mapsto$	c	α_5	$\mapsto$	**nil**

Example 4.1.6. Consider the finitely based parameter valuation **val** and parameter replacement **repl** given by Table 2. All values of **val** and **repl** not shown in the tables are **nil**. These tables represent some of the consequences of the equations

$$\mathbf{T}_{22} = \{f(f(x, y), z) \doteq g(x, z),\ b \doteq c\}$$

when used to rewrite the term $f(f(a, b), b)$ The linked heap structure associated with **val** and **repl** is shown in Figure 6. The rightmost link in each node α points to $\mathbf{repl}(\alpha)$. By following links from α_0 in the table we can construct the six ground terms in $\mathbf{val}^*_{\mathbf{repl}}(\alpha_0)$: $\mathbf{val}^*(\alpha_0) = f(f(a, b), b)$, $f(f(a, b), c)$, $f(f(a, c), b)$, $f(f(a, c), c)$, $g(a, b)$, and $\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0) = g(a, c)$. Every equality between these terms is a logical consequence of $\mathbf{T}_{22}$, and all of these equalities may be read immediately from the data structure by following links from α_0.

A prime weakness of data structures based on parameter valuations and replacements is that both functions require a parameter as argument. Given a newly constructed signature, there is no direct way, other than searching through the parameter valuation, to discover whether information on that signature is already available. Signature indexes are intended precisely to allow a newly constructed signature to be translated to an equivalent parameter. While finitely based parameter valuations and replacements are normally implemented by direct memory access, using parameters as addresses, the number of possible signatures is generally too great to allow such an implementation of a finitely based signature index. General-purpose table-look-up methods are used instead, usually hash tables [Knu73]. A typical application of a hash-table representation of a signature index is the *hashed cons* optimization in Lisp [SLR78], where every newly constructed node is looked up in a hash table to see whether it already exists in the heap—if it does the existing node may be shared instead of creating another copy in the heap. The most obvious use of

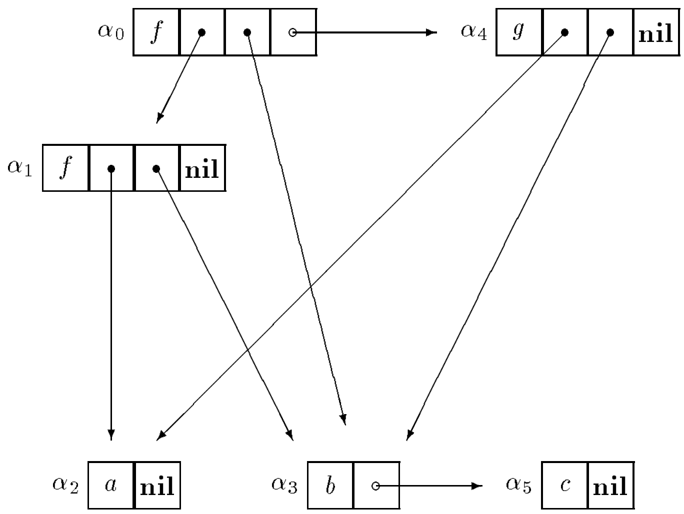


Fig. 6. Linked structure showing val and repl links.

a signature index **ind**, such as the hashed cons application, requires that whenever $\mathbf{ind}(f(\beta_1, \ldots, \beta_i)) = \alpha \neq \mathbf{nil}$, then $\mathbf{val}(\alpha) = f(\beta_1, \ldots, \beta_i)$; that is, **val** is a partial inverse to **ind**. It may be advantageous in some cases to let $\mathbf{ind}(f(\beta_1, \ldots, \beta_i))$ be a parameter known to be equal to $f(\beta_1, \ldots, \beta_i)$. The proper logical interpretation of $\mathbf{ind}(f(\beta_1, \ldots, \beta_i)) = \alpha \neq \mathbf{nil}$ is merely the formula $f(\beta_1, \ldots, \beta_i) \doteq \alpha$. So, **ind** provides the same type of logical information as **val**, but allows access to that information through a signature argument, instead of a parameter argument.

4.1.2 Logical Interpretation of Term Data Structures

Each point in the graph of a parameter valuation **val**, a parameter replacement **repl**, and a signature index **ind** represents an equation. An entire data structure consisting of finitely based functions **val**, **repl**, and **ind** represents a suitably quantified conjunction of these equations. For definiteness, suppose that inputs and outputs are rooted at the parameter α_0. Then the logical meaning of **val**, **repl**, and **ind** is the conjunction of all their equations, with all parameters *except* α_0 existentially quantified.

Definition 4.1.7. Let **val**, **repl**, **ind** be a parameter valuation, a parameter replacement, and a signature index, respectively, all three finitely based. Let $\alpha_1, \ldots, \alpha_n$ be all of the parameters β occurring in the finite basis of the domain of **val** or of **repl** ($\mathbf{val}(\beta) \neq nil$ or $\mathbf{repl}(\beta) \neq nil$), or in the range of **repl**, or as a component of a signature in the finite basis of the domain of **ind** or in the range of **val**, except for the input/output parameter α_0. The *logical interpretation* of **val**, **repl**, **ind** is the formula $F_{\mathbf{val},\mathbf{repl},\mathbf{ind}}$ defined by

$$F_{\mathbf{val},\mathbf{repl},\mathbf{ind}} = (\exists \alpha_1, \ldots \alpha_n : G)$$

where G is the conjunction of all the equations

1. $\beta \doteq f(\gamma_1, \ldots, \gamma_i)$ where $\mathbf{val}(\beta) = f(\gamma_1, \ldots, \gamma_i)$
2. $\beta \doteq \gamma$ where $\mathbf{repl}(\beta) = \gamma$
3. $f(\beta_1, \ldots, \beta_i) \doteq \gamma$ where $\mathbf{ind}(f(\beta_1, \ldots, \beta_i)) = \gamma$

Example 4.1.8. Consider the finitely based parameter valuation **val** and parameter replacement **repl** discussed in Example 4.1.6, and shown in Table 2 and Figure 6. If α_0 is the parameter used for the root of the input and output, then the logical interpretation of **val** and **repl** is

$$\begin{array}{ll} \exists \alpha_1, \ldots, \alpha_5 : & \alpha_0 \doteq f(\alpha_1, \alpha_3) \wedge \alpha_1 \doteq f(\alpha_2, \alpha_3) \wedge \alpha_2 \doteq a \wedge \alpha_3 \doteq b \wedge \\ & \alpha_4 \doteq g(\alpha_2, \alpha_3) \wedge \alpha_5 \doteq c \wedge \alpha_0 \doteq \alpha_4 \wedge \alpha_3 \doteq \alpha_5 \end{array}$$

The interpretation of parameter valuations, parameter replacements, and signature indexes as encodings of existentially quantified conjunctions of equations makes it much easier to insure correctness of proposed algorithms for manipulating data structures based on these functions. The essential idea is that a transformation of a data structure from a state representing $\mathbf{val}_0$, $\mathbf{repl}_0$, $\mathbf{ind}_0$ to one representing $\mathbf{val}_1$, $\mathbf{repl}_1$, $\mathbf{ind}_1$ in the computation of an equational program $\mathbf{T}$ is logically permissible if and only if the formula represented by $\mathbf{val}_1$, $\mathbf{repl}_1$, $\mathbf{ind}_1$ is a logical consequence of $\mathbf{T}$ plus the formula represented by $\mathbf{val}_0$, $\mathbf{repl}_0$, $\mathbf{ind}_0$. So, an evaluation algorithm may take input s, start in a state where $\mathbf{val}^{max}(\alpha_0) = s$, and apply permissible transformations to reach a state where $t \in \mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$, for some normal form t. The permissibility of the transformations guarantees that t is a correct answer.

The interpretation of the term data structures as sets of formulae applies the concepts of logic programming to the implementation of a logic programming language. A similar use of logical concepts within the implementation of a logic programming language occurs in recent presentations of *unification* algorithms (used in the implementation of Prolog) as processes that derive solutions to equations, where every intermediate step represents a new and simpler set of equations to be solved [KKN82, MM76]. The view of unification as transformation of equations to be solved clarifies the correctness of a number of clever and efficient algorithmic techniques.

Similarly, the logical view of term data structures shows immediately that a wide range of transformations of such data structures are correct, leaving an implementer free to choose the ones that appear to be most efficient. In order to represent new terms during a computation, we must be careful not to introduce spurious information about the input/output parameter α_0, nor to postulate the solvability of nontrivial equations. The concept of *reachability*, defined below, may be used to enforce both of these conditions.

Definition 4.1.9. The set of parameters *reachable* from a given parameter α is defined inductively by

1. α is reachable from α
2. If β is reachable from α, and one of the following holds as well
   (a) $\mathbf{val}(\gamma) = f(\ldots, \beta, \ldots)$
   (b) $\mathbf{val}(\beta) = f(\ldots, \gamma, \ldots)$
   (c) $\mathbf{repl}(\gamma) = \beta$
   (d) $\mathbf{repl}(\beta) = \gamma$
   (e) $\mathbf{ind}(f(\ldots, \gamma, \ldots)) = \beta$
   (f) $\mathbf{ind}(f(\ldots, \beta, \ldots)) = \gamma)$

   then γ is reachable from α

Permissible transformations include

1. Building new terms: when α is not reachable from any of $\alpha_0, \beta_1, \ldots, \beta_i$, reassign $\mathbf{val}(\alpha) := f(\beta_1, \ldots, \beta_i)$. (This allows both bottom-up and top-down construction of directed-acyclic representations of terms, using names completely unconnected to α_0. More liberal conditions than the unreachability of α from $\alpha_0, \beta_1, \ldots, \beta_i$ are possible, but they are somewhat complex to define.)
2. Representing rewriting in **repl**: when some $s \in \mathbf{val}^*_{\mathbf{repl}}(\alpha)$ rewrites to some $t \in \mathbf{val}^*_{\mathbf{repl}}(\beta)$, reassign $\mathbf{repl}(\alpha) := \beta$.
3. Rewriting in **val**: when $\mathbf{repl}(\alpha) \neq nil$, reassign $\mathbf{val}(\alpha) := \mathbf{val}(\mathbf{repl}(\alpha))$.
4. Compressing paths in **repl**: when $\mathbf{repl}(\mathbf{repl}(\alpha)) \neq nil$, reassign $\mathbf{repl}(\alpha) := \mathbf{repl}(\mathbf{repl}(\alpha))$.
5. Rewriting arguments in **val**: when $\mathbf{val}(\alpha) = f(\beta_1, \ldots, \beta_i)$ and $\mathbf{repl}(\beta_j) \neq nil$, reassign $\mathbf{val}(\alpha) := f(\beta_1, \ldots, \mathbf{repl}(\beta_j), \ldots, \beta_i)$.
6. Opportunistic sharing: when $\mathbf{ind}(\mathbf{val}(\alpha)) \neq nil, \alpha$, reassign $\mathbf{repl}(\alpha) := \mathbf{ind}(\mathbf{val}(\alpha))$

7. Indexing: when $\mathbf{val}(\alpha) = f(\beta_1, \ldots, \beta_i)$, reassign $\mathbf{ind}(f(\beta_1, \ldots, \beta_i)) := \alpha$.

8. Garbage collecting: reassign $\mathbf{val}(\alpha) := nil$, and/or $\mathbf{repl}(\alpha) := nil$, and/or $\mathbf{ind}(f(\ldots)) := nil$. (This is always *permissible,* because it only erases assertions. It is *desirable* only when the information removed is not useful for the remainder of the computation.)

A straightforward evaluator scans $\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$ to find an instance of a left-hand side s of an equation rooted at some node β, uses transformation 1 repeatedly to build a copy of the corresponding right-hand side t in free nodes rooted at γ, then links β to γ by transformation 2. Transformations 3–5 allow the same term $\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$ to be scanned more efficiently, by reducing the number of **repl** links that must be followed. Transformations 6 and 7 are used for optimizations such as *hashed cons, congruence closure,* and *memo functions* to avoid re-evaluating the same subterm when it is constructed repeatedly in a computation. Section 4.3 discusses several optimizations using transformations 3–7.

The existential quantification of all parameters other than α_0 is required for the logical correctness of transformation 1 (term building), which changes the meaning of some existentially quantified parameter, without affecting any assertion about α_0. 2–8 are permissible independent of the quantification. Notice that only transformation 2 depends on the given equational program, and only 2 adds information to the data structure. 1 is needed to build new terms required by 2, but by itself 1 does not change the logical assertions about α_0. 3–8 can only *reduce* the information given by **val** and **repl**, but they may improve the efficiency of access to information that is retained. 3–7 never change $\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$. Transformation 8 is normally applied only to nodes that are inaccessible from α_0, in which case 8 also preserves $\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$.

The logical interpretation introduced in Definition 4.1.7 seems to be the most natural one for explaining currently known techniques for implementing equational logic programming, but others are possible, and might lead to useful extensions of, or variations on, the current sort of equational logic programming. Let $\alpha_1, \ldots, \alpha_n$ be all of the parameters used in $\mathbf{val}, \mathbf{repl}, \mathbf{ind}$, other than the input/output parameter α_0, and let G be the conjunction of all the equations represented by points in the graphs of $\mathbf{val}, \mathbf{repl}, \mathbf{ind}$, just as in Definition 4.1.7. Two other interesting logical interpretations worthy of study are

1. $(\exists \alpha_0, \alpha_1, \ldots, \alpha_n : G) \Rightarrow (\exists \alpha_1, \ldots, \alpha_n : G)$

2. $\forall \alpha_0, \alpha_1, \ldots, \alpha_n : (G \Rightarrow (input \doteq \alpha_0))$, where $input$ is a new zeroary symbol

The first proposal above allows introduction of arbitrary structures not connected to the input/output variable α_0. The second one allows for solution of equations, where every solution to the output is guaranteed to be a solution to the input problem as well.

4.2 Pattern-Matching and Sequencing Methods

Given a representation of a term, the implementation of an equational logic program must perform *pattern-matching* to discover instances of left-hand sides of equations that may be rewritten, and must apply *sequencing* techniques to determine which of several such redexes to rewrite. These two conceptual tasks appear to be inextricably connected, so it is best to provide a single procedure to do both—that is, to find the next appropriate redex to rewrite in a given term. In order to find and choose a redex in a term, a procedure traverses the term until it has gathered enough information about the symbols in the term to make its choice. The full details of pattern-matching and sequencing methods are too long for this chapter. So, I describe two basic approaches to the problem and illustrate them by examples. The reader who needs more detail should consult [HL91, HO79, HOS85, O'D85, Klo91, KM91].

4.2.1 Representing Sequencing Information by Ω-Terms

A natural way to represent partial information about a term is to present the known structure as a term, with a special symbol (Ω) representing unknown portions.

Definition 4.2.1. The set $\mathbf{T}_P^\Omega$ of Ω-*terms* is defined in the same way as the set $\mathbf{T}_P$ of terms (Definition 2.3.1 of the chapter 'Introducation: Logic and Logic Programming Languages'), except the new constant Ω is added to the set $\mathbf{Fun}_0$ of symbols of arity 0.

An Ω-term s is *less defined than* an Ω-term t, written $s \sqsubseteq t$, if and only if s is the result of replacing zero or more subterms of t by Ω.

$s \sqcap t$ denotes the greatest lower bound of s and t.

When Ω-terms s and t have a common upper bound (that is, when there is a u such that $s \sqsubseteq u$ and $t \sqsubseteq u$), we write $s \uparrow t$.

The Ω in an Ω-term behaves formally much like a variable, except that each occurrence of the same symbol Ω is treated as a *unique* variable, occurring only at that location.

There is an elegant and simple procedure, called *melting,* for computing all of the possible effects of ω-rewriting on an Ω-term.

Definition 4.2.2. The function $\mathbf{patt} : \mathbf{T}_P \to \mathbf{T}_P^\Omega$ is defined by

$$\mathbf{patt}(t) = t[\Omega, \ldots, \Omega/x_1, \ldots, x_m]$$

where $x_1, \ldots, x_m$ are all of the variables occurring in t.

Extend **patt** to sets of equations $\mathbf{T} = \{l_1 \doteq r_1, \ldots, l_n \doteq r_n\}$ by

$$\mathbf{patt}(\mathbf{T}) = \{\mathbf{patt}(l_1), \ldots, \mathbf{patt}(l_n)\}$$

Given a set $\mathbf{T}$ of equations, and an Ω-term $s \in \mathbf{T}_P^\Omega$, s is transformed into $\mathbf{melt}_\omega^\mathbf{T}(s)$ by the following extremely simple polynomial time procedure:

1. If $s = t[u/x]$, where $u \neq \Omega$ and $u \uparrow p$ for some $p \in \mathbf{patt}(\mathbf{T})$, then replace s by $t[\Omega/x]$.
2. Repeat (1) above until it is not applicable.

If s is an Ω-term representing current information about a term t that we are rewriting to normal form, then $\mathbf{melt}_\omega^\mathbf{T}(s)$ represents precisely the information in s that is guaranteed to hold for all t' such that $t \xrightarrow[\omega]{\mathbf{T}}{}^* t'$. By marking an occurrence with a new inert symbol, and melting, we can determine whether ω-rewriting may eliminate that occurrence without rewriting it—that is, we determine whether the occurrence is strongly needed.

Proposition 4.2.3. *Let s be an Ω-term, and let α be an occurrence in s. Let s' be the result of replacing α in s with a new constant symbol, $\bullet$. α is strongly needed if and only if $\mathbf{melt}_\omega^\mathbf{T}(s)$ contains an occurrence of $\bullet$.*

Huet and Lévy [HL91] propose a pattern-matching and sequencing technique that accumulates an Ω-term s_α (initially just Ω), representing information about the subterm at a particular node α in a term t. At each step, they choose a strongly needed occurrence of Ω in s_α, and extend s_α by reading the symbol at the corresponding subterm occurrence in t. The information in s_α may be used to control pattern-matching and sequencing as follows:

1. If there is a pattern $p \in \mathbf{patt}(\mathbf{T})$ such that $p \sqsubseteq s_\alpha$, then α is a redex occurrence (in strongly sequential systems we always get $s_\alpha = p$, but it is better to think of the more general case).
2. If there is no pattern $p \in \mathbf{patt}(\mathbf{T})$ such that $s_\alpha \uparrow p$, then α is not currently a redex occurrence.
3. If $\mathbf{melt}_\omega^\mathbf{T}(s_\alpha) \neq \Omega$, then α will never be a redex occurrence.
4. It is safe to query the symbol in t corresponding to a strongly needed occurrence of Ω in s_α, and to rewrite any redex occurring there, since it must be strongly needed.

In cases 2 and 3, Huet and Lévy move to a proper descendant α' of α corresponding to the largest subterm s' of s_α, containing the newly read symbol, such that $s' \uparrow p$ for some $p \in \mathbf{patt}(\mathbf{T})$ (because of strong sequentiality, in

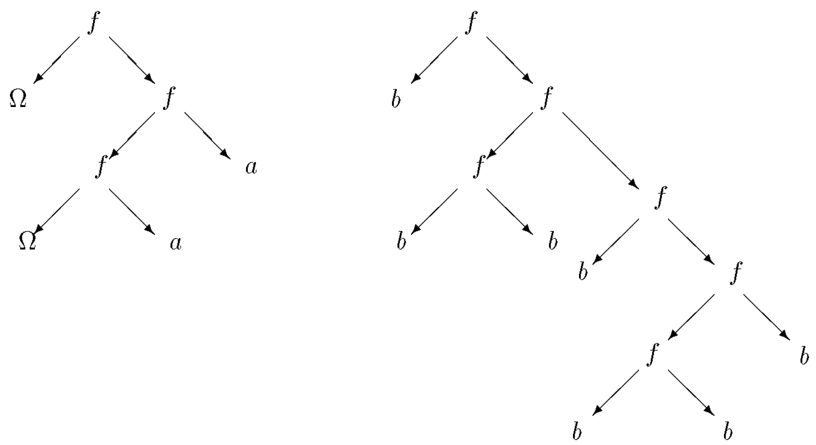


Fig. 7. Pictures of terms for sequencing example

fact $s' \sqsubseteq p$). Then they let $s_{\alpha'} = s'$ and proceed at α'. In case 1, the fact that we have reached α by safe moves implies that the redex is strongly needed. So, they rewrite, and continue processing at an appropriate ancestor of α, rereading whatever symbols have been changed by rewriting at α.

Example 4.2.4. Consider the strongly sequential system

$$\mathbf{T}_{23} = \{f(x, f(f(y, a), a)) \doteq a,\ b \doteq a\}$$

$$\mathbf{patt}(\mathbf{T}_{23}) = \{f(\Omega, f(f(\Omega, a), a)),\ b\}$$

and search for a strongly needed redex in the term

$$t = f(b, f(f(b, b), f(b, f(f(b, b), b))))$$

with occurrences Λ (the root), 1 (leftmost b), 2 ($f(f(b, b), f(b, f(f(b, b), b)))$), 2.1 (leftmost $f(b, b)$), 2.1.1, 2.1.2 (second and third bs), 2.2 ($f(b, f(f(b, b), b))$), 2.2.1 (fourth b), 2.2.2 ($f(f(b, b), b)$), 2.2.2.1 (rightmost $f(b, b)$), 2.2.2.1.1, 2.2.2.1.2, 2.2.2.2 (fifth, sixth and seventh bs). The pattern $f(\Omega, f(f(\Omega, a), a))$, and the term t, are shown pictorially in Figure 7. The occurrence of Ω chosen for expansion at each step is underlined, and the subterm replacing it is shown in a box.

- Start at the root of t, with $s_\Lambda = \underline{\Omega}$.

- ∗ There is only one choice, so read the f at the root, and expand to $s_\Lambda = \boxed{f(\Omega, \underline{\Omega})}$.
- ∗ Only the rightmost Ω is strongly needed, so read the corresponding symbol, and expand to $s_\Lambda = f(\Omega, \boxed{f(\Omega, \underline{\Omega})})$.
- ∗ Only the rightmost Ω is strongly needed, so expand to $s_\Lambda = f(\Omega, f(\Omega, \boxed{f(\Omega, \Omega)}))$.
- ∗ The new Ω-term is incompatible with left-hand side patterns (the f conflicts with an a in the pattern), so we find the largest subterm containing the box that is compatible with a pattern. That subterm is the second principal subterm at 2, $s' = f(\Omega, \boxed{f(\Omega, \Omega)})$, so initialize s_2 to s' and continue at 2.

- Now process the second principal subterm of t, $t_2 = f(f(b, b), f(b, f(f(b, b), b)))$, with $s_2 = f(\Omega, f(\Omega, \underline{\Omega}))$.
  - ∗ Only the rightmost Ω is strongly needed, so expand to $s_2 = f(\Omega, f(\Omega, \boxed{f(\Omega, \Omega)}))$.
  - ∗ The new Ω-term is incompatible with the patterns (the f conflicts with an a in the pattern), so we find $s' = f(\Omega, f(\Omega, \Omega))$, initialize $s_{2.2}$ to s', and continue at 2.2.
- Now process the subterm $t_{2.2} = f(b, f(f(b, b), b))$ at 2.2, with $s_{2.2} = f(\Omega, f(\Omega, \underline{\Omega}))$.
  - ∗ Only the rightmost Ω is strongly needed, so expand to $s_{2.2} = f(\Omega, \boxed{f(\Omega, \underline{\Omega})})$.
  - ∗ Only the rightmost Ω is strongly needed, so expand to $s_{2.2} = f(\Omega, f(\Omega, \boxed{b}))$.
  - ∗ The new Ω-term is incompatible with the patterns (the b conflicts with an a in the pattern), so we find $s' = b$, initialize $s_{2.2.2.2}$ to s', and continue at 2.2.2.2.
- Now process the subterm $t_{2.2.2.2} = b$ at 2.2.2.2, with $s_{2.2.2.2} = b$. $b \sqsubseteq s_{2.2.2.2}$, and $b \in \mathbf{patt}(\mathbf{T}_{23})$, so this subterm is a strongly needed redex. Rewrite it to a, yielding

$$t = f(b, f(f(b, b), f(b, f(f(b, b), a))))$$

  and continue at 2.2, using the last value for $s_{2.2}$ before reading 2.2.2.2: $s_{2.2} = f(\Omega, f(\Omega, \underline{\Omega}))$.
  - ∗ Expand the fourth Ω again, but this time we read an a instead of a b, yielding $s_{2.2} = f(\Omega, f(\underline{\Omega}, \boxed{a}))$.
  - ∗ Only the rightmost Ω is strongly needed, so expand to $s_{2.2} = f(\Omega, f(\boxed{f(\Omega, \underline{\Omega})}, a))$.

  * Only the rightmost Ω is strongly needed, so expand to $s_{2.2} = f(\Omega, f(f(\Omega, \boxed{b}), a))$.
  * Again the new Ω-term is incompatible with the patterns, and further processing at 2.2.2.1.2 rewrites the b to a, yielding

    $$t = f(b, f(f(b, b), f(b, f(f(b, a), a))))$$

    We continue at 2.2, with the last version of $s_{2.2}$ before reading 2.2.2.1.2. Extend $s_{2.2} = f(\Omega, f(f(\Omega, \underline{\Omega}), a))$ again, this time to $s_{2.2} = f(\Omega, f(f(\Omega, \boxed{a}), a))$.
  * Now $f(\Omega, f(f(\Omega, a), a)) \sqsubseteq s_{2.2}$, and $f(\Omega, f(f(\Omega, a), a)) \in \mathbf{patt}(\mathbf{T}_{23})$, so we have a strongly needed redex, which we rewrite to a, yielding

    $$t = f(b, f(f(b, b), a))$$

- Continue at 2 with the last Ω-term before reading 2.2: $s_2 = f(\Omega, \underline{\Omega})$.

  * Expand the rightmost Ω again, but this time we read an a instead of an f, yielding $s_2 = f(\Omega, \boxed{a})$.
  * The new Ω-term is incompatible with the patterns, and there is no compatible subterm, so we move back toward the root.

- Continue at Λ with the last Ω term before reading 2.2: $s_\Lambda = f(\Omega, f(\Omega, \underline{\Omega}))$.

  * This time we read an a instead of an f, yielding: $s_\Lambda = f(\Omega, f(\underline{\Omega}, \boxed{a}))$.
  * Only the rightmost Ω is strongly needed, so expand to $s_\Lambda = f(\Omega, f(\boxed{f(\Omega, \underline{\Omega})}, a))$.
  * Only the rightmost Ω is strongly needed, so expand to $s_\Lambda = f(\Omega, f(f(\Omega, \boxed{b}), a))$.
  * As before, the Ω-term is incompatible with the patterns; further processing at 2.1.2 discovers that b is a redex, and rewrites it to a, yielding

    $$t = f(b, f(b, a), a)$$

    Then we continue at Λ with $s_\Lambda = f(\Omega, f(f(\Omega, \underline{\Omega}), a))$ again, this time reading an a and extending to $s_\Lambda = f(\Omega, f(f(\Omega, \boxed{a}), a))$.
  * Now we have a redex pattern $f(\Omega, f(f(\Omega, a), a))$ at the root. We rewrite to a and start over at the root.

- In general, there might be more computation at the root, but in this example we immediately get $s_\Lambda = a$, which is incompatible with the patterns. Since there is nowhere left to go, we have a normal form.

Huet and Lévy use Ω-terms in a pattern-matching and sequentializing method that succeeds for every strongly sequential system of equations. Several similar but less general methods have been proposed in order to simplify the method in special cases [O'D85, Str88, Str89, DS93, Dur94, RS90].

4.2.2 Representing Sequencing Information by Sets of Subpatterns

Another natural approach to pattern-matching and sequencing is to focus on the left-hand sides of equations, and represent information about a term according to the ways that it does and does not match portions of those left-hand sides.

Definition 4.2.5. Let $\mathbf{T}$ be a set of equations. $\mathbf{U_T}$ is the set of all subterms of members of $\mathbf{patt}(\mathbf{T})$.

A *subpattern set* for $\mathbf{T}$ is a subset $B \subseteq \mathbf{U_T}$.

Subpattern sets may be used to present information about pattern-matching and sequencing in several ways:

1. A *match set* is a subpattern set containing all of the subpatterns known to match at a particular node in a term.
2. A *possibility set* is a subpattern set containing all of the subpatterns that might match at a particular node in a term, as the result of ω-rewriting the proper subterms of that node.
3. A *search set* is a subpattern set containing subpatterns to search for at a particular node, in order to contribute to a redex.

Notice that match sets and possibility sets always contain Ω (except in the unusual case when $\mathbf{T}$ has no occurrence of a variable), because everything matches Ω. Search sets never contain Ω, since it is pointless to search for something that is everywhere; they always contain $\mathbf{patt}(\mathbf{T})$, since finding an entire redex occurrence is always useful.

In order to control pattern-matching and sequencing with subpattern sets, associate a match set M_α and a possibility set P_α with each node in a term t. Initially, $M_\alpha = \{\Omega\}$ and $P_\alpha = \mathbf{U_T}$ for all α. At all times the subterm at α will match every subpattern in M_α, but no subpattern that is not in P_α. That is, $[M, P]$ is an interval in the lattice of subpattern sets in which the true current description of the subterm at α always resides. At every visit to a node α, update M_α and P_α based on the symbol f at α and the information at the children $\alpha_1, \ldots, \alpha_n$ of α as follows:

$$M_\alpha \;:=\; M_\alpha \cup$$
$$\{f(m_1, \ldots, m_n) \in \mathbf{U_T} : m_1 \sqsubseteq m'_1, \ldots, m_n \sqsubseteq m'_n \text{ for some } m'_1 \in M_{\alpha_1}, \ldots, m'_n \in M_{\alpha_n}\}$$

$$P'_\alpha \;:=\; P_\alpha \cap$$
$$(\{f(p_1, \ldots, p_n) \in \mathbf{U_T} : p_1 \sqsubseteq p'_1, \ldots, p_n \sqsubseteq p'_n \text{ for some } p'_1 \in P_{\alpha_1}, \ldots, p'_n \in P_{\alpha_n}\} \cup \{\Omega\})$$

$$P_\alpha \;:=\; \begin{cases} P'_\alpha & \text{if } P'_\alpha \cap \mathbf{patt}(\mathbf{T}) = \emptyset \\ \mathbf{U_T} & \text{otherwise} \end{cases}$$

P'_α represents the set of all subpatterns that may match at node α, as the result of ω-rewriting at *proper* descendants of α. Notice the similarity between the calculation of P_α and the process of melting—$\mathbf{U_T}$ above plays the role of Ω in melting.

Similarly, keep a search set S_α at each node that is visited in the traversal, entering at the root with $S_\Lambda = \mathbf{patt}(\mathbf{T})$. When moving from a node α to the ith child $\alpha.i$ of α, update the search set at $\alpha.i$ by

$$\begin{aligned} S_{\alpha.i} \ := \ & \mathbf{patt}(\mathbf{T}) \cup \\ & \{s : s \text{ is the } i\text{th principal subterm of a member of } S_\alpha \cap P'_\alpha\} \end{aligned}$$

The information in M_α, P_α, S_α, and the corresponding information at the children $\alpha_1, \ldots \alpha_n$ of α, may be used to control pattern-matching and sequencing as follows:

1. If $M_\alpha \cap \mathbf{patt}(\mathbf{T}) \neq \emptyset$, then α is a redex occurrence.
2. If $P_\alpha \cap \mathbf{patt}(\mathbf{T}) = \emptyset$, then α will never be a redex occurrence.
3. If $M_\alpha \cap S_\alpha \neq \emptyset$, then α contributes to a possible redex at one of its ancestors.
4. If $P_\alpha \cap S_\alpha = \emptyset$, then the symbol at α will never contribute to a redex occurrence.
5. If there is no Ω-term $s \in S_\alpha$ whose ith principal subterm is in $M_{\alpha.i}$, then it is safe to move to $\alpha.i$ and process it further—in particular, any redex occurrence reached by safe moves is strongly needed.

Hoffmann and I proposed a pattern-matching and sequencing method based on Match/Possibility/Search sets. The method appears heuristically to succeed on most naturally defined orthogonal systems, but Jiefei Hong noticed that there are some strongly sequential systems for which it fails.

Example 4.2.6. Consider the strongly sequential system

$$\mathbf{T}_{24} = \{f(g(a,x),a) \doteq d,\ f(g(x,a),b) \doteq e,\ c \doteq a\}$$

and search for a strongly needed redex in the term $f(g(c,c),c)$ with occurrences Λ (the root), 1 (the subterm $g(c,c)$), 1.1, 1.2, and 2 (the three occurrences of c, from left to right).

- Initially, we search at Λ, with

$$S_\Lambda = \mathbf{patt}(\mathbf{T}_{24}) = \{f(g(a,\Omega),a),\ f(g(\Omega,a),b),\ c\} \qquad M_\Lambda = \{\Omega\}$$

$$P_\Lambda = \mathbf{U}_{\mathbf{T}_{24}} = \{f(g(a,\Omega),a),\ f(g(\Omega,a),b),\ c,\ g(a,\Omega),\ g(\Omega,a),\ a,\ b,\ \Omega\}$$

Update M_Λ and P_Λ based on $M_1 = M_2 = \{\Omega\}$, $P_1 = P_2 = \mathbf{U}_{\mathbf{T}_{24}}$, and the symbol f at the root—in this case M_Λ and P_Λ do not change, and

$$P'_\Lambda = \{f(g(a,\Omega),a),\ f(g(\Omega,a),b),\ \Omega\}$$

From condition (5) above, it is safe to move to either of 1 or 2.

Suppose that we choose arbitrarily to move to 1.

- Now, we search at 1, with

$$S_1 = \{f(g(a,\Omega),a),\ f(g(\Omega,a),b),\ c,\ g(a,\Omega),\ g(\Omega,a)\}$$

$$M_1 = \{\Omega\} \qquad\qquad P_1 = \mathbf{U}_{\mathbf{T}_{24}}$$

Update M_1 and P_1 to

$$M_1 = \{\Omega\} \quad P_1 = P'_1 = \{g(a,\Omega),\ g(\Omega,a),\ \Omega\}$$

The possibility set decreases in this case, because there is no left-hand side with g at the root. Now by condition (5), it is *not* safe to move to the first child of 1, 1.1, because $g(\Omega,a) \in S_1$ has Ω as its first principal subterm, and $\Omega \in M_{1.1}$. Similarly, it is not safe to move to 1.2, because of $g(a,\Omega) \in S_1$. The method suggested in [HO79] is not capable of backing up to parents and siblings of 1 in order to decide which of the children of 1 to visit first.

On the other hand, suppose we choose arbitrarily to move from Λ to 2 instead of 1.

- Now, we search at 2, with

$$S_2 = \{f(g(a,\Omega),a),\ f(g(\Omega,a),b),\ c,\ a,\ b\}$$

$$M_2 = \{\Omega\} \qquad\qquad P_2 = \mathbf{U}_{\mathbf{T}}$$

Update M_2 and P_2 to

$$M_2 = \{c,\ \Omega\} \quad P'_2 = \{c,\Omega\} \quad P_2 = \mathbf{U}_{\mathbf{T}}$$

Since $c \in M_2 \cap \mathbf{patt}(\mathbf{T})$, there is a redex, which is strongly needed since we found it by safe moves, and we rewrite c to a, yielding $f(g(c,c),a)$. Recalculate M_2 and P_2 to

$$M_2 = \{a,\ \Omega\} \quad P_2 = P'_2 = \{a,\ \Omega\}$$

$a \in M_2 \cap S_2$, so continue back at Λ trying to complete a redex using the a at 2.

- Search again at Λ; update M_Λ, P'_Λ and P_Λ again. M_Λ and P_Λ do not change, but now
$$P'_\Lambda = \{f(g(a,\Omega),a),\ \Omega\}$$
It is still safe to move to 1.

- Search at 1 with
$$S_1 = \{f(g(a,\Omega),a),\ f(g(\Omega,a),b),\ c,\ g(a,\Omega)\}$$
$$M_1 = \{\Omega\} \qquad\qquad P_1 = \mathbf{U_T}$$
Notice that information from 2 has led to a smaller search set, not containing $g(\Omega,a)$. Update M_1 and P_1 to
$$M_1 = \{\Omega\} \quad P_1 = P'_1 = \{g(a,\Omega),\ g(\Omega,a),\ \Omega\}$$
This time it is safe to move to 1.1, but not to 1.2, so we choose the former.

- Search at 1.1 with
$$S_{1.1} = \{f(g(a,\Omega),a),\ f(g(\Omega,a),b),\ c,\ a\}$$
$$M_{1.1} = \{\Omega\} \qquad\qquad P_{1.1} = \mathbf{U_T}$$
The results are the same as in the search at 2 above—c rewrites to a and we continue back at 1.

- Search again at 1, update M_1, P_1 again to
$$M_1 = \{g(a,\Omega),\ \Omega\} \quad P_1 = P'_1 = \{g(a,\Omega),\ \Omega\}$$
$g(a,\Omega) \in M_1 \cap S_1$, so continue back at Λ

- Search one more time at Λ, recalculating M_Λ, P_Λ to
$$M_\Lambda = \{f(g(a,\Omega),a),\ \Omega\} \quad P'_\Lambda = \{f(g(a,\Omega),a),\ \Omega\} \quad P_\Lambda = \mathbf{U_T}$$
$f(g(a,\Omega),a) \in M_\Lambda \cap \mathbf{patt(T)}$, so there is a redex at Λ. Rewrite it, yielding d. Update M_Λ, P_Λ to
$$M_\Lambda = \{\Omega\} \quad P_\Lambda = P'_\Lambda = \{\Omega\}$$
$P_\Lambda \cap \mathbf{patt(T)} = \emptyset$, so the d at Λ will never change. Since Λ now has no children, we are done, and d is a normal form.

$\mathbf{T}_{24}$ shows that for some strongly sequential systems, the success of the subpattern set analysis depends on the choice of traversal order. In the strongly sequential system

$$\begin{aligned} \mathbf{T}_{25} \quad = \quad & \{f(g(a,x), g(a,x)) \doteq b,\ f(g(x,a), h(a,x)) \doteq c, \\ & f(h(a,x), g(x,a)) \doteq d,\ f(h(x,a), h(x,a)) \doteq e\} \end{aligned}$$

no order of traversal allows the subpattern set analysis to succeed. The only way to sequence with $\mathbf{T}_{25}$ is to visit *both* of the children of an f before descending to grandchildren. The Huet-Lévy method allows this breadth-first behavior, but the Hoffmann-O'Donnell method does not.

4.2.3 Applying the Sequencing Techniques

Neither Ω-terms nor subpattern sets should be computed explicitly at run time. Rather, at compile time we compute the finite set of values that can possibly occur while executing a given system of equations $\mathbf{T}$, along with tables of the operations required to update them at run time. It is natural to think of Ω-terms and subpattern sets as components of the state of a finite automaton, and the tables of operations as a representation of the transition graph. A key problem is that the number of states may grow exponentially in the size of $\mathbf{T}$. [HO82] analyzes the number of match sets associated with a set of patterns. There is no published analysis of the number of Ω-terms, nor of possibility and search sets, but it appears that all three may grow at an exponential rate also. Most implementations of equational logic programming appear to use methods that are equivalent to highly restricted forms of the Ω-term method.

The Huet-Lévy Ω-term method is the only published method that succeeds for all strongly sequential systems. The only published method using subpatterns fails in cases, such as $\mathbf{T}_{25}$, where it is necessary to back up before reaching the leaves of a pattern, in order to carry information about sequencing to a sibling. I conjecture that a simple variation in the algorithm of [HO79] will succeed on all strongly sequential systems. Besides the superior generality of the Huet-Lévy algorithm, Ω-term methods have the advantage of a simple notation, making it relatively easy to discover restrictions that control the number of possible values. On the other hand, subpattern sets express the significance of information for pattern matching purposes in a particularly transparent way. The subpattern set method also has the advantage of separating clearly the information that is passed up the tree (match and possibility sets) from the information that is passed down the tree (search sets). Match and possibility sets may be associated with nodes in a heap representation of a tree, taking advantage of sharing in the heap. Search sets and Ω-terms depend on the path by which a node was reached, so they must be stored on a traversal stack, or otherwise marked so that they are not shared inappropriately.

4.3 Driving Procedures for Term Rewriting

4.3.1 A Recursive Schema for Lazy Evaluation

It is natural to drive the conventional strict evaluation of a term by a recursive procedure, *eval*, shown in Figure 8 below. For each function symbol f that appears in a term, there is a predefined procedure $\bar{f}$ to compute the value of that function on given arguments. The simplicity of recursive

Procedure $eval(t)$
    Let $t = f(t_1, \ldots, t_n)$; (1)
    For $i := 1, \ldots, n$ **do** (2)
        $v_i := eval(t_i)$ (3)
    end for;
    Return $\bar{f}(v_1, \ldots, v_n)$ (4)
end procedure *eval*

Fig. 8. Recursive schema for strict evaluation.

evaluation is appealing, and it has a direct and transparent correspondence to the semantic definition of the value of a term (Definition 2.3.3 of the chapter 'Introduction: Logic and Logic Programming Languages). It is straightforward to let the value of a term be its normal form, and use the recursive *eval* above to find the normal form of its input. But, the structure of the procedure is heavily biased toward strict evaluation. Even the conditional function *cond* requires a special test between lines (1) and (2) to avoid evaluating both branches. Lazy functional languages have been implemented with conventional recursive evaluation, by adding new values, called *suspensions, thunks,* or *closures* to encode unevaluated subterms [Pey87, BHY64]. The overhead of naive implementations of suspensions led to early skepticism about the performance of lazy evaluation. Clever optimizations of suspensions have improved this overhead substantially, but it is still arguably better to start with an evaluation schema that deals with lazy evaluation directly.

For orthogonal systems of equations, a function symbol that contributes to a redex occurrence at a proper ancestor cannot itself be at the root of a redex. So, a term may be normalized by a recursive procedure that rewrites its argument only until the root symbol can never be rewritten.

Definition 4.3.1. Let $\mathbf{T}$ be a set of equations, and t be a term. t is a *(weak) head-normal form* for $\mathbf{T}$ if and only if there is no redex u such that $t \xrightarrow{\mathbf{T}}{}^* u$.

t is a *strong head-normal form* for $\mathbf{T}$ if and only if there is no redex u such that $t \xrightarrow[\omega]{\mathbf{T}}{}^* u$.

Head normality is undecidable, but strong head normality is easy to detect by melting.

Proposition 4.3.2. *Let t be a term. t is a strong head-normal form for $\boldsymbol{T}$ if and only if $\mathbf{melt}^{\boldsymbol{T}}_{\omega}(t) \neq \Omega$.*

Let t be an Ω-term. The folowing are equivalent:

- *u is a strong head-normal form for $\boldsymbol{T}$, for all $u \sqsupseteq t$*
- *$\mathbf{melt}^{\boldsymbol{T}}_{\omega}(t) \neq \Omega$.*

The procedure *head-eval* in Figure 9 below rewrites its argument only to strong head-normal form, instead of all the way to normal form. The

Procedure *head-eval*(t)
    While t is not in strong head-normal form **do** (1)
        If t is a redex **then** (2)
            $t :=$ *right-side*(t) (3)
        else
            Let $t = f(t_1, \ldots, t_n)$; (4)
            Choose $i \in [1, n]$ that is safe to process; (5)
            $t_i :=$ *head-eval*(t_i); (6)
            $t := f(t_1, \ldots, t_n)$ (7)
        end if
    end while;
    Return t (8)
end procedure *head-eval*

Fig. 9. Recursive schema to find head-normal form.

tests whether t is in strong head-normal form (1), whether t is a redex (2), and the choice of a safe child i (5), may be accomplished by any pattern-matching and sequencing method that succeeds on the given system of equations. The procedure *right-side* called by *head-eval* above builds and returns the right-hand side instance corresponding to a given redex. *right-side* contains the information about function symbols analogous to that contained in the collection of $\bar{f}$s used by the strict evaluator *eval*. In a detailed implementation, pattern-matching and sequencing require some additional data structures, and possibly some additional values returned by *head-eval*.

In order to produce a normal form for t, *head-eval* must be called recursively on the subterms of t, as shown by the procedure *norm* in Figure 10 below. Section 7 shows how other schemes for calling *head-eval*, more so-

Procedure *norm*(t)
  $t := head\text{-}eval(t)$; (1)
  Let $t = f(t_1, \ldots, t_n)$; (2)
  For $i := 1, \ldots, n$ **do** (3)
    $t_i := norm(t_i)$ (4)
  end for;
  $t := f(t_1, \ldots, t_n)$; (5)
  Return t (6)
end procedure *norm*

Fig. 10. Recursive schema to normalize a term lazily.

phisticated than *norm*, may be used to generalize the behavior of input and output in equational logic programming.

4.3.2 *Using the Term Data Structure in Rewriting*

The basic recursive schema of *head-eval* may manipulate the tables **val**, **repl** and **ind** in a variety of different ways, with various advantages and disadvantages in performance. In most cases, particularly in implementations of orthogonal systems of equations, the pattern-matching and sequencing method operates on $\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$, where α_0 is a heap address that initially represents the root of the input term. $\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$ is typically the closest thing to a normal form for the original input that can be read easily from **val**, **repl** and **ind**, so we achieve the best progress by rewriting it further. It is also clearly correct to run the pattern-matcher/sequencer on another term in $\mathbf{val}^{*}_{\mathbf{repl}}(\alpha_0)$. I am not aware of any current implementation that does this, but it is likely to be neccessary in complete implementations of nonconfluent systems, and equation solving systems (Section 7.2), which will presumably need to explore several rewriting sequences in parallel. The details of access to **val** and **repl** for pattern-matching/sequencing fit into the implementation of the conditions tested in lines (1) and (2) of *head-eval*, and in the choice of i in line (5). If the calculation of $\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$ follows **repl** links, it is usually desirable to compress the paths using transformations (4) and (5) from Section 4.1.2, which speeds up future calculations of $\mathbf{val}_{\mathbf{repl}}^{max}(\alpha_0)$. The benefits of transformations (4) and (5) are essentially the same as the benefits of path compression in the UNION/FIND algorithm [AHU74] (think of **repl** as representing a partition of terms into equivalence classes of terms that have been proved equal).

The implementation of *right-side*, called in line (3) of *head-eval*, is the only place where updating of **val**, **repl** and **ind** is required. Suppose that we are applying the equation $l \doteq r$ to rewrite the term t, represented in the heap with root node α. The most obvious implementation of *right-side* builds all of the nonvariable structure of r in newly allocated heap nodes in **val**, using pointers to pre-existing structure for the variable substitutions (this is essentially what a conventional Lisp implementation does). Let β be the root node of the constructed representation of the appropriate instance of r. Once the instance of r is built, β may be copied into α. Unfortunately, when r consists of a single variable, as in $car(cons(x, y)) \doteq x$, β may have already been shared extensively, and this copying loses an obvious opportunity for further sharing [O'D77]. The natural alternative to copying β is to assign $\mathbf{repl}(\alpha) := \beta$ (represent the rewriting step by a link, instead of rewriting in place). This avoids any loss of sharing, and if path compression is applied in calculating $\mathbf{val}_{\mathbf{repl}}^{max}(\alpha_0)$, it appears to have an acceptable cost in overhead for access to the heap. In many programs, a more sophisticated implementation of *right-side* can save an exponentially growing amount of wasted work by sharing identical subterms of r. Many natural programs have such identical subterms of right-had sides, and in practice this makes the difference between efficiency and infeasibility often enough to be well worth the effort. The cost of detecting identical subterms of right-hand sides using the tree isomorphism algorithm [AHU74] is quite modest, and can be borne entirely at compile time.

4.3.3 Dynamic Exploitation of Sharing and Memoing of Ground Terms

There are several ways that the signature index **ind** may be used to improve sharing dynamically and opportunistically at run time (these are examples of transformations (6) and (7) of Section 4.1.2). A further sophistication of *right-side* builds the structure of the instance of r from the leaves and variable instances up, and for each constructed signature $f(\alpha_1, \ldots, \alpha_n)$ it checks $\mathbf{ind}(f(\alpha_1, \ldots, \alpha_n))$. If the result is *nil*, *right-side* builds a new heap node β containing signature $f(\alpha_1, \ldots, \alpha_n)$, and updates **ind** by $\mathbf{ind}(f(\alpha_1, \ldots, \alpha_n) := \beta$. If the result is $\gamma \neq nil$ it shares the pre-existing representation rooted at γ. This technique is essentially the same as the *hashed cons* optimization in Lisp implementations [SLR78]. There are more places where it may be valuable to check the signature index for opportunistic sharing. Immediately after line (6) of *head-eval*, the signature at the root α of the heap representation of t may have changed, due to rewriting of t_i and path-compression. A sophisticated implementation may check $\mathbf{ind}(\mathbf{val}(\alpha))$, and if the result is $\beta \neq nil, \alpha$, reassign $\mathbf{repl}(\alpha) := \beta$, so that further path compression will replace links pointing to α by links to β, thus increasing sharing. If $\mathbf{ind}(\mathbf{val}(\alpha)) = nil$, then reassign $\mathbf{ind}(\mathbf{val}(\alpha)) := \alpha$. Finally, when *head-eval* is called recursively on a subterm rooted at the heap address α, the signature at α may be modified as the result of rewriting of a shared descendant of α that was visited previously by a path not passing through α. So, the same checking of **ind** and updating of **repl** or **ind** may be performed in step (1) of *head-eval*, after any implicit path compression, but before actually testing for strong head-normal form.

The opportunistic sharing techniques described above have a nontrivial cost in overhead added to each computation step, but in some cases they reduce the number of computation steps by an exponentially growing amount. See [She90] for more details on opportunistic sharing, and discussion of the tradeoff between ovehead on each step and number of steps. The method that checks new opportunities for sharing at every node visit as described above is called *lazy directed congruence closure*. An even more aggressive strategy, called *directed congruence closure* [Che80], makes extra node visits wherever there is a chance that a change in signature has created new sharing opportunities, even though the pattern-matcher/sequencer has not generated a need to visit all such nodes (i.e., when the signature at a shared node changes, directed congruence closure visits all parents of the node, many of which might never be visited again by *head-eval*). Directed congruence closure may reduce the number of steps in lazy directed congruence closure by an exponential amount, but the added overhead appears heuristically large, and changes the structure of the implementation significantly from the recursive schema of *head-eval*.

Opportunistic sharing strategies are generalizations of the idea of *memo functions* [KS86, Mit68, FH88]. Most memoing techniques limit themselves to remembering equalities of the form $f(t_1, \ldots, t_n) \doteq u$, where the t_is and u are all in normal form. Lazy directed congruence closure and its variants can remember partial evaluations as well, as can the *lazy memo functions* of Hughes [Hug85a]. There is a nice theoretical characterization of the memoing power of directed congruence closure.

Proposition 4.3.3 ([Che80]). *At any step in the process of rewriting a term, there is a set* $\mathbf{G}$ *of ground instances of equations that have been applied in rewriting. Using directed congruence closure, we never apply a ground instance that is a semantic consequence of* $\mathbf{G}$.

Not only does directed congruence closure avoid ever doing the same rewriting step twice in different contexts, it only performs a new rewriting step when new substitution of terms for variables in the given equations is absolutely necessary to derive that step.

4.3.4 Sharing and Memoing Nonground Terms—Paramodulation

The next natural step beyond directed congruence closure is sharing/memoing nonground terms. Abstractly in theorem-proving language, this amounts to applying *paramodulation* [RW69, Lov78, Gal86] to derive new equations with variables, instead of using demoduation to merely rewrite a given term. Paramodulation takes two equations $p \doteq q[r/x]$ and $s \doteq t$ and substitutions σ_r, σ_s of terms for variables such that $r\sigma_r = s\sigma_s$, and derives the new equation $p\sigma_r \doteq q\sigma_r[t\sigma_s/x]$, which follows from the first two by instantiation, substitution, and transitivity. Paramodulation sometimes improves the lengths of proofs by exponentially growing amounts compared to pure term rewriting [Lov78].

Example 4.3.4. Consider the set $\mathbf{T}_{rev}$ of equations defining list reversal (rev) and appending an element to the end of a list (app).

$$\begin{array}{lcl} \mathbf{T}_{rev} & = & \{ \quad rev(nil) \doteq nil, \\ & & \quad\; rev(cons(x,y)) \doteq app(x, rev(y)), \\ & & \quad\; app(x, nil) \doteq x, \\ & & \quad\; app(x, cons(y,z)) \doteq cons(y, app(x,z)) \quad \} \end{array}$$

The number of rewriting steps required to append an element to a list of i elements is $i+1$. But, consider applying paramodulation instead. One step of paramodulating the last equation in $\mathbf{T}_{rev}$ with itself yields $app(w, cons(x, cons(y,z))) \doteq cons(x, cons(y, app(w,z)))$, which moves the appended element w past the first 2 elements in the list. The new equation paramodulates again with $app(x, cons(y,z)) \doteq cons(y, app(x,z))$ to move the appended element past 3 list elements, but paramodulating the new equation with itself does even better, moving the appended element past 4 list elements. It is straightforward to produce a sequence of equations, each by paramodulating the previous one with itself, so that the ith equation in the sequence moves an appended element past 2^i list elements. So, paramodulation reduces the number of steps involved in normalizing $app(\alpha, \beta)$, where β is a list of i elements, from $i+1$ to $\mathcal{O}(\log i)$.

The improvement in normalizing $rev(\alpha)$ is less dramatic, but perhaps more useful in practice. If α has i elements, then rewriting requires $\Omega(i^2)$ steps to normalize $rev(\alpha)$, because it involves appending to the end of a sequence of lists of lengths $0, 1, \ldots, i$. But, we may produce a sequence of equations, each by paramodulating the previous one with

$$app(x, cons(y,z)) \doteq cons(y, app(x,z))$$

so that the ith equation in the sequence appends an arbitrary element to a list of length i. The whole sequence requires only i steps of paramodulation, so a list of i elements may be reversed by $\mathcal{O}(i)$ paramodulation steps.

There are several problems in reducing the ideas on paramodulation suggested in Example 4.3.4 to useful practice. First, we need a way to control paramodulation—that is, to choose useful steps of paramodulation to perform (equivalently, useful derived nonground equations to save and use along with the originally given program equations for further evaluation steps). Sequential strategies for lazy evaluation appear to solve this problem in principle. Whenever a sequential rewriting process discovers that a ground term s rewrites in one step to a ground term t, and existing links in the **repl** table rewrite t further to u, ground sharing techniques such as lazy directed congruence closure in effect save the derived ground equation $s \doteq u$ for later use as a single step (this is accomplished by setting $\mathbf{repl}(\alpha) := \gamma$ where $t = \mathbf{val}^*_{\mathbf{repl}}(\alpha)$ and $u = \mathbf{val}^*_{\mathbf{repl}}(\gamma)$). A method for nonground sharing should save instead the nonground equation $s' \doteq u'$, where s' is the generalization of s containing all of the symbol instances in s that were scanned by the sequential rewriting process in order to rewrite s to u, with unique variables replacing the unscanned portions of s. The appropriate generalization u' of u is easy to derive from the same information that determines s'. The equation $s' \doteq u'$ may be derived by paramodulation, using the same equations that were used to rewrite s to u. This strategy for driving paramodulation by the same sequentializing process that drives rewriting accomplishes the improvement of list reversal from quadratic to a linear number of inference steps in Example 4.3.4 above.

A second problem is to deal efficiently with the more general sets of equations that we get by adding the results of paramodulation to the original program equations. Notice that, even when the original program is orthogonal, there are overlaps between the original equations and the results of paramodulation. When several different equations apply to the same subterm, we probably should choose the most specific one, but that may not always be uniquely determined (see Figure 4 in Section 3.2.4 for the different sorts of overlap). At least in Example 4.3.4 above, choosing the most specific applicable equation accomplishes the improvement of list appending from linear to logarithmic.

Finally, we need an efficient data structure for representing sets of nonground equations, with a good way to add the new equations that result from paramodulation. Suffix trees [AHU74] appear to provide a good basis for representing dynamically increasing sets of patterns and/or equations [Str84], but it is not at all clear how to generalize these ideas efficiently to terms instead of strings (a good dissertation project: implement string rewriting with as much sharing as possible at both ends of a substring—this is essentially nonground sharing with only unary functions). A more detailed examination of Example 4.3.4 above reveals that, although the number of abstract steps required to reverse a list of length i using paramodulation is only $\mathcal{O}(i)$, each abstract step involves an equation with a larger left-hand side than the one before. So, while the number of steps reduces, the cost of pattern-matching in each step increases, and the total cost for reversal remains $\Omega(i^2)$. The exponential improvement in appending to the end of a list still yields a net win, but it is not clear how often such exponential improvements will arise in practice. In order to decrease the total cost of reversal to $\mathcal{O}(i)$, we need a way to make the pattern matching process incremental, so that the work done to match a smaller pattern may be reused in matching a larger pattern derived from it. I cannot find a reason why such efficient incremental pattern matching is impossible, but no known technique accomplishes it today.

Another starting point for research on nonground sharing is the work on sharing in the lambda calculus. [Sta82, Lam90, ?, GAL92] Some surprisingly powerful data structures have been discovered already, but the overhead of using them is still not understood well enough to determine their practical impact.

4.3.5 Storage Allocation

Essentially all of the known ideas for automatic storage allocation and deallocation are applicable to the **val/repl/ind** heap [Coh81, App91]. Unfortunately, one of the most attractive methods in current practice—generational garbage collection [LH83]—has been analyzed on the assumption that it is rare for old nodes in the heap to point to much more recently allocated ones. The **repl** links in lazy implementations of equational logic programming appear to violate that assumption. I know of no experimental or theoretical study of the efficiency of generational garbage collection with lazy evaluation. The aggressive sharing techniques also call into question the normal definition of 'garbage' nodes as those that *cannot* be reached starting from some set of directly accessible root nodes. In the discussion above, the directly accessible nodes are α_0 and all of the nodes entered in the **ind** table. With the more aggressive approaches to sharing, every node in the heap is usually accessible from nodes in **ind**, so there is no garbage according to the strict definition. On the other hand, nodes that are inaccessible from α_0 (the root of the term being rewritten to normal form) contain information that may be recalculated from the α_0-accessible nodes and the equational program, so, while they are not useless garbage, they are not *essential* to the calculation of a final result. I am not aware of any published literature on deallocation of useful but inessential nodes in a heap, which might be called 'rummage sale' instead of 'garbage collection.' Phillip Wadler proposed that a garbage-collection/rummage-sale procedure might also perform rewriting steps that reduce the size of the given term. Such a technique appears to provide the space-saving benefits of strictness analysis [Myc80, Hug85b], avoiding both the compile-time cost and the nonuniformity in the run-time control. Research is needed on the practical impact of various heuristic strategies for looking ahead in a rewriting sequence in search of a smaller version of the current term.

5 Compiling Efficient Code from Equations

Conventional approaches to compiling functional programs, following early Lisp compilers, associate a block of machine code with each function symbol f, and use the recursive schema for *eval* in Figure 8, with some representation of *suspensions* to encode partially evaluated terms as a new sort of value [Pey87]. A natural and less explored alternative is to associate a block of machine code with each state in a finite automaton whose states are derived from an analysis of the program using Ω-terms or subpattern sets [Bon89, DSS91]. The state-based approach has no explicit representation of suspensions, although in effect terms that it must build in the heap are implicit suspensions. And, a naive implementation of the state-based approach leads to more and smaller recursive procedures than the symbol-based approach. Of course, sophisticated symbol-based compilers may optimize by compiling multiple specialized versions of a single function for different sorts of arguments, and sophisticated state-based compilers may optimize by inlining some of the recursive calls. The *supercombinator* method [Hug82] introduces new symbols internal to the implementation, reducing the dependence of symbol-based compiling on the actual symbols in the source program. After such optimizations, it is not clear whether symbol-based compiling and state-based compiling will actually generate different code. The function symbols appear to be conceptually tied to the programmer's view of her program, while the automaton states appear to be conceptually tied to the computational requirements of the program. I conjecture that in the long run the state-based approach will prove more convenient for achieving the best run-time performance, while the symbol-based approach will maintain a more intuitive connection between program structure and performance.

Both the function-based and state-based methods compile the information in the set of equations constituting a program, but they both treat the input term and all of the intermediate terms produced in rewriting to normal form as static data structures, which must be interpreted by the compiled equational code. The Tigre system [KL89, Koo90] by contrast compiles each function symbol *in the term being evaluated* into a block of code—the entire term becomes a self modifying program that reduces itself to normal form. Tigre compiles only a particular extension of the combinator calculus; an interesting generalization would be a Tigre-like compiler for an arbitrary orthogonal system.

Several abstract machine languages have been proposed as intermediate languages for equational compilers, or as real machine languages for specialized term-rewriting machines. Landin's SECD machine [Lan65] is designed to support evaluation of lambda terms. The G Machine [Joh84, Aug84, BPR88] is intended as a specialized machine language or intermediate language for term rewriting in general. Equational Machine (EM) code [DSS91, Str88] is designed to support state-based compiling of equational programs, and the optimizations that seem most important in that approach. Several other abstract machine proposals are in [Car83, Car84, CCM85] . Turner proposes to compile equational systems into the combinator calculus, using the well-known capability of the combinator calculus to encode substitution in the lambda calculus [Tur79]. Then, an implementation of the combinator calculus, fine tuned for performance, can support essentially all equational logic programming languages. The impact of such translations of one rewriting system to another is ill understood—in particular the effects on sharing and parallelism (see Section 6) are quite subtle. The Warren Abstract Machine (WAM) [War83], intended as an intermediate language for Prolog compilers, has also been used in theorem provers as a representation for terms—t is represented by WAM code to unify an arbitrary input with t. I am not aware of any published study of the applicability of such a WAM encoding of terms to functional or equational logic programming.

6 Parallel Implementation

One of the primary early motivations for studying functional programming languages was the apparent opportunities for parallel evaluation [Bac78]. So, it is ironic that most of the effort in functional and equational logic programming to date involves sequential implementation. A variety of proposals for parallel implementation may be found in [Pey87].

Both strictness analysis and sequentiality analysis are used primarily to choose a sequential order of computation that avoids wasted steps. An important open topic for research is the extension of sequentiality analysis to support parallel computation. Huet and Lévy's sequentiality analysis is already capable of identifying more than one strongly needed redex in a term. A parallel implementation might allocate processors first to the strongly needed redexes, and then to other more speculative efforts. It appears that sequentiality analysis can be generalized rather easily (although this has not been done in print) to identify strongly needed *sets* of redexes, where no individual redex is certain to be needed, but at least one member of the set is needed. For example, with the positive parallel-or equations in $\mathbf{T}_{or+}$ of Examples 2.3.16 and 3.2.1, it is intuitively clear that if α is needed in t, and β is needed in u, then at least one of α and β must be rewritten in $or(t, u)$ (although neither is *needed* according to the formal Definition 3.2.3). Further research is required on the practical impact of heuristic strategies for allocating parallel processors to the members of strongly needed sets. It is natural to give priority to the singleton sets (that is, to the strongly needed redexes), but it is not clear whether a set of size 2 should be preferred to one of size 3—perhaps other factors than the size of the set should be considered. Strongly needed redex sets are essentially *disjunctive* assertions about the need for redexes—more general sorts of boolean relations may be useful (e.g., either *all* of $\alpha_1, \ldots, \alpha_m$ or *all* of $\beta_1, \ldots, \beta_n$ are needed).

Unfortunately, since strongly needed redexes are all outermost, sequentiality analysis as known today can only help with parallelism between different arguments to a function. But, one of the most useful qualities of lazy programming is that it simulates a parallel producer-consumer relationship between a function and its arguments. It seems likely that much of the useful parallelism to be exploited in equational logic programming involves parallel rewriting of nested redexes. An analysis of nonoutermost needed redexes appears to require the sort of abstract interpretation that is used in strictness analysis [Myc80, Hug85b]—it certainly will depend on right-hand sides as well as left-hand sides of equations. Unfortunately, most of the proposed intermediate languages for compiling equational programs are inherently sequential, and a lot of work is required to convert current sequential compiling ideas to a parallel environment. The idea of compiling everything into combinators may not be useful for parallel implementation. The known translations of lambda terms into combinators eliminate some apparent parallelism between application of a function and rewriting of the definition of the function (although no direct implementation is known to support all of this apparent parallelism either). Only very preliminary information is available on the inherent *parallel* power of rewriting systems: even the correct definition of such power is problematic [O'D85].

At a more concrete level, there are a lot of problems involved in parallelizing the heap-based execution of evaluation/rewriting sequences. A data structure, possibly distributed amongst several processors, is needed to keep track of the multiple locations at which work is proceeding. If any speculative work is done on redexes that are not known to be needed, there must be a way to kill off processes that are no longer useful, and reallocate their processing to to useful processes (although aggressive sharing through the signature index confuses the question of usefulness of processes in the same way that it confuses the usefulness of heap nodes).

Sharing presents another challenge. Several different processors may reach a shared node by different paths. It is important for data integrity that they do not make simultaneous incompatible updates, and important for efficiency that they do not repeat work. But, it is incorrect for the first process to lock all others completely out of its work area. Suppose we are applying a system of equations including $car(cons(x, y)) \doteq x$. There may be a node α in the heap containing the signature $cons(\beta_1, \beta_2)$ shared between two parents, one of them containing $cons(\alpha, \delta)$ and the other containing $car(\alpha)$. A process might enter first through the *cons* parent, and perform an infinite loop rewriting inside β_2. If a second process enters through the *car* node, it is crucial to allow it to see the *cons* at α, to link to β_1, and depending on the context above possibly to continue rewriting at β_1 and below.

Abstractly, we want to lock node/state pairs, and allow multiple processes to inspect the same node, as long as they do so in different states, but when a process reaches a node that is already being processed *in the same state* it should wait, and allow its processor to be reassigned, since any work that it tries to do will merely repeat that of its predecessor. It is not at all clear how to implement such a notion of locking with acceptable overhead. Perhaps a radically different approach for assigning work to processors is called for, that does not follow the structure of current sequential implementations so closely. For example, instead of the shared-memory approach to the heap in the preceeding discussion, perhaps different sections of the heap should be assigned to particular processors for a relatively long period, and demands for evaluation should be passed as messages between processors when they cross the boundaries of heap allocation.

7 Extensions to Equational Logic Programming

7.1 Incremental Infinite Input and Output

The logic programming approach to answering normal form queries leads naturally to the use of lazy evaluation to guarantee completeness. But, once we commit to lazy evaluation, the basic scheme of normal form queries seems too limited. Inside an equational computation, it is natural to conceive of procedures consuming and producing potentially infinite terms incrementally, in the manner of communicating coroutines. It is perfectly sensible to define objects that correspond intuitively to infinite terms, as long as only finite portions of these infinite terms are required to produce output. It is annoying that the full flexibility of incremental demand-driven computation in the internal computation is not available at the input/output interface. Aside from the loss of input/output programming power per se, such a discrepancy between internal and external communication tends to encourage large, unwieldy, monolithic programs, and discourage the collections of small, separate, modular programs that are more desirable in many ways.

The desired abstract behavior of a demand-driven incremental I/O interface is essentially clear. The consumer of output demands the symbol at the root of the output. The equational program computes until it has produced a head-normal form equivalent to the input, then it outputs the root symbol of that head-normal form, and recursively makes the principal subterms available to the consumer of output, who can demand symbols down different paths of the output tree in any order. During the computation of a head-normal form, the equational program generates demands for symbols in the input. The producer of input is responsible only for providing those input symbols that are demanded. In this way, both input and output terms are treated incrementally as needed, and each may be infinite.

A direct generalization of equational logic programming by extending the set of terms to include infinite ones is semantically problematic.

Example 7.1.1. Consider the system of equations

$$\mathbf{T}_{27} = \{a \doteq f(c,a),\ b \doteq f(c,f(c,b))\}$$

The natural infinite output to expect from input a is $f(c,f(c,\ldots))$. b produces the same infinite output. So, if in our semantic system $f(c,f(c,\ldots))$ is a generalized term, with a definite value (no matter what that value is), then $\mathbf{T}_{27} \models (a \doteq b)$. But, according to the semantics of Definition 2.3.14 in Section 2.3.4 of the chapter 'Introduction: Logic and Logic Programming Languages,' $\mathbf{T}_{27} \not\models_{\doteq} (a \doteq b)$, because there are certainly models in which the equation $x \doteq f(c,x)$ has more than one solution. For example, interpret f as integer addition, and c as the identity element zero. Every integer n satisfies $n = 0 + n$.

Even if we replace the second equation with the exact substitution of b for a in the first, that is $b \doteq f(c,b)$, it does not follow by standard equational semantics that $a \doteq b$. With some difficulty we may define a new semantic system, with a restricted set of models in which all functions have sufficient continuity properties to guarantee unique values for infinite terms. But it is easy to see that the logical consequence relation for such a semantic system is not semicomputable (recursively enumerable). It is always suspect to say that we understand a system of logic according to a definition of meaning that we cannot apply effectively.

Instead, I propose to interpret outputs involving infinite terms as abbreviations for infinite conjunctions of formulae in the First-Order Predicate Calculus (FOPC) (such infinite conjunctions, and infinite disjunctions as well, are studied in the formal system called $L_{\omega_1,\omega}$ [Kar64]).

Definition 7.1.2. The *First-Order Predicate Calculus with Equality* ($\text{FOPC}_{\doteq}$) is the result of including the equality symbol $\doteq$ in the set $\mathbf{Pred}_2$

of binary predicate symbols, and combining the semantic rules for FOPC (Definitions 2.3.4–2.3.1, Section 2.3.1 of the chapter 'Introduction: Logic and Logic Programming Languages') and Equational Logic (Definitions 2.3.13–2.3.14, Section 2.3.4 of the 'Introduction ...' chapter) in the natural way (a model for FOPC$_{\doteq}$ satisfies the restrictions of a model for FOPC and the restrictions of a model for Equational Logic).

$L_{\omega_1,\omega,\doteq}$ is the extension of FOPC$_{\doteq}$ to allow countably infinite conjunctions and disjunctions in formulae. Let $\mathbf{F}_{P,\omega_1,\omega,\doteq}$ be the set of formulae in $L_{\omega_1,\omega,\doteq}$. Extend the semantic system for FOPC$_{\doteq}$ to a semantic system for $L_{\omega_1,\omega,\doteq}$ by the following additional rules for infinite conjunctions and disjunctions:

1. $\rho_{\tau,\nu}(\bigwedge\{A_1, A_2, \ldots\}) = 1$ if and only if $\rho_{\tau,\nu}(A_i) = 1$ for all $i \geq 1$
2. $\rho_{\tau,\nu}(\bigvee\{A_1, A_2, \ldots\}) = 1$ if and only if $\rho_{\tau,\nu}(A_i) = 1$ for some $i \geq 1$

That is, an infinite conjunction is true precisely when all of its conjuncts are true; and an infinite disjunction is true precisely when at least one of its disjuncts is true.

A set T of terms is a *directed set* if and only if, for every two terms $t_1, t_2 \in T$, there is a term $t_3 \in T$ such that t_3 is an instance of t_1 and also an instance of t_2.

Let U be a countable directed set of linear terms. For each $u \in U$, let $\vec{y}_u$ be a list of the variables occurring in u. Let t be a finite ground term. The *conjunctive equation* of t and U is the infinite conjunction

$$\bigwedge\{(\exists \vec{y}_u : (t \doteq u)) : u \in U\}$$

$\mathbf{T}_P^\omega$ is the set of finite and infinite terms.

The *term limit* of a directed set U (written $\lim(U)$) is the possibly infinite term resulting from overlaying all of the terms $u \in U$, and substituting the new symbol $\perp$ for any variables that are not overlaid by nonvariable symbols. Since members of U are pairwise consistent, every location in the limit gets a unique symbol, or by default comes out as $\perp$. To see more rigorously that the term limit of U is well defined, construct a (possibly transfinite) *chain*, $\langle t_0 \sqsubseteq t_1 \sqsubseteq \cdots \rangle$. First, $t_0 = \perp$. Given t_α, choose (axiom of choice) an $s \in U$ such that $s \not\sqsubseteq t_\alpha$, and let $t_{\alpha+1}$ be the overlay of s with t_α. At limit ordinals λ, let t_λ be the limit of the chain of terms preceding λ. With a lot of transfinite induction, we get to a t_β such that, for all $s \in U$, $s \sqsubseteq t_\beta$, at which point the chain is finished. $\lim(U)$ is the limit of that chain.

The *canonical set* of an infinite term t (written $\mathbf{approx}(t)$) is the set of all finite linear terms t' (using some arbitrary canonical scheme for naming the variables) such that t is an instance of t'. $\mathbf{approx}(t)$ is a directed set, and $\lim(\mathbf{approx}(t)) = t$.

A careful formal definition of infinite terms and limits is in [KKSd91, DKP91], but an intuitive appreciation suffices for this section.

When the finite input t produces the infinite output $\lim(U)$, instead of interpreting the output as the equation $t \doteq \lim(U)$, interpret it as the conjunctive equation of t and U. Notice that a *chain* (a sequence $u_1, u_2, \ldots$ such that u_{i+1} is an instance of u_i) is a special case of a directed set. The infinite outputs for finite inputs may be expressed by chains, but the generality of directed sets is needed with infinite outputs for infinite inputs below.

Infinite inputs require a more complex construction, since they introduce universal quantification that must be nested appropriately with the existential quantification associated with infinite output. Also, different orders in which a consumer explores the output may yield different sequences of demands for input, and this flexibility needs to be supported by the semantics.

Definition 7.1.3. Let T and U be directed sets of linear terms such that no variable occurs in both $t \in T$ and $u \in U$. For each term $t \in T$ and $u \in U$, let $\vec{x}_t$ and $\vec{y}_u$ be lists of the variables occurring in t and u, respectively. Let $f : T \rightarrow \mathbf{2}^U$ be a function from terms in T to directed subsets of U, such that when t_2 is an instance of t_1, then every member of $f(t_2)$ is an instance of every member of $f(t_1)$. The *conjunctive equation* of T and U by f is the infinite conjunction

$$\bigwedge\{(\forall \vec{x}_t : \exists \vec{y}_u : (t \doteq u)) : t \in T \text{ and } u \in f(t)\}$$

Definition 7.1.4. Let $s_1, \ldots, s_i \in \mathbf{T}_P$ be terms. A term u is an *incremental normal form* for $\{s_1, \ldots, s_i\}$ if and only if no nonvariable subterm of u unifies with a term in $\{s_1, \ldots, s_i\}$. Equivalently, u is an incremental normal form if and only if u is a normal form and every substitution of normal forms for variables leaves u in normal form. Equivalently, u is an incremental normal form if and only if melting preserves u_Ω, where u_Ω results from substituting Ω for each variable in u.

Let $\mathbf{norm}^\omega$ be a new formal symbol. Let

$$\mathbf{Q}^\omega_{\doteq} = \{(\mathbf{norm}^\omega\ s_1, \ldots, s_i : t_\omega) : t_\omega \in \mathbf{T}^\omega_P, s_1, \ldots, s_i \in \mathbf{T}_P\}$$

Define the relation $?\!\!-^\omega_{\doteq} \subseteq \mathbf{Q}^\omega_{\doteq} \times \mathbf{F}_{P,\omega_1,\omega,\doteq}$ by

$$(\mathbf{norm}^\omega\ s_1, \ldots, s_i : t_\omega)\ ?\!\!-^\omega_{\doteq}\ A$$

if and only if there exists a directed set U and a monotonic (in the instance ordering $\sqsubseteq$) function $f : \mathbf{approx}(t_\omega) \rightarrow \mathbf{2}^U$ such that

1. A is the conjunctive equation of $\mathbf{approx}(t_\omega)$ and U by f

2. u is in incremental normal form, for all $u \in U$

Now $\langle \mathbf{F}_{P,\omega_1,\omega,\doteq}, \mathbf{Q}^\omega_{\doteq}, ?\!\!-^\omega_{\doteq} \rangle$ is a query system representing the answers to questions of the form 'what set of incremental normal forms for $s_1, \ldots, s_i$ is conjunctively equal to t?' In an implementation of equational logic programming based on $\mathbf{norm}^\omega$, the consumer of output generates demands causing the computation of some $u \in U$. Some of the symbols in u are demanded directly by the consumer, others may be demanded by the system itself in order to get an incremental normal form. The implementation demands enough input to construct $t \in \mathbf{approx}(t_\omega)$ such that $u \in f(t)$. By modelling the input and output as directed sets, rather than sequences, we allow enough flexibility to model all possible orders in which the consumer might demand output. The function f is used to model the partial synchronization of input with output required by the semantics of equational logic. Unfortunately, the trivial output $y, y, \ldots$, with term limit $\perp$, always satisfies 1–2 above. The most useful implementation would provide *consequentially strongest* answers (Definition 2.2.5, Section 2.2 of the chapter 'Introduction: Logic and Logic Programming Languages')—that is, they would demand the minimum amount of input semantically required to produce the demanded output. If consequentially strongest answers appear too difficult to implement, a more modest requirement would be that $\lim(U)$ is maximal among all correct answers—that is, all semantically correct output is produced, but not necessarily from the minimal input. The techniques outlined above do not represent sharing information, neither within the input, within the output, nor between input and output. Further research is needed into semantic interpretations of sharing information.

Example 7.1.5. Consider the system

$$\mathbf{T}_{28} = \{f(g(x,y)) \doteq h(h(g(x,y)))\}$$

and the infinite input term

$$t_\omega = f(g(t_\omega, t_\omega))$$

and the query

$$\mathbf{norm}^\omega\ f(g(x,y)) : t_\omega$$

The natural desired answer A to this query is the conjunction of

$$
\begin{array}{llrcl}
\forall x_1 : & \exists y_1 : & x_1 & \doteq & y_1 \\
\forall x_1 : & \exists y_1 : & f(x_1) & \doteq & y_1 \\
\forall x_1, x_2 : & \exists y_1, y_2 : & f(g(x_1, x_2)) & \doteq & h(h(g(y_1, y_2))) \\
\forall x_1, x_2 : & \exists y_1, y_2 : & f(g(f(x_1), x_2)) & \doteq & h(h(g(y_1, y_2))) \\
\forall x_1, x_2, x_3 : & \exists y_1, y_2, y_3 : & f(g(f(g(x_1, x_2), x_3))) & \doteq & h(h(g(h(h(g(y_1, y_2))), y_3))) \\
\forall x_1, x_2 : & \exists y_1, y_2 : & f(g(x_1, f(x_2))) & \doteq & h(h(g(y_1, y_2))) \\
\forall x_1, x_2, x_3 : & \exists y_1, y_2, y_3 : & f(g(x_1, f(g(x_2, x_3)))) & \doteq & h(h(g(y_1, h(h(g(y_2, y_3)))))) \\
 & & & \vdots &
\end{array}
$$

A represents the infinite output $t'_\omega = h(h(g(t'_\omega, t'_\omega)))$. U does not contain all approximations to t'_ω, but only the ones that have gs as the rootmost non-$\perp$ symbols. The montonte function mapping partial inputs to the portions of output that they determine is:

$$
\begin{array}{rcl}
\perp & \mapsto & \perp \\
f(\perp) & \mapsto & \perp \\
f(g(\perp, \perp)) & \mapsto & h(h(g(\perp, \perp))) \\
f(g(f(\perp), \perp)) & \mapsto & h(h(g(\perp, \perp))) \\
f(g(f(g(\perp, \perp)), \perp)) & \mapsto & h(h(g(h(h(g(\perp, \perp))), \perp))) \\
f(g(\perp, f(\perp))) & \mapsto & h(h(g(\perp, \perp))) \\
f(g(\perp, f(g(\perp, \perp)))) & \mapsto & h(h(\perp, g(h(h(g(\perp, \perp)))))) \\
 & \vdots &
\end{array}
$$

A natural approach to representing sharing information would be to use $\forall x_1, x_2 : f(g(x_1, x_2)) \doteq h(h(g(x_1, x_2)))$ instead of $\forall x_1, x_2 : \exists y_1, y_2 : f(g(x_1, x_2)) \doteq h(h(g(y_1, y_2)))$, etc., but this idea has yet to be explored.

It is simple in principle to modify an implementation of equational logic programming to deal incrementally with infinite input queries and output answers in the query system $\mathbf{Q}^\omega_{\doteq}$. The practical details of such an implementation are interesting and challenging. The *tours* protocol [Reb92, Reb93] provides one design for incremental input and output of infinite terms. Implementations based on the recursive scheme for *head-eval* in Figure 9 may be adapted relatively simply to incremental input and output, simply by replacing the scheme for *norm* in Figure 10 by a scheme that issues calls to *head-eval* only as symbols are demanded by the consumer of output.

Unfortunately, such adaptations are not normally complete, because of sequentializing problems. The trouble is that conjunctive behavior (which is easy to sequentialize) in the application of a rule corresponds to disjunctive behavior (which is impossible to sequentialize) in the production of strong head-normal forms, and vice versa.

Example 7.1.6. Consider the following equation defining the *negative sequential-or:*

$$\mathbf{T}_{or-s} = \{or(\mathit{false}, \mathit{false}) \doteq \mathit{false}\}$$

This equation causes no problem in sequentializing the rewriting of a term to a finite normal form, since it requires *both* arguments to or. But, $or(s, t)$ is a strong head-normal form if and only if *either* of s or t is a strong head-normal form $\neq \mathit{false}$. So, it is not safe to rewrite s, because of the possibility that s has no strong head-normal form, while t rewrites to strong head-normal form $u \neq \mathit{false}$. For symmetric reasons, it is not safe to rewrite t. Only a parallel rewriting of both s and t is complete, and such parallel rewriting is very likely to waste steps.

By contrast, the positive parallel-or equations $\mathbf{T}_{or+}$ of Examples 2.3.16 and 3.2.1 prevent sequential computation of a finite normal form, but pose no problem to sequential head-normalization of $or(s, t)$ by rewriting s and t—*both* arguments must rewrite to strong head-normal forms $\neq \mathit{true}$ in order to head-normalize $or(s, t)$.

So, to achieve complete implementations of equational logic programming with incremental input and output, we must solve the sticky problems of efficient implementation of parallel rewriting, even if the parallelism is simulated by multitasking on a sequential processor. In the meantime, sequential and incomplete implementations are likely to be practically useful for many problems—the incompleteness appears to be no more serious than the incompleteness of Prolog, which is widely accepted.

Beyond the incremental evaluation of an input that increases over time, there is an even more challenging problem to evaluate a *changing* input. In principle, changing input may be reduced to increasing input by representing the entire edit history of a term as another term [Reb93]. In practice, there is a lot of research required to achieve practical efficiency. A highly efficient general scheme for re-evaluation after a change in input will be extremely valuable. There are several techniques known for re-evaluation in special structures, such as attributed parse trees [DRT81, PT89], but none approaches the generality or fine-grainedness of equational re-evaluation. A late result for the λ-calculus that I have not had a chance to review is in [FT90]. Powerful memo techniques seem to be crucial to efficient re-evaluation of a changed input term. In some sense efficient re-evaluation is precisely the intelligent reuse of partial results from the previous evaluation. I conjecture that nonground memoing will be especially valuable for this problem.

Finally, we should study implementations of equational logic programming in which the set of equations is changing. Changing sets of equations arise in rapid recompiling during program development, in applications that use equations as inputs rather than as programs, and in methods based on nonground memoing or paramodulation (Section 4.3.4). Very little is known about this problem—all that I am aware of is a technique for incremental maintenance of a finite automaton for pattern matching [Str84].

7.2 Solving Equations

Queries of the form (**solve** $x_1, \ldots, x_n : s \doteq t$) (Definition 2.3.18, Section 2.3.4 of the chapter 'Introduction: Logic and Logic Programming Languages') require the solution of the equation $s \doteq t$ for the values of $x_1, \ldots, x_n$. This appears to be much more difficult than finding normal forms, and to offer more programming power. In particular, it appears that solving equations is the essential problem in the most natural approach to combining the useful qualities of Prolog and lazy functional programming. Notice that **solve** queries are quite similar to the **what** queries processed by Prolog (Section 2.3.1 of the 'Introduction ...' chapter). A combination of the methods for solving equations with the techniques of Prolog should provide a way of answering **what** queries in the First-Order Predicate Calculus with Equality.

Much of the theory of term rewriting falls short of dealing with the problems posed by equation solving. Confluence appears to help—if the given set $\mathbf{T}$ of equations is confluent, then we may seek an instance s' of s and t' of t, and a term u, such that $s' \xrightarrow{\mathbf{T}}{}^{*} u$ and $t' \xrightarrow{\mathbf{T}}{}^{*} u$, ignoring the symmetric rule and applying equations only from left to right. But, for a complete implementation, it is not sufficient to rewrite needed redexes. That is, it may be that there is a u such that $s' \xrightarrow{\mathbf{T}}{}^{*} u$ and $t' \xrightarrow{\mathbf{T}}{}^{*} u$, but no such u may be reached by rewriting only needed redexes (Definition 3.2.3, Section 3.2.3). Outermost complete rewriting (Definition 3.1.5, Section 3.1) is similarly insufficient. Complete rewriting sequences (Definition 3.1.3) come a bit closer. If s' and t' have a common form, then a complete rewriting sequence starting with s' is guaranteed eventually to produce a term u such that $t' \xrightarrow{\mathbf{T}}{}^{*} u$. But, another complete rewriting sequence starting with t' is not guaranteed to produce the same u. Finally, even if we generate rewriting sequences from s' and t' that both contain u, it is not clear how to synchronize the sequences so that u appears in both of them at the same time.

No thoroughly satisfactory method is known for a complete implementation of equation solving in an arbitrary strongly sequential system. But, enough is known to make two observations that are likely to be useful in such an implementation.

1. Suppose we are solving $f(s_1, \ldots, s_m) \doteq g(t_1, \ldots, t_n)$, where $f \neq g$. It suffices to rewrite only needed redexes until the head symbols agree. Unfortunately, it is hard to know whether to rewrite $f(s_1, \ldots, s_m)$ or $g(t_1, \ldots, t_n)$ or both, but within each we can use the same sort of sequentializer that rewrites a single term to normal form.
2. There are only two cases in which it is helpful to instantiate a variable:
   (a) when the instantiation creates a redex at a proper ancestor of the variable, which we immediately rewrite (so, in the situation of (1) above, the redex must be needed);
   (b) when the instantiation unifies corresponding subterms of s and t.

   In each case, we should substitute the most general (smallest) term that achieves the redex or the unification.

Jayaraman noticed (1) [Jay85], and used it to implement a system called EqL for solving equations $s \doteq t$ only in the case where instances s' of s and t' of t reduce to a common form consisting entirely of constructor symbols in a constructor-orthogonal system (Definition 2.3.13, Section 2.3.2). For this purpose, it suffices to rewrite s' and t' to head-normal forms. If the head symbols agree, recursively solve equations between the corresponding arguments, otherwise there is no solution. It should be straightforward to generalize EqL to find solutions in normal form in strongly sequential rewrite- and rule-orthogonal systems. If we want all solutions, even those that are not in normal form, we must somehow explore in parallel the attempt to rewrite the head symbol of s' and the attempt to solve the equation without head rewriting one or both of them (and a similar parallelism for t'). This introduces all of the problems of parallelism discussed in Section 6, and it probably limits or eliminates the ability to do path compression on **repl** links (Section 4.3), since it is not sufficient to work only on the most rewritten form of a term ($\mathbf{val}^{max}_{\mathbf{repl}}(\alpha_0)$, where α_0 is the root of the heap representation of the term being rewritten).

Heuristically, it appears that the majority of the individual steps in solving an equation are unlikely to create such parallelism. But, EqL's uniform requirement of finding a solution in normal form is likely to introduce the same huge inefficiencies in some cases as strict evaluation (notice that the individual evaluations of s', t' to normal form are lazy, but the equation solution is not lazy, as it may take steps that are required to reach normal form, but not to solve the equation). If these heuristic observations are accurate, even an implementation with rather high overhead for parallelism may be very valuable, as long as the cost of parallelism is introduced only in proportion to the actual degree of parallelism, rather than as a uniform overhead on sequential evaluation as well.

Among the theoretical literature on term rewriting, the most useful material for equation solving will probably be that on *narrowing* [Fay79] (see the chapter 'Equational Reasoning and Term Rewriting Systems in Volume 1), which is a careful formalization of observation (2) above. Information on sequencing narrowing steps seems crucial to a truly efficient implementation: some new results are in [AEH94].

7.3 Indeterminate Evaluation in Subset Logic

Until now, I have considered only quantified conjunctions of equations as formulae, else I cannot claim to be using equational logic. The discipline of equational logic constrains the use of term rewriting systems, and leads us to insist on confluent systems with lazy evaluation. Lazy evaluation appears to be mostly a benefit, but the restriction to confluent systems is annoying in many ways, and it particularly weakens the modularity of equational logic programming languages, since orthogonality depends on the full textual structure of left-hand sides of equations, and not just on an abstract notion of their meanings. As well as improving the techniques for guaranteeing confluence, we should investigate nonconfluent term rewriting. Since, without confluence, term rewriting cannot be semantically complete for equational logic, we need to consider other logical interpretations of term rewriting rules.

A natural alternative to equational logic is *subset logic* [O'D87]. Subset logic has the same terms as equational logic, the formulae are the same except that they use the backward subset relation symbol $\supseteq$ instead of $\doteq$ (Definition 2.3.13, Section 2.3.4 of the chapter 'Introduction: Logic and Logic Programming Languages'). Subset logic semantics are the same as equational (Definition 2.3.14, Section 2.3.4 of the 'Introduction ...' chapter), except that every term represents a *subset* of the universe of values instead of a single value, function symbols represent functions from subsets to subsets that are extended pointwise from functions of individual values (i.e., $f(S) = \bigcup\{f(\{x\}) : x \in S\}$). Subset logic is complete with the *reflexive*, *transitive*, and *substitution* rules of equality, omitting the *symmetric* rule (Definition 2.1.2, Section 2.1). So, term rewriting from left to right is complete for subset logic, with no restrictions on the rules. Technically, it doesn't matter whether the left side of a rule is a subset of the right, or *vice versa*, as long as the direction is always the same. Intuitively, it seems more natural to think of rewriting as producing a subset of the input term, since then a term may be thought of as denoting a set of possible answers. Thus, a single line of a subset logic program looks like $l \supseteq r$. Note that normal forms do not necessarily denote singleton sets, although it is always possible to construct models in which they do.

Subset logic programming naturally supports programs with indeterminate answers. When a given input s rewrites to two different normal forms t and u, subset logic merely requires that $s \supseteq t$ and $s \supseteq u$ are true, from which it does not follow that $t \doteq u$ is true, nor $t \supseteq u$, nor $u \supseteq t$. While equational logic programming extends naturally to infinite inputs and outputs, without changing its application to finite terms, such extension of subset logic programming is more subtle. If only finite terms are allowed, then infinite computations may be regarded as a sort of failure, and a finite normal form must be found whenever one exists. If incremental output of possibly infinite normal forms is desired, then there is no effective way to give precedence to the finite forms when they exist. The most natural idea seems to be to follow all possible rewriting paths, until one of them produces a stable head symbol (that is, the head symbol of a strong head-normal form) for output. Whenever such a symbol is output, all rewriting paths producing different symbols at the same location are dropped. Only a reduction path that has already agreed with all of the symbols that have been output is allowed to generate further output. The details work out essentially the same as with infinite outputs for equational programs, merely substituting $\supseteq$ for $\doteq$. It is not at all clear whether this logically natural notion of commitment to a symbol, rather than a computation path, is useful. I cannot find a natural semantic scheme to support the more conventional sort of commitment to a computation path instead of an intermediate term, although a user may program in such a way that multiple consistent computation paths never occur and the two sorts of commitment are equivalent.

Efficient implementation of nonconfluent rewriting presents a very interesting challenge. It is obviously unacceptable to explore naively the exponentially growing set of rewriting sequences. A satisfying implementation should take advantage of partial confluent behavior to prune the search space down to a much smaller set of rewriting sequences that is still capable of producing all of the possible outputs. The correct definition of the right sort of partial confluence property is not even known. It is *not* merely confluence for a subset of rules, since two rules that do not interfere with one another may interfere differently with a third, so that the order in which the two noninterfering rules are applied may still make a difference to the outcome. Pattern-matching and sequencing techniques must be generalized as well, and a good data structure designed to represent simultaneously the many different rewritings under consideration as compactly as possible. A good complete implementation of nonconfluent rewriting will probably allow the prohibited terms in normal forms (Definition 2.3.16, Section 2.3.4 of the chapter 'Introduction: Logic and Logic Programming Languages') to be different from the left-hand sides of rules. Sharing of equivalent subterms becomes problematic since it may be necessary to reduce two different occurrences of the same subterm in two different ways. The difficulties in discovering a complete implementation of subset logic programming would be well rewarded, since this style of programming can capture the useful indeterminate behavior of Prolog, while avoiding the repeated generation of the same solution by slightly different paths that slows some Prolog programs down substantially.

Even an incomplete implementation of nonconfluent rewriting might be very valuable. In the spirit of logic programming, such an implementation should depend on inherent structural properties of the rules in a program to determine which of a set of possible rewriting steps is chosen when the redex patterns overlap. From the point of view of implementation, the most natural approach appears to be to select an outermost connected component of overlapping redex patterns, but to rewrite within the component from innermost out. It is not clear how to resolve the ambiguous case where two redex patterns have the same root, nor has anyone proposed a sensible semantic explanation based on logic programming for any principle for choosing between conflicting rules.

Jayaraman has a different concept of subset logic programming [Jay92]. Meseguer's *rewriting logic* has the same rules as my subset logic, with a different semantic justification. [Mes92]

7.4 Relational Rewriting

Equation solving can lead to a combination of Prolog and functional programming through logic programming in the First-Order Predicate Calculus with Equality. Subset logic programming, implemented with nonconfluent term rewriting, can lead to an implementation of Predicate Calculus programming in a sort of indeterminate functional programming. Another intriguing way to reconcile some of the good qualities of Prolog and functional programming is relational logic programming. Relational logic programming can be approached as a generalization of terms to systems of relational constraints, or as a purification of Prolog by removing the function symbols (called 'functors' in Prolog).

A relational formula is a FOPC formula with no function symbols (this prohibition includes zeroary functions, or constants). The natural analogue to a term in relational logic is a conjunction of atomic formulae, which may be represented as a hypergraph (like a graph, but edges may touch more than two nodes) with nodes standing for variables and hyperedges (edges touching any number of nodes, not necessarily two) representing predicate symbols. Two fundamental problems stand in the way of a useful implementation of relational logic programming through hypergraph rewriting.

1. We need an efficient implementation of hypergraph rewriting. The

pattern-matching problem alone is highly challenging, and no practical solution is yet known.

2. We need a semantic system that makes intuitive sense, and also supports computationally desirable methods of hypergraph rewriting.

A first cut at (2) might use the semantic system for FOPC, and express rewrite rules as formulae of the form

$$\forall \vec{x} : \exists \vec{y} : (A_1 \wedge \cdots \wedge A_m) \Leftarrow (B_1 \wedge \cdots \wedge B_n)$$

These clauses differ from the Horn clauses of Prolog in two ways. The quantification above is $\forall \vec{x} : \exists \vec{y} :$, where clauses are universally quantified. And, the consequence of the implication above is a conjunction, where Horn clauses allow at most one atomic formula in the consequence, and even general clauses allow a disjunction, rather than a conjunction. Notice that, with Prolog-style all-universal quantification, the implication distributes over conjunctions in the consequence ($(A_1 \wedge A_2) \Leftarrow B$ is equivalent to $(A_1 \Leftarrow B) \wedge (A_2 \Leftarrow B)$). But, if existentially quantified variables from $\vec{y}$ appear in $A_1, \ldots, A_m$, the distribution property does not hold.

Since the consequence and hypothesis of our formula are both conjunctions of atomic formulae, they may be represented by hypergraphs, and it is natural to read the formula as a hypergraph rewrite rule. A natural sort of hypergraph rewriting will be sound for deriving universally quantified implications from $\forall\exists$ implications, as long as the existentially quantified variables $\vec{y}$ correspond to nodes in the graph that participate *only* in the hyperedges representing $A_1, \ldots, A_m$. The universally quantified variables in $\vec{x}$ correspond to nodes that may connect in other ways as well as by hyperedges representing $A_1, \ldots, A_m$, so they are the interface nodes to the unrewritten part of the hypergraph. Interestingly, the final result of a computation on input I is an implication $\forall \vec{w} : I \Leftarrow O$. This implication does not give a solution to the goal I, as in Prolog. Rather, it rewrites the goal I into a (presumably simpler or more transparent) goal O, such that every solution to O is also a solution to I. This is formally similar to functional and equational logic programming, in that the class of legitimate outputs is a subclass of the legitimate inputs—in Prolog inputs are formulae and outputs are substitutions. Constraint logic programming [JL87, Las91] is perhaps heading in this direction.

The semantic treatment suggested above is not very satisfactory, as it rules out lazy evaluation, and even storage reclamation (an intermediate part of a hypergraph that becomes disconnected from the input and output cannot be thrown away, even though it cannot affect the nature of a solution, since if any part of the hypergraph is unsolvable, the whole conjunction is unsolvable). Also, the logically sound notion of unification of the left-hand side of a rule with a portion of a hypergraph in order to rewrite it seems far too liberal with this semantic system—without the single-valued quality of functions, additional relations can be postulated to hold anywhere. I conjecture that the first problem can be solved by a semantic system in which every unquantified formula is solvable, so that a disconnected portion of the hypergraph may be discarded as semantically irrelevant. One way to achieve this would be to use some sort of measured universe [Coh80], and require every atomic predicate to hold for all but a subset of measure 0—then every finite intersection of atomic predicates (equivalently, the value of every finite conjunction of atomic formulae) has a solution. Such a semantic system is not as weird as it first seems, if we understand the universe as a space of *concepts*, rather than real objects, and think of each predicate symbol as giving a tiny bit of information about a concept. We can conceive of every combination of information (such as a green unicorn with a prime number of feet that is a large power of 2), even though most such combinations never enter our notion of reality. Although the question of solvability becomes trivial in this semantic system, implications between solutions still have interesting content. The problem of limiting unifications in a useful way appears more difficult. I suggest that nonclassical 'relevant' interpretations of implication [AB75] are likely to be helpful.

References

[AB75] Alan Ross Anderson and Nuel D. Belnap Jr. *Entailment—the Logic of Relevance and Necessity*, volume 1. Princeton University Press, Princeton, NJ, 1975.

[AEH94] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 268–279. ACM, January 1994.

[AH87] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, UK, 1987.

[AHU74] A. V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[App91] Andrew Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation Techniques*. MIT Press, 1991.

[Aug84] Lennart Augustsson. A compiler for lazy ML. In *ACM Symposium on Lisp and Functional Programming*, August 1984.

[AW85] E. A. Ashcroft and W. W. Wadge. *Lucid, the Dataflow Programming Language*. Academic Press, London, UK, 1985.

[Bac74] John Backus. Programming language semantics and closed applicative languages. In *Proceedings of the 1st ACM Symposium on Principles of Programming Languages*, pages 71–86. ACM, 1974.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[Bar84] Hendrik Peter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[BC79] D. Bobrow and D. Clark. Compact encodings of list structure. *ACM Transactions on Programming Languages and Systems*, 1(2):266–286, 1979.

[BHY64] A. Bloss, Paul Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation: an International Journal*, 1, 147–164.

[BL79] Gérard Berry and Jean-Jacques Lévy. Letter to the editor, SIGACT News, v. 11, no. 1, 1979. pp. 3–4.

[Bon89] Anders Bondorf. A self-applicable partial evaluator for term-rewriting systems. In *International Joint Conference on the Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[BPR88] G. L. Burn, Simon L. Peyton Jones, and J. D. Robson. The spineless g-machine. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 244–258, 1988.

[BPW89] T. Baird, G. Peterson, and R. Wilkerson. Complete sets of reductions modulo associativity, commutativity, and identity. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 29–44, 1989.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, New York, NY, 1988.

[Car83] Luca Cardelli. The functional abstract machine. *Polymorphism*, 1(1), 1983.

[Car84] Luca Cardelli. Compiling a functional language. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, August 1984.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified framework for static analysis of programs by construction or approximation of fixpoints. In *Fourth Annual ACM Symposium on Principles of Programming Languages*. ACM, 1977.

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In *Symposium on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[Che80] Leslie Paul Chew. An improved algorithm for computing with equations. In *21st Annual Symposium on Foundations of Computer Science*, pages 108–117. IEEE, 1980.

[Che81] Leslie Paul Chew. Unique normal forms in term rewriting systems with repeated variables. In *13th Annual ACM Symposium on Theory of Computing*, pages 7–18, 1981.

[CO91] Yiyun Chen and Michael James O'Donnell. Testing confluence of nonterminating overlapping systems of rewrite rules. In *Conditional and Typed Rewriting Systems 2nd International CTRS Workshop, Montreal, June 1990*, volume 516 of *Lecture Notes in Computer Science*, pages 127–136. Springer-Verlag, 1991.

[Coh80] Donald L. Cohn. *Measure Theory*. Birkhauser, Boston, MA, 1980.

[Coh81] J. Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3), September 1981.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[DHJP83] Nachum Dershowitz, Jieh Hsiang, N. Josephson, and David A. Plaisted. Associative-commutative rewriting. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 940–944, August 1983.

[DKP91] Nachum Dershowitz, Simon Kaplan, and David A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite. *Theoretical Computer Science*, 83:71–96, 1991.

[DRT81] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *Conference Record of the Thirteenth*

Annual ACM Symposium on Principles of Programming Languages, pages 105–116. ACM, 1981.

[DS93] Irène Durand and Bruno Salinier. Constructor equivalent term rewriting systems. *Information Processing Letters*, 47, 1993.

[DSS91] Irène Durand, David J. Sherman, and Robert I. Strandh. Optimization of equational programs using partial evaluation. In *Proceedings of the ACM/IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation, New Haven, CT*, 1991.

[Dur94] Irène Durand. Bounded, strongly sequential, and forward-branching term-rewriting systems. *Journal of Symbolic Computation*, 1994. to appear.

[Fay79] M. Fay. First order unification in equational theories. In *Proceedings of the 4th Workshop on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 161–167. Springer-Verlag, 1979.

[FGJM85] K. Futatsugi, Joseph A. Goguen, J.-P. Jouannaud, and José Meseguer. Principles of OBJ2. In *12th Annual Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[FT90] John Field and Tim Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 307–322. ACM Press, 1990.

[FW76] Daniel Friedman and David S. Wise. Cons should not evaluate its arguments. In *3rd International Colloquium on Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, 1976.

[Gal86] Jean H. Gallier. *Logic for Computer Science—Foundations of Automatic Theorem Proving*. Harper & Row, New York, NY, 1986.

[GAL92] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 15–26. ACM, 1992.

[GH78] John V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):1–26, 1978.

[GKM83] John V. Guttag, Deepak Kapur, and David Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM Journal on Computing*, 12:189–214, 1983.

[Gog77] Joseph A. Goguen. Abstract errors for abstract data types. In E. J. Neuhold, editor, *Proceedings of IFIP Working Conference on Formal Description of Program Concepts*. North-Holland, 1977.

[Hen80] P. Henderson. *Functional Programming—Application and Implementation*. Prentice-Hall, 1980.

[HL91] Gérard Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic—Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, Cambridge, MA, 1991.

[HM76] P. Henderson and J. H. Morris. A lazy evaluator. In *3rd Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. SIGPLAN and SIGACT, 1976.

[HO79] C. M. Hoffmann and M. J. O'Donnell. Interpreter generation using tree pattern matching. In *6th Annual Symposium on Principles of Programming Languages*, pages 169–179. SIGPLAN and SIGACT, 1979.

[HO82] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):169–179, 1982.

[HOS85] C. M. Hoffmann, M. J. O'Donnell, and R. I. Strandh. Implementation of an interpreter for abstract equations. *Software — Practice and Experience*, 15(12):1185–1203, 1985.

[HS88] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional I/O systems. Technical Report YALEU/DCS/RR665, Yale University, New Haven, CT, December 1988.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[Hud92] Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting. *Journal of the ACM*, 27(4):797–821, October 1980.

[Hug82] R. J. M. Hughes. Super-combinators: A new implementation method for applicative languages. In *ACM Symposium on Lisp and Functional Programming*, August 1982.

[Hug85a] J. Hughes. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 129–146. Springer-Verlag, 1985.

[Hug85b] R. J. M. Hughes. Strictness detection in non-flat domains. In Neil Jones and Harald Ganzinger, editors, *Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[Ive62] K. E. Iverson. *A Programming Language*. John Wiley and Sons, Inc., New York, NY, 1962.

[Jay85] Bharat Jayaraman. Equational programming: A unifying approach to functional and logic programming. Technical Report 85-030, The University of North Carolina, 1985.

[Jay92] Bharat Jayaraman. Implementation of subset-equational programming. *The Journal of Logic Programming*, 12(4), April 1992.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, 1984. SIGPLAN Notices 19(6) June, 1984.

[Kar64] Carol R. Karp. *Languages with Expressions of Infinite Length*. North-Holland, Amsterdam, 1964.

[Kar81] K. Karlsson. Nebula, a functional operating system. Technical report, Chalmers University, 1981.

[KB70] Donald E. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 127–146. Pergamon Press, Oxford, 1970.

[KKN82] Deepak Kapur, M. S. Krishnamoorthy, and P. Narendran. A new linear algorithm for unification. Technical Report 82CRD-100, General Electric, 1982.

[KKSd91] J. R. Kenneway, Jan Willem Klop, M. R. Sleep, and F. J. de Vries. Transfinite reductions in orthogonal term rewriting systems. In *Proceedings of the 4th International Conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[KL89] Philip J. Koopman and Peter Lee. A fresh look at combinator graph reduction. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, October 1989.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematisch Centrum, Amsterdam, 1980.

[Klo91] Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, Oxford, 1991.

[KM91] Jan Willem Klop and A. Middeldorp. Sequentiality in orthogonal term rewriting systems. *Journal of Symbolic Computation*, 12:161–195, 1991.

[Knu73] Donald E. Knuth. *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 1973.

[Koo90] Philip J. Koopman. *An Architecture for Combinator Graph Reduction*. Academic Press, Boston, MA, 1990.

[KP78] Gilles Kahn and Gordon Plotkin. Domaines concrets. Technical report, IRIA Laboria, LeChesnay, France, 1978.

[KS86] R. M. Keller and M. R. Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems*, 8(1):88–108, 1986.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30. ACM, 1990.

[Lan65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, 1965.

[Las91] Jean-Louis Lassez. From LP to CLP: Programming with constraints. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software: International Conference*, volume 526 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Lév78] Jean-Jacques Lévy. *Reductions Correctes et Optimales dans le Lambda-Calcul*. PhD thesis, Université Paris, January 1978.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Lov78] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*. Elsevier North-Holland, New York, NY, 1978.

[MAE+65] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, MA, 1965.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[Mes92] José Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proceedings of the 3rd International Conference on Algebraic and Logic Programming, Volterra, Italy, September 1992*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[Mit68] D. Mitchie. 'Memo' functions and machine learning. *Nature*, 1968.

[MM76] A. Martelli and U. Montanari. Unification in linear time and space: A structured presentation. Technical Report B76-16, Institut di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, 1976.

[MY78] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Theory of Computation. North-Holland, New York, NY, 1978.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[New42] M. H. A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[O'D77] Michael James O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.

[O'D79] Michael James O'Donnell. Letter to the editor, SIGACT News, v. 11, no. 2, 1979. p. 2.

[O'D85] Michael James O'Donnell. *Equational Logic as a Programming Language*. Foundations of Computing. MIT Press, Cambridge, MA, 1985.

[O'D87] Michael James O'Donnell. Tree-rewriting implementation of equational logic programming. In Pierre Lescanne, editor, *Rewriting Techniques and Applications — Bordeaux, France, May 1987 — Proceedings*, volume 256 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[PA85] Keshav Pingali and Arvind. Efficient demand-driven evaluation, part 1. *ACM Transactions on Programming Languages and Systems*, 7(2):311–333, April 1985.

[PA86] Keshav Pingali and Arvind. Efficient demand-driven evaluation, part 2. *ACM Transactions on Programming Languages and Systems*, 8(1):109–139, January 1986.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[PT89] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 315–328. ACM, 1989.

[Reb92] Samuel A. Rebelsky. I/O trees and interactive lazy functional programming. In Maurice Bruynooghe and Martin Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 458–472. Springer-Verlag, August 1992.

[Reb93] Samuel A. Rebelsky. *Tours, a System for Lazy Term-Based Communication*. PhD thesis, The University of Chicago, June 1993.

[RS90] I. V. Ramakrishnan and R. C. Sekar. Programming in equational logic: Beyond strong sequentiality. In *Proceedings of the IEEE Conference on Logic in Computer Science*, 1990.

[RW69] G. A. Robinson and L. Wos. Paramodulation and theorem-proving in first-order logic with equality. *Machine Intelligence*, 4:135–150, 1969.

[She90] David J. Sherman. Lazy directed congruence closure. Technical Report 90-028, The University of Chicago, 1990.

[SLR78] Jay M. Spitzen, Karl N. Levitt, and Lawrence Robinson. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–1075, December 1978.

[Sta82] John Staples. Two-level expression representation for faster evaluation. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Graph Grammars and their Application to Computer Science: 2nd International Workshop*, volume 153 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[Str84] R. I. Strandh. Incremental suffix trees with multiple subject strings. Technical Report JHU/EECS-84/18, The Johns-Hopkins University, 1984.

[Str88] Robert I. Strandh. *Compiling Equational Programs into Efficient Machine Code*. PhD thesis, The Johns Hopkins University, Baltimore, MD, 1988.

[Str89] Robert I. Strandh. Classes of equational programs that compile into efficient machine code. In *Proceedings of the Third International Conference on Rewrite Techniques and Applications*, 1989.

[Tha85] Satish Thatte. On the correspondence between two classes of reduction systems. *Information Processing Letters*, 1985.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[Wan76] Mitchell Wand. First order identities as a defining language. *Acta Informatica*, 14:336–357, 1976.

[War83] David H. D. Warren. An abstract prolog instruction set. Technical Report 309, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1983.

[Win93] Glynn Winksel. *The Formal Semantics of Programming Languages—An Introduction*. Foundations of Computing. MIT Press, Cambridge, MA, 1993.

Index

Index

Index