Rule-based Computation and Deduction

Hélène Kirchner Pierre-Etienne Moreau

ESSLLI 2001

Contents

1 Introduction to term rewriting **5**
1.1 Introduction . . . 5
1.2 Rewriting as an abstract reduction system . . . 6
1.3 Rewriting as a relation on terms . . . 8
1.4 Rewriting as a logic for concurrent computations . . . 19
1.5 Superposition and simplification . . . 23
1.6 Further reading . . . 29

2 Introduction to the rewriting calculus **31**
2.1 Introduction . . . 31
2.2 Definition of the ρ_T-calculus . . . 34
2.3 Encoding λ-calculus and term rewriting in the ρ-calculus . . . 45
2.4 The confluence of ρ-calculus . . . 50
2.5 Conclusion . . . 57

3 Compilation of rule-based programs **59**
3.1 Introduction . . . 59
3.2 Preliminary concepts . . . 61
3.3 Many-to-one AC matching . . . 64
3.4 Optimisations . . . 72
3.5 Determinism analysis . . . 73
3.6 A compiler for ELAN . . . 79
3.7 Conclusion . . . 80

Chapter 1

Introduction to term rewriting

H. Kirchner [Kir99]

1.1 Introduction

In the area of system specifications, rewriting techniques have been developed for two main applications: prototyping algebraic specifications of user-defined data types and theorem proving related to program verification. A first formulation of term rewriting has been proposed by Evans [Eva51] and later by Knuth and Bendix [KB70]. Originally its main purpose was for generating *canonical term rewriting systems* which can be used as decision procedures for proving the validity of equalities in some equational theories. Then in addition to the validity problem of equational logic, term rewriting has also been applied to inductive theorem proving, checking consistency and completeness of equational or conditional specifications, first-order theorem proving, unification theory, geometry theorem proving, etc. Through various implementations and experiments in automated theorem proving, it has been demonstrated that simplification of formulas by rewriting is indeed an effective way of pruning the search space. On the other hand, with the emergence of equationally specified abstract data types in the late 1970's, term rewriting has gained considerable popularity also as a bridge between programming language theory and program verification. Several specification languages or programming environments, such as LARCH, OBJ, ASF, RAP, MAUDE, ELAN, to cite a few, are using rewriting as their basic evaluation mechanism. Rewriting techniques are also used in functional programming languages such as ML.

This chapter introduces term rewriting and some of its applications from different points of view. Rewriting is first presented in Section 1.2 as an abstract relation on a set, and properties of such relations are introduced, mainly confluence and well-foundedness. Confluence is the key concept to ensure determinism of a rewriting relation seen as a computation process. It says that two computations starting from a same term will eventually give the same result. Well-foundedness ensures that any computation terminates. Then in Section 1.3 a more concrete notion of rewriting is defined on first-order terms. Several examples of rewriting relations on terms or equivalence classes of terms are then given, including rewriting modulo axioms and conditional rewriting. Emphasis is put in this part on the Church-Rosser properties of these different relations, and their application to decide equality in equational or conditional theories. Then in Section 1.4, the rewriting logic is defined and shown especially suited to describe concurrent computations. It also provides a logical framework in which other logics can be represented and a semantic framework for the specification of languages and systems. Experiments with applications, such as the specification of constraint solving systems or theorem provers, reveal the importance of the concept of strategy to restrict the computation space or to select relevant computations.

The rest of the chapter is concerned with applications of rewriting to proofs of various program properties. Section 1.5 addresses the question of building automatically, for an equational theory, an equivalent convergent rewrite system, when one exists. This is answered via a completion process that orients equalities into rewrite rules, computes new derived equalities and reduces them. When this process terminates, the

result is a confluent and terminating set of rewrite rules. Several generalisations of the initial completion mechanism, based of superposition and simplification, are also surveyed in this section.

1.2 Rewriting as an abstract reduction system

Most of basic definitions and properties of rewrite systems can be stated abstractly, by considering rewriting as a binary relation on sets. This point of view, taken from [Klo92], has the great advantage of defining these properties once for all for different rewriting relations.

Definition 1.2.1 *An* abstract reduction system *is a structure $\langle \mathcal{T}, \longrightarrow \rangle$ consisting of a set $\mathcal{T}$ and a binary relation $\longrightarrow$ on $\mathcal{T}$.*

For each binary relation $\longrightarrow$, $\xrightarrow{+}$ and $\xrightarrow{*}$ respectively denote its transitive and its reflexive transitive closure. On a set $\mathcal{T}$ whose elements are denoted by $t, t', \ldots$, we also define:

$$\begin{array}{lll} t \longleftarrow t' & \text{if} & t' \longrightarrow t \\ t \longleftrightarrow t' & \text{if} & t \longrightarrow t' \text{ or } t \longleftarrow t' \\ t \xleftarrow{*} t' & \text{if} & t' \xrightarrow{*} t \\ t \xleftarrow{+} t' & \text{if} & t' \xrightarrow{+} t. \end{array}$$

For any natural number n, the composition of n steps of $\longrightarrow$ or $\longleftrightarrow$ is denoted by $\xrightarrow{n}$ or $\xleftrightarrow{n}$ respectively. By convention, $\xrightarrow{0}$ and $\xleftrightarrow{0}$ are nothing else but syntactic equality. As usual, $\xleftrightarrow{+}$ and $\xleftrightarrow{*}$ denote respectively the transitive and the reflexive transitive closure of $\longleftrightarrow$.

Composition of binary relations $\longrightarrow_1$ followed by $\longrightarrow_2$ will be denoted by $\longrightarrow_1 \circ \longrightarrow_2$.

Termination and confluence are two properties often required for a reduction system. They are defined here as well as their relationship with other properties especially local confluence, the Church-Rosser property and the existence of normal forms.

Definition 1.2.2 *Let $\langle \mathcal{T}, \longrightarrow \rangle$ be an abstract reduction system.*

- *An element $t \in \mathcal{T}$ is a* $\longrightarrow$-normal form *if there exists no $t' \in \mathcal{T}$ such that $t \longrightarrow t'$. Furthermore t' is called a normal form of $t \in \mathcal{T}$ if $t \xrightarrow{*} t'$ and t' is a $\longrightarrow$-normal form. This will be later denoted by $t \xrightarrow{!} t'$.*
- *The relation $\longrightarrow$ is* terminating *(or* strongly normalising*, or* well-founded*) if there is no infinite sequence of related elements $t_1 \longrightarrow t_2 \longrightarrow \ldots \longrightarrow t_i \longrightarrow \ldots$.*
- *The relation $\longrightarrow$ is* weakly terminating *(or weakly normalising) if every element $t \in \mathcal{T}$ has a normal form.*
- *The relation $\longrightarrow$ has the* unique normal form property *if for any $t, t' \in \mathcal{T}$, $t \xleftrightarrow{*} t'$ and t, t' are normal forms imply $t = t'$. The unique normal form of t when it exists is denoted $t\!\downarrow$.*

Example 1.2.1 *Let a, b, c, d be distinct elements of $\mathcal{T}$.*

1. *The relation $a \longrightarrow b$ is terminating, but $a \longrightarrow a$ is not.*
2. *The relation defined by $a \longrightarrow b, b \longrightarrow a, a \longrightarrow c, b \longrightarrow d$ is weakly terminating but not terminating. It does not have the unique normal form property.*

The termination property of an abstract relation associated with an abstract reduction system $\langle \mathcal{T}, \longrightarrow \rangle$ is studied in the general framework of well-founded orderings on the set $\mathcal{T}$.

Proposition 1.2.1 [MN70] *Let $\langle \mathcal{T}, \longrightarrow \rangle$ be an abstract reduction system. The relation $\longrightarrow$ is terminating on $\mathcal{T}$ if and only if there exists a well-founded ordering $>$ over $\mathcal{T}$ such that $s \longrightarrow t$ implies $s > t$.*

For abstract reduction systems, the Church-Rosser property is expressed as follows:

Definition 1.2.3 *The relation* $\longrightarrow$ *is* Church-Rosser *on a set* $\mathcal{T}$ *if* $\langle \mathcal{T}, \longrightarrow \rangle$ *satisfies*

$$\overset{*}{\longleftrightarrow} \subseteq \overset{*}{\longrightarrow} \circ \overset{*}{\longleftarrow} .$$

A relation is confluent if it satisfies the diamond property defined as follows:

Definition 1.2.4 *The relation* $\longrightarrow$ *is* confluent *on* $\mathcal{T}$ *if* $\langle \mathcal{T}, \longrightarrow \rangle$ *satisfies*

$$\overset{*}{\longleftarrow} \circ \overset{*}{\longrightarrow} \quad \subseteq \quad \overset{*}{\longrightarrow} \circ \overset{*}{\longleftarrow} .$$

Since $\overset{*}{\longleftarrow} \circ \overset{*}{\longrightarrow} \quad \subseteq \quad \overset{*}{\longleftrightarrow}$, the Church-Rosser property obviously implies the confluence. The converse is shown by induction on the number of $\longleftrightarrow$-steps that appear in $\overset{*}{\longleftrightarrow}$.

Theorem 1.2.1 *The relation* $\longrightarrow$ *is confluent if and only if it is Church-Rosser.*

In the following, $t \downarrow t'$ means that both t and t' have a common descendant, i.e. a element t'' such that $t \overset{*}{\longrightarrow} t'' \overset{*}{\longleftarrow} t'$. In this case, the pair (t, t') is said to be *convergent.*

If an element t has two normal forms t_1 and t_2, then by confluence $t_1 \downarrow t_2$ and thus $t_1 = t_2$ since both are normal forms. So we get the next result:

Proposition 1.2.2 *If* $\longrightarrow$ *is confluent, the normal form of any element is unique, provided it exists.*

The confluence property has a local version that only considers two different applications of the relation.

Definition 1.2.5 *The relation* $\longrightarrow$ *is* locally confluent *on* $\mathcal{T}$ *if* $\langle \mathcal{T}, \longrightarrow \rangle$ *satisfies:*

$$\longleftarrow \circ \longrightarrow \quad \subseteq \quad \overset{*}{\longrightarrow} \circ \overset{*}{\longleftarrow} .$$

Confluence clearly implies local confluence but the converse is not true, as shown by the next example:

Example 1.2.2 *Consider four distinct elements* a, b, c, d *of* $\mathcal{T}$ *and the relation defined by* $a \longrightarrow b, b \longrightarrow a, a \longrightarrow c, b \longrightarrow d$. *The relation, although locally confluent (consider any possible case of ambiguity) and weakly terminating, is not confluent since* $c \overset{*}{\longleftarrow} a \overset{*}{\longrightarrow} d$ *but neither* c *nor* d *can be rewritten.*

In this example of course, the relation is not terminating since there exists a cycle $a \longrightarrow b \longrightarrow a$. Provided the relation terminates, confluence and local confluence are equivalent.

Lemma 1.2.1 [New42] *If* $\longrightarrow$ *is terminating, then* $\longrightarrow$ *is confluent if and only if* $\longrightarrow$ *is locally confluent.*

Then the Church-Rosser property is equivalent to the local confluence property, under the termination assumption. The results are summarised in the following theorem.

Theorem 1.2.2 *If the relation* $\longrightarrow$ *is terminating, the following properties are equivalent:*

1. $\longrightarrow$ *is Church-Rosser.*
2. $\longrightarrow$ *is confluent.*
3. $\longrightarrow$ *is locally confluent.*
4. *for any* $t, t' \in \mathcal{T}, t \overset{*}{\longleftrightarrow} t'$ *if and only if* $t \downarrow = t' \downarrow$.

(1) and (2) are equivalent by Theorem 1.2.1, (2) and (3) are equivalent by Lemma 1.2.1. (4) clearly implies (1). Finally (1) implies (4) by applying the Church-Rosser property to $t \downarrow$ and $t' \downarrow$.

Definition 1.2.6 *A relation* $\longrightarrow$ *is* convergent *if it is confluent and terminating.*

Other notions of confluence appear in the literature. Let $\stackrel{0,1}{\longrightarrow}$ denote at most one application of a rewriting step, and $\stackrel{0,1}{\longleftarrow}$ the symmetric relation.

Definition 1.2.7 *Let $\langle \mathcal{T}, \longrightarrow \rangle$ an abstract reduction system. The relation $\longrightarrow$ is* strongly confluent *on $\mathcal{T}$ if:*

$$\longleftarrow \circ \longrightarrow \subseteq \stackrel{0,1}{\longrightarrow} \circ \stackrel{0,1}{\longleftarrow}.$$

Strong confluence implies confluence [New42]. Confluence of $\longrightarrow$ coincides with strong confluence of $\stackrel{*}{\longrightarrow}$. This property is used in classical proofs of the Church-Rosser property for the lambda-calculus [Bar84]. In general, this may be of some help to establish the confluence of non-terminating systems. Concerning systems that are only weakly normalising, a method for establishing confluence has been proposed in [CG91, Har89]. The main idea of the following result is to establish a relation between two abstract reduction systems in such a way that one will inherit of the confluence property of the other.

Theorem 1.2.3 [CG91] *Let $\langle \mathcal{T}, \longrightarrow \rangle$ be a weakly terminating abstract reduction system and $\langle \mathcal{T}', \longrightarrow' \rangle$ be a confluent abstract reduction system such that there exists a mapping π from $\mathcal{T}$ to $\mathcal{T}'$ satisfying:*

1. *for all elements r and s in $\mathcal{T}$, if $r \longrightarrow s$ then $\pi(r) \stackrel{*}{\longleftrightarrow}' \pi(s)$,*
2. *The image with π of a normal form in $\langle \mathcal{T}, \longrightarrow \rangle$ is a normal form in $\langle \mathcal{T}', \longrightarrow' \rangle$.*
3. *π is injective on the normal forms in $\langle \mathcal{T}, \longrightarrow \rangle$.*

Under these conditions, $\langle \mathcal{T}, \longrightarrow \rangle$ is confluent.

This result can be specialised to several cases, in particular when $\mathcal{T} = \mathcal{T}'$ and $\pi = Id$. One can also notice that if $\langle \mathcal{T}', \longrightarrow' \rangle$ has the unique normal form property, then under the previous hypothesis on $\langle \mathcal{T}, \longrightarrow \rangle$ and π, $\langle \mathcal{T}, \longrightarrow \rangle$ has also the unique normal form property. This has been applied for instance to prove the confluence of the $\lambda\beta$-calculus, as developed in [CG91].

1.3 Rewriting as a relation on terms

Abstract reduction systems are now specialised to terms and equivalence classes of terms. In this context, the Church-Rosser property is especially interesting since it provides a decision procedure for equality.

For the usual notions and operations on terms and substitutions, the reader can refer to [DJ90]. In this chapter, $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of terms built on function symbols $\mathcal{F}$ and variables $\mathcal{X}$. $\mathcal{T}(\mathcal{F})$ is the set of ground terms, i.e. without variables. $\mathcal{V}ar\, t$ denotes the set of variables of a term t, $t_{|\omega}$ the subterm of t at position ω, and $t[u]_\omega$ the term t that contains the subterm u at position ω (ω may be omitted). Greek letters $\sigma, \theta, \ldots$ are used to denote substitutions, that are completely defined by a mapping of variables to terms written $\sigma = (x_1 \mapsto t_1) \ldots (x_n \mapsto t_n)$ when $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$. The *domain* of the substitution σ is the finite set of variables $\mathcal{D}om(\sigma) = \{x \mid x \in \mathcal{X} \text{ and } \sigma(x) \neq x\}$, the *range* of σ is the set of terms $\mathcal{R}an(\sigma) = \cup_{x \in \mathcal{D}om(\sigma)} \sigma(x)$. A substitution σ is *ground* if $\mathcal{R}an(\sigma) \subseteq \mathcal{T}(\mathcal{F})$. Composition of σ and ρ is denoted $\rho\sigma$.

1.3.1 Term rewriting

The notion of abstract reduction system can be instantiated by considering $\mathcal{T}$ as the set of terms and $\longrightarrow$ as the rewriting relation generated by a set of rewrite rules.

Rewrite systems.

Definition 1.3.1 *A* rewrite rule *is an ordered pair of terms denoted $l \to r$. The terms l and r are respectively called the* left-hand side *and the* right-hand side *of the rule. In a rewrite rule, every variable occurring in the right-hand side of a rule also occurs in the left-hand side. All variables are implicitly universally quantified. A* rewrite system *or* term rewriting system *is a (finite or infinite) set of rewrite rules.*

Example 1.3.1 *Combinatory Logic, originally devised by Schönfinkel (1924) and then by Curry, is an example of a theory with a binary function symbol $\cdot$ (application), constant symbols S, K, I (combinators) and variables. The theory is defined by three axioms that can be applied as rewrite rules, using the following rewrite system CL:*

$$\begin{array}{rcl} ((S \cdot x) \cdot y) \cdot z & \longrightarrow & (x \cdot z) \cdot (y \cdot z) \\ (K \cdot x) \cdot y & \longrightarrow & x \\ (I \cdot x) & \longrightarrow & x. \end{array}$$

The symbol $\cdot$ is often omitted for better readability, as well as parentheses associating terms from left to right. For instance xyz stands for $(x \cdot y) \cdot z$.

A rule is applied by replacing an instance of the left-hand side by the same instance of its right-hand side, but never the converse, contrary to equalities. Note that two rules are considered to be the same if they only differ by a renaming of their variables.

Definition 1.3.2 *Given a rewrite system R, a term t* rewrites *to a term t', which is denoted by $t \longrightarrow_R t'$ if there exist a rule $l \to r$ of R, a position ω in t, a substitution σ, satisfying $t_{|\omega} = \sigma(l)$ and called* match *from l to $t_{|\omega}$, such that $t' = t[\sigma(r)]_\omega$.*

When t rewrites to t' with a rule $l \to r$ and a substitution σ, it will be always assumed that variables of l and t are disjoint. This is not a restriction, since variables of rules can be renamed without loss of generality. When either the rewrite system, the rule, the substitution and/or the position need to be made precise, a rewriting step is denoted by $t \longrightarrow_R^{\omega,\sigma,l\to r} t'$. The subterm $t_{|\omega}$ where the rewriting step is applied is called the *redex*. A term that has no redex is said to be *R-irreducible* or in *R-normal form*. An R-irreducible form of t is denoted by $t{\downarrow_R}$. A *rewriting derivation* is any sequence of rewriting steps $t_1 \longrightarrow_R t_2 \longrightarrow_R \ldots$. A rewrite system R induces a reflexive transitive binary relation on terms called the *rewriting relation* or the *derivability relation* denoted by $\stackrel{*}{\longrightarrow}_R$.

Example 1.3.2 *In Combinatory Logic (cf. Example 1.3.1), we can derive*

$$SKKx \longrightarrow_{CL} Kx(Kx) \longrightarrow_{CL} x.$$

We also have

$$S(KS)Kxyz \longrightarrow_{CL} KSx(Kx)yz \longrightarrow_{CL} S(Kx)yz \longrightarrow_{CL} Kxz(yz) \longrightarrow_{CL} x(yz)$$

Abbreviating $S(KS)K$ by B, this establishes that $Bxyz \stackrel{*}{\longrightarrow}_{CL} x(yz)$ and defines Bxy as the composition $x \circ y$ where x and y are variables representing functions.
Another interesting combinator is SII often abbreviated as Ω and called self-applicator. This name is justified by the following derivation:

$$\Omega x = SIIx \longrightarrow_{CL} Ix(Ix) \longrightarrow_{CL} Ixx \longrightarrow_{CL} xx.$$

But the term $\Omega\Omega$ admits a cyclic derivation, since: $\Omega\Omega \stackrel{+}{\longrightarrow}_{CL} \Omega\Omega$.
It can also be proved that any term F in Combinatory Logic has a fixed point P defined as $\Omega(BF\Omega)$. Indeed:

$$P = \Omega(BF\Omega) \stackrel{*}{\longrightarrow}_{CL} (BF\Omega)(BF\Omega) \stackrel{*}{\longrightarrow}_{CL} F(\Omega(BF\Omega)) = FP.$$

Termination.

Working now on terms instead of an abstract set $\mathcal{T}$ yields methods for proving termination of term rewriting. The rewrite relation $\longrightarrow_R$ is terminating if it is included into a suitable well-founded ordering. A quasi-ordering is a reflexive transitive binary relation and is well-founded if its strict part is well-founded. A *reduction (quasi-)ordering* $>$ is a (quasi-)ordering closed under context and substitution, i.e. such that for

any term t and any substitution σ, if $u > s$ then $t[u] > t[s]$ and $\sigma(u) > \sigma(s)$. If there exists a well-founded reduction ordering $>$ on terms such that $l > r$ for any rewrite rule $(l \to r) \in R$, then $\longrightarrow_R$ is terminating.

There are essentially two types of well-founded reduction orderings on terms, surveyed in [Der87]. The first one, called *syntactical*, provides the ordering via an analysis of the term structure. Among these orderings are the *recursive path ordering* [Der82], the *lexicographic path ordering* [KL82] and the *recursive decomposition ordering* [JLR82, Les90]. These orderings are convenient since they are based on a concept of precedence which is a partial order on the signature. In addition they enjoy a property called *incrementality* that enables one to handle an incremental set of ordering problems. This is particularly useful when a new equality must be ordered and added to a set of rewrite rules without modifying their previous orientations. Another family consists of *semantical orderings*, that interpret the terms in another structure where a well-founded ordering is known. For such a purpose, two common ordered sets are the natural numbers and the terms ordered by a syntactical ordering. The first choice enables one to consider functions over natural numbers which are stable under substitutions. The most frequently used are polynomials [Lan79, CL87, Ste90, CL92, Gie95]. The set of terms ordered by a syntactical ordering may also be chosen as a target for the "semantics" [BD86, BL87]. Both methods can be combined as in [KB70] that uses a semantical ordering (polynomial interpretation of degree 1) and a syntactical ordering. A general path ordering dealing with semantical as well as syntactical aspects has more recently been proposed in [DH95].

Syntactical orderings contain the important family of simplification orderings. A *simplification ordering* is a reduction ordering such that a term is greater than any of its subterms. The last subterm condition suffices for ensuring well-foundedness of simplification orderings, provided that the signature is finite. Simplification orderings can be built from a well-founded ordering on the function symbols $\mathcal{F}$ called a *precedence*. A simple example of path ordering is the *multiset path ordering*. To understand the next definition, the reader should know that a multiset over a set $\mathcal{T}$ is a function M from $\mathcal{T}$ to natural numbers. Intuitively for any element x in $\mathcal{T}$, $M(x)$ specifies the number of occurrences of x in M. A multiset is finite if $M(x) = 0$ for all but finitely many x.

Let $>$ be a partial ordering on $\mathcal{T}$. Its multiset extension $>^{mult}$ is the transitive closure of the following relation on finite multisets over $\mathcal{T}$:

$$\{s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n\} >^{mult} \{s_1, \ldots, s_{i-1}, t_1, \ldots, t_k, s_{i+1}, \ldots, s_n\}$$

if for some element s_i, there exist k elements, $k \geq 0$, such that $s_i > t_j$ for any $j = 1, \ldots, k$.
If $>$ is well-founded on $\mathcal{T}$, its multiset extension $>^{mult}$ is well-founded on finite multisets over $\mathcal{T}$.

Definition 1.3.3 *Let $>_{\mathcal{F}}$ be a precedence on $\mathcal{F}$. The* multiset path ordering *$>_{mpo}$ is defined on ground terms by $s = f(s_1, .., s_n) >_{mpo} t = g(t_1, \ldots, t_m)$ if one at least of the following conditions holds:*

1. *$f = g$ and $\{s_1, \ldots, s_n\} >^{mult}_{mpo} \{t_1, \ldots, t_m\}$*
2. *$f >_{\mathcal{F}} g$ and $\forall j \in \{1, \ldots, m\}, s >_{mpo} t_j$*
3. *$\exists i \in \{1, \ldots, n\}$ such that either $s_i >_{mpo} t$ or $s_i \sim t$*
   where $\sim$ means equivalent up to permutation of subterms.

Note that the precedence on $\mathcal{F}$ need not be total and that the condition $f = g$ could be replaced by $f \sim_{\mathcal{F}} g$ where $\sim_{\mathcal{F}}$ is the equivalence induced on $\mathcal{F}$ by $>_{\mathcal{F}}$: $f \sim_{\mathcal{F}} g$ if $f >_{\mathcal{F}} g$ and $g >_{\mathcal{F}} f$.

Example 1.3.3 *Let h, k, f be respectively $1, 2, 2$-ary function symbols and x, y be variables. Clearly by the subterm relation: $h(f(x, y)) >_{mpo} f(x, y) >_{mpo} y$.*
Assuming a precedence where $k >_{\mathcal{F}} f$, $k(f(x, y), z) >_{mpo} f(x, k(z, y)) >_{mpo} y$, since both $k(f(x, y), z) >_{mpo} x$ and $k(f(x, y), z) >_{mpo} k(z, y)$. The latter holds since $f(x, y) >_{mpo} y$.

Another promising direction for proving termination of rewrite systems has been opened more recently by [AG96, AG97]. Looking at rewrite systems as programs, the idea is that each rule whose left-hand side begins with f is defining this function f. Let f be called a *defined symbol*. Such a program is terminating if any recursive call is terminating. This motivates the idea that only subterms of the right-hand sides that have a defined symbol at top, have to be considered for the examination of the termination behaviour.

Clearly a term without defined symbol is in R-normal form. If no defined symbol occurs in any right-hand side of rules, any term can only finitely many times be reduced by R. Thus infinite reductions originate from the fact that defined symbols are introduced by the right-hand sides of rules. By tracing the introduction of these defined symbols, information can be obtained about termination of R. This is the idea behind the dependency pairs criterion. In the next definition, to avoid handling tuples, an auxiliary symbol F is introduced for any defined symbol f.

Definition 1.3.4 *If $f(t_1, \ldots, t_n) \to t[g(s_1, \ldots, s_m)]$ is a rule in R where f and g are defined symbols, and t is some context term, then $(F(t_1, \ldots, t_n), G(s_1, \ldots, s_m))$ is a dependency pair of R.*

A R-chain is a sequence of renamings of dependency pairs such that there exists a substitution σ with $\sigma(t_i) \xrightarrow{*}_R \sigma(s_{i+1})$ for every two consecutive pairs (s_i, t_i) and (s_{i+1}, t_{i+1}) in the sequence.

Theorem 1.3.1 *A rewrite system R is terminating if and only if no infinite R-chain exists.*

Since termination is undecidable, the criterion given by Theorem 1.3.1 is undecidable too. The technique proposed in [AG96, AG97] is the following: a dependency graph is built, whose nodes are labelled with the dependency pairs. There is an arc from (s, t) to (u, v) if there exists a substitution σ such that $\sigma(t) \xrightarrow{*}_R \sigma(u)$. Since this last property is in general undecidable, the dependency graph is actually approximated by a super-graph with the same cycles. Dependency pairs are then used to generate a set of inequalities. If these equalities can be satisfied by a suitable well-founded ordering, R is terminating.

Theorem 1.3.2 *If there exists a reduction quasi-ordering $\geq$ such that $l \geq r$ for each rule $l \to r$ in R, $s \geq t$ for each dependency pair (s, t) on a cycle of the dependency graph, and $s > t$ for at least one dependency pair (s, t) on every cycle of the dependency graph, then R is terminating.*

Example 1.3.4 *In the following rewrite system, defined symbols are minus and quot.*

$$\begin{array}{rcl} \forall x, y, & & \\ minus(x, 0) & \to & x \\ minus(s(x), s(y)) & \to & minus(x, y) \\ quot(0, s(y)) & \to & 0 \\ quot(s(x), s(y)) & \to & s(quot(minus(x, y), s(y))) \end{array}$$

The associated dependency pairs are

$$\begin{array}{rcl} (MINUS(s(x), s(y)) & , & MINUS(x, y)) \\ (QUOT(s(x), s(y)) & , & MINUS(x, y)) \\ (QUOT(s(x), s(y)) & , & QUOT(minus(x, y), s(y))) \end{array}$$

The dependency graph has two cycles corresponding to the 1st and 3rd dependency pairs. To prove termination, it is sufficient to find a reduction quasi-ordering that satisfies the following inequalities.

$$\begin{array}{rcl} minus(x, 0) & \geq & x \\ minus(s(x), s(y)) & \geq & minus(x, y) \\ quot(0, s(y)) & \geq & 0 \\ quot(s(x), s(y)) & \geq & s(quot(minus(x, y), s(y))) \\ MINUS(s(x), s(y)) & > & MINUS(x, y) \\ QUOT(s(x), s(y)) & > & QUOT(minus(x, y), s(y)) \end{array}$$

A polynomial interpretation in which 0 is mapped to 0, $s(x)$ to $x+1$, $minus(x, y), quot(x, y), MINUS(x, y), QUOT(x, y)$ all mapped to x, provides the solution.

Church-Rosser property.

Results valid for any abstract reduction system specialise to the set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ and the rewriting relation $\longrightarrow_R$. In this context, the Church-Rosser property states the relation between replacement of equals by equals and rewriting. Reformulating Definition 1.2.3 on page 7, we get:

The relation $\longrightarrow_R$ is Church-Rosser on $\mathcal{T}(\mathcal{F},\mathcal{X})$ if

$$\forall t, t',\ t \overset{*}{\longleftrightarrow}_R t' \text{ if and only if } \exists t'',\ t \overset{*}{\longrightarrow}_R t'' \overset{*}{\longleftarrow}_R t'.$$

Provided termination, this is equivalent to confluence:

$$\forall t, t_1, t_2,\ t_1 \overset{*}{\longleftarrow}_R t \overset{*}{\longrightarrow}_R t_2 \text{ implies } \exists t'',\ t_1 \overset{*}{\longrightarrow}_R t'' \overset{*}{\longleftarrow}_R t_2.$$

A rewrite system R is Church-Rosser, confluent, locally confluent, terminating, convergent on $\mathcal{T}(\mathcal{F},\mathcal{X})$ if $\longrightarrow_R$ is Church-Rosser, confluent, locally confluent, terminating, convergent on $\mathcal{T}(\mathcal{F},\mathcal{X})$.

A rewrite system R is *ground Church-Rosser*, *ground confluent*, *ground locally confluent*, *ground terminating*, *ground convergent* if $\longrightarrow_R$ is Church-Rosser, confluent, locally confluent, terminating, convergent on the set of ground terms $\mathcal{T}(\mathcal{F})$.

Example 1.3.5 *Right distributivity of an operator $+$ on an operator $*$, expressed by the rule $(x+y)*z \to (x*z)+(y*z)$, or left distributivity $x*(y+z) \to (x*y)+(x*z)$, taken separately, are confluent rewrite systems. Put together, the rewrite system composed of these two distributivity rules is terminating (using for instance the precedence $* >_{\mathcal{F}} +$ in Definition 1.3.3 on page 10), but not confluent since*

$$((x*x)+(x*y))+((y*x)+(y*y)) \overset{+}{\longleftarrow} (x+y)*(x+y) \overset{+}{\longrightarrow} ((x*x)+(y*x))+((x*y)+(y*y))$$

The following example is chosen in the area of hardware verification, and illustrates how to use rewrite systems to perform a simple proof of equivalence of two circuits.

Example 1.3.6 [Kri94] *A ripple-carry adder with n bits is composed of n full adders. Inputs of the i-th full adder are the i-th bits $p(i)$ and $q(i)$ of the two binary numbers to be added, and the carry $r(i)$ from the $(i-1)$-th adder. Outputs of the i-th full adder are the carry $r(i+1)$ and the i-th bit of the sum $sum(i)$. Inputs of the whole circuit are $p(0), q(0), r(0)$ and outputs are $sum(0), \ldots, sum(n)$ and $r(n)$. A ripple-carry adder is described by the following term rewriting system, where for a better readalibility, parentheses for associative boolean operators $\oplus$ and $\vee$ are omitted.*

$$\begin{array}{lcll}
sort & & Nat,\ Bool & \\
\oplus & : & Bool \times Bool & \mapsto\ Bool \\
\wedge & : & Bool \times Bool & \mapsto\ Bool \\
\vee & : & Bool \times Bool & \mapsto\ Bool \\
p & : & Nat & \mapsto\ Bool \\
q & : & Nat & \mapsto\ Bool \\
r & : & Nat & \mapsto\ Bool \\
sum & : & Nat & \mapsto\ Bool \\
s & : & Nat & \mapsto\ Nat
\end{array}$$

$$\begin{array}{rcl}
\forall i : Nat, & & \\
sum(i) & \to & p(i) \oplus q(i) \oplus r(i) \\
r(s(i)) & \to & (p(i) \wedge q(i)) \vee (p(i) \wedge r(i)) \vee (q(i) \wedge r(i))
\end{array}$$

The termination of the rewrite system can be proved for instance by using the multiset path ordering (see Definition 1.3.3 on page 10) induced by the following precedence on function symbols: $sum >_{\mathcal{F}} \oplus, r$ and $r >_{\mathcal{F}} \vee, \wedge, p, q$.

Another kind of adder, a carry-lookahead adder, is described with auxiliary functions g and h to compute the carry, in the rewrite system below.

$$\begin{array}{lcll} sort & & Nat,\ Bool & \\ g & : & Nat & \mapsto\ Bool \\ h & : & Nat & \mapsto\ Bool \end{array}$$

$$\begin{array}{rcl} \forall i : Nat, & & \\ sum(i) & \rightarrow & g(i) \oplus p(i) \oplus r(i) \\ g(i) & \rightarrow & p(i) \wedge q(i) \\ h(i) & \rightarrow & p(i) \vee q(i) \\ r(s(i)) & \rightarrow & g(i) \vee (h(i) \wedge r(i)) \end{array}$$

A proof of equivalence between these two adders is achieved by the fact that normalising the rewrite rules of the second system with the rules for boolean expressions (namely $(x \vee y \rightarrow (x \wedge y) \oplus x \oplus y)$ and $(x \oplus x \rightarrow 0)$), yields the rules of the first system.

$$\begin{array}{rcl} r(s(i)) & \longrightarrow & g(i) \vee (h(i) \wedge r(i)) \\ & \longrightarrow & (p(i) \wedge q(i)) \vee ((p(i) \vee q(i)) \wedge r(i)) \\ & \longrightarrow & (p(i) \wedge q(i)) \vee (p(i) \wedge r(i)) \vee (q(i)) \wedge r(i)). \\ sum(i) & \longrightarrow & g(i) \oplus h(i) \oplus r(i) \\ & \longrightarrow & (p(i) \wedge q(i)) \oplus (p(i) \vee q(i)) \oplus r(i) \\ & \longrightarrow & (p(i) \wedge q(i)) \oplus (p(i) \wedge q(i) \oplus p(i) \oplus q(i)) \oplus r(i) \\ & \longrightarrow & p(i) \oplus q(i) \oplus r(i). \end{array}$$

Orthogonal systems.

For non-terminating systems, confluence can be however ensured by adding strong syntactic conditions on the left-hand sides. This is the case for instance for the class of orthogonal systems.

Definition 1.3.5 *A non-variable term t'* overlaps *a term t if there exists a non-variable position ω in t such that $t_{|\omega}$ and t' are unifiable.*
A rewrite system is non-overlapping if there is no overlap between the rules left-hand sides.
A term is linear if each variable occurs only once. A rewrite system is left-linear if the left-hand side of each rule is a linear term. A rewrite system that is both left-linear and non-overlapping is called orthogonal.

Example 1.3.7 *The system from Combinatory Logic given in Example 1.3.1 on page 9 is orthogonal.*

Theorem 1.3.3 *If a rewrite system R is orthogonal, then it is confluent.*

This is proved in [Hue80], using the fact that concurrent reduction (i.e. application in one step of rewrite of rules of R at disjoint positions) is strongly confluent.

Orthogonal rewrite systems have been extensively studied in the context of operational semantics of recursive programming languages. Of particular interest is the existence of normalising reduction strategies. In [HL91a, HL91b], it is shown that in an orthogonal rewrite system, each term contains a needed redex, i.e. a redex that has to be rewritten in order to reach a normal form. A call-by-need strategy is obtained by repeatedly rewriting needed redexes, and always succeeds in finding a normal form when it exists. Although it is in general undecidable whether a redex is needed, for the class of strongly sequential systems, needed redexes can be found effectively. Moreover it is decidable whether an orthogonal rewrite system is strongly sequential.

Decidability results.

Most properties of rewrite systems are undecidable. For a rewrite system built on a finite signature with finitely many rewrite rules, it is undecidable whether confluence holds and whether termination holds. However for ground rewrite systems (i.e. with no variable), confluence is decidable [DHLT87] and termination is decidable [HL78]. But ground confluence is undecidable as shown in [KNO90].

A rewrite system R provides a decision procedure for an equational theory E if R is finite, $\longrightarrow_R$ is convergent and $\overset{*}{\longleftrightarrow}_R$ coincides with $=_E$. The word problem is the problem to decide whether an equality $s = t$ between two ground terms follows from E. This is a particular case of provability in E for arbitrary terms. If R is ground convergent, the word problem is decidable by reducing both terms s and t to their normal forms and by testing the syntactic equality of the results. Of course not every equational theory can be decided by rewriting. Some decidable theories are not finitely presented. Moreover some finitely presented theories are undecidable (the combinatorial logic for instance). Even some finitely presented and decidable theories are not decidable by rewriting.

Example 1.3.8 *Kapur and Narendran [KN85] found the following finite Thue system, made of the only axiom: $h(k(h(x))) = k(h(k(x)))$, with a decidable word problem and without equivalent finite convergent system.*

1.3.2 Rewriting modulo a set of equalities

A second kind of abstract reduction system is obtained by considering $\mathcal{T}$ as the set of equivalence classes of terms modulo the congruence generated by a set of axioms A, and $\longrightarrow$ as the rewriting relation induced on equivalence classes by a set of rewrite rules. This is especially important to deal with the problem of commutative or associative and commutative (AC for short) theories. This requires the elaboration of new abstract concepts, namely the notion of *class rewrite systems*, also called equational term rewriting systems. Let A be any set of axioms with decidable unification, matching and word problems and $\overset{*}{\longleftrightarrow}_A$ be the generated congruence relation on $\mathcal{T}(\Sigma, \mathcal{X})$.

Definition 1.3.6 *A* class rewrite system *denoted (R/A), is defined by a set of axioms A and a set of rewrite rules R assumed disjoint.*

The class rewrite relation applies to a term if there exists a term in the same equivalence class modulo A that is reducible with R.

Definition 1.3.7 *A term t* rewrites *to t'* with a class rewrite system (R/A), *denoted $t \longrightarrow_{R/A} t'$, if there exist a rewrite rule $(l \to r) \in R$, a term u, a position ω in u and a substitution σ such that $t \overset{*}{\longleftrightarrow}_A u[\sigma(l)]_\omega$, and $t' \overset{*}{\longleftrightarrow}_A u[\sigma(r)]_\omega$.*

A term irreducible for $\longrightarrow_{R/A}$ is said to be in (R/A)-normal form. An (R/A)-normal form of a term t is denoted $t \downarrow_{R/A}$. Let us consider more in detail the termination and confluence properties for this new relation.

A reduction ordering $>$ is A-compatible if $\overset{*}{\longleftrightarrow}_A \circ > \circ \overset{*}{\longleftrightarrow}_A \subseteq >$. Well-founded A-compatible reduction orderings do not exist when R is non-empty and A contains an axiom like idempotency $(x + x = x)$ where a lone variable occurs on one side and several times on the other. From an instance σ of a rewrite rule $(l \to r) \in R$, a contradiction to the well-foundedness of $>$ may be built, provided $\sigma(l) > \sigma(r)$:
$\sigma(l) \overset{*}{\longleftrightarrow}_A \sigma(l) + \sigma(l) > \sigma(r) + \sigma(l) \overset{*}{\longleftrightarrow}_A \sigma(r) + (\sigma(l) + \sigma(l)) > \sigma(r) + (\sigma(r) + \sigma(l)) \ldots$
Other axioms that prevent the existence of a well-founded A-compatible reduction ordering are equalities like $(x * 0 = 0)$ where a variable occurs on one side and not on the other. Then $0 \longleftrightarrow_A \sigma(l) * 0 > \sigma(r) * 0 \longleftrightarrow_A 0$ provides a contradiction to well-foundedness. Indeed, if such axioms are present, they must be considered as rewrite rules.

The rewrite relation $\longrightarrow_{R/A}$ is not completely satisfactory from an operational point of view: even if R is finite and $\overset{*}{\longleftrightarrow}_A$ decidable, $\longrightarrow_{R/A}$ may not be computable since equivalence classes modulo A may be infinite or not computable. For instance, the axiom $(-x = x)$ generates infinite equivalence classes. To avoid

searching through equivalence classes, the idea is to use a weaker relation on terms, called *rewriting modulo* A, which incorporates A in the matching process.

Definition 1.3.8 *Given a class rewrite system (R/A), the term t (R,A)-*rewrites *to t', denoted by $t \longrightarrow_{R,A}^{(l\to r),\omega,\sigma} t'$, if there exist a rewrite rule $(l \to r)$ of R, a position ω in t, a substitution σ, such that $t_{|\omega} \overset{*}{\longleftrightarrow}_A \sigma(l)$, and $t' = t[\sigma(r)]_\omega$.*

In order to get the same effect as rewriting in equivalence classes, some properties called *confluence and coherence modulo* a set of axioms are needed [Hue80, Jou83, JK86a].

Definition 1.3.9 *The class rewrite system (R/A) is* Church-Rosser *on a set of terms $\mathcal{T}$ if*

$$\overset{*}{\longleftrightarrow}_{R\cup A} \quad \subseteq \quad \overset{*}{\longrightarrow}_{R/A} \circ \overset{*}{\longleftrightarrow}_A \circ \overset{*}{\longleftarrow}_{R/A} .$$

The rewriting relation $\longrightarrow_{R,A}$ ((R,A) for short) defined on $\mathcal{T}$ is:

- Church-Rosser modulo A *if*
$\overset{*}{\longleftrightarrow}_{R\cup A} \quad \subseteq \quad \overset{*}{\longrightarrow}_{R,A} \circ \overset{*}{\longleftrightarrow}_A \circ \overset{*}{\longleftarrow}_{R,A}$

- confluent modulo A *if*
$\overset{*}{\longleftarrow}_{R,A} \circ \overset{*}{\longrightarrow}_{R,A} \quad \subseteq \quad \overset{*}{\longrightarrow}_{R,A} \circ \overset{*}{\longleftrightarrow}_A \circ \overset{*}{\longleftarrow}_{R,A}$

- coherent modulo A *if*
$\overset{*}{\longleftarrow}_{R,A} \circ \overset{*}{\longleftrightarrow}_A \quad \subseteq \quad \overset{*}{\longrightarrow}_{R,A} \circ \overset{*}{\longleftrightarrow}_A \circ \overset{*}{\longleftarrow}_{R,A}$

- locally confluent modulo A *if*
$\longleftarrow_{R,A} \circ \longrightarrow_R \quad \subseteq \quad \overset{*}{\longrightarrow}_{R,A} \circ \overset{*}{\longleftrightarrow}_A \circ \overset{*}{\longleftarrow}_{R,A}$

- locally coherent modulo A *if*
$\longleftarrow_{R,A} \circ \longleftrightarrow_A \quad \subseteq \quad \overset{*}{\longrightarrow}_{R,A} \circ \overset{*}{\longleftrightarrow}_A \circ \overset{*}{\longleftarrow}_{R,A}$

The next theorem relates the above properties.

Theorem 1.3.4 [JK86a] *The following properties of a class rewrite system (R/A), are equivalent on $\mathcal{T}$:*
- *(R,A) is Church-Rosser modulo A.*
- *(R,A) is confluent modulo A and coherent modulo A.*
- *(R,A) is locally confluent and locally coherent modulo A.*
- *$\forall t, t'$, $t \overset{*}{\longleftrightarrow}_{R\cup A} t'$ if and only if $t \downarrow_{R,A} \overset{*}{\longleftrightarrow}_A t' \downarrow_{R,A}$.*

For the specific case of associative and commutative theories, there exists a systematic way to fulfill local coherence by adding extended rules.

Definition 1.3.10 *Let R be a rewrite system and A be the axioms of commutativity and associativity of the symbol f. Any rule $l \to r \in R$ such that the top symbol of l is f, has an* extended rule $f(l,x) \to f(r,x)$ *where x is variable that does not occur in l (thus nor in r).*
Let R^{ext} be R plus all extended rules $f(l,x) \to f(r,x)$ of rules $l \to r \in R$ for which there exist no rule $l' \to r'$ in R and no substitution σ, such that $f(l,x) \overset{*}{\longleftrightarrow}_{AC} \sigma(l')$ and $f(r,x) \overset{*}{\longrightarrow}_{AC\cup R/AC} \sigma(r')$.

Indeed if there exist a rule $l' \to r'$ in R and a substitution σ, such that $f(l,x) \overset{*}{\longleftrightarrow}_{AC} \sigma(l')$ and $f(r,x) \overset{*}{\longrightarrow}_{AC\cup R/AC} \sigma(r')$, the extended rule is not needed for achieving local coherence.

Example 1.3.9 *This Church-Rosser axiomatisation of free distributive lattices is taken from [PS81]. Let $\cup$ and $\cap$ be two associative and commutative symbols:*

$$\begin{array}{rcl} x \cup y & = & y \cup x \\ (x \cup y) \cup z & = & x \cup (y \cup z) \end{array} \qquad \begin{array}{rcl} x \cap y & = & y \cap x \\ (x \cap y) \cap z & = & x \cap (y \cap z) \end{array}$$

The following set of rules R

$$\begin{array}{rcl} \forall x, y, z, & & \\ x \cup (x \cap y) & \to & x \\ x \cap (y \cup z) & \to & (x \cap y) \cup (x \cap z) \\ x \cap x & \to & x \\ x \cup x & \to & x \end{array}$$

is such that (R^{ext}, AC) is Church-Rosser modulo AC and R/AC is terminating.

The term $(b \cup (a \cup b))$ where a, b are constants, can be reduced at top by the extended rule $(x \cup x) \cup z \to x \cup z$ to $a \cup b$, since $(b \cup (a \cup b))$ is AC-equivalent to $(b \cup b) \cup a$.

Many mathematical structures (abelian groups, commutative rings for instance) involve associativity and commutativity laws.

Example 1.3.10 *Boolean ring*
Since the 1950s it is known that any term of the boolean algebra (i.e. any boolean formula) admits a set of prime implicants *that can be computed algorithmically [Qui59, SCL70]. But attempts to find a Church-Rosser class rewrite system for Boolean algebra using a completion procedure failed to terminate in all experiments reported by [Hul80b, PS81]. Only in 1991, a formal proof on the non-existence of a convergent system for boolean algebra was given in [Soc91]. Previously in 1985, J. Hsiang observed that there exists however a convergent class rewrite system in a signature $\mathcal{F} = \{0, 1, \wedge, \vee, \neg, \oplus\}$, where $\oplus$ and $\wedge$ are operators from a boolean ring. A class rewrite system for boolean rings, where the conjunction $\wedge$ and the exclusive-or $\oplus$ are associative and commutative, is given by the following sets of rewrite rules and equalities, denoted BR/AC:*

$$\begin{array}{rcl} x \oplus 0 & \to & x \\ x \oplus x & \to & 0 \\ x \wedge 0 & \to & 0 \\ x \wedge 1 & \to & x \\ x \wedge x & \to & x \\ x \wedge (y \oplus z) & \to & (x \wedge y) \oplus (x \wedge z) \end{array}$$

$$\begin{array}{rcl} x \wedge y & = & y \wedge x \\ (x \wedge y) \wedge z & = & x \wedge (y \wedge z) \end{array} \qquad \begin{array}{rcl} x \oplus y & = & y \oplus x \\ (x \oplus y) \oplus z & = & x \oplus (y \oplus z) \end{array}$$

Using this class rewrite system, we get the following derivation where p, q, r are predicate symbols:

$$(p(x) \wedge p(x)) \oplus p(x) \oplus (q(y) \wedge r(x, y) \wedge 1) \oplus (p(x) \wedge 0) \oplus 1 \xrightarrow{*}_{BR/AC} q(y) \wedge r(x, y) \oplus 1.$$

In order to deal with Boolean algebras negation $\neg$ and disjunction $\vee$ [HD83], one may add rules:

$$\begin{array}{rcl} x \vee y & \to & (x \wedge y) \oplus x \oplus y \\ \neg x & \to & x \oplus 1 \end{array}$$

A refinement of rewriting modulo A called normalised rewriting has been introduced in [Mar94]. Considering A as a Church-Rosser class rewrite system (S/AC), it consists in normalising modulo AC with S before applying a rule in R. It allows complex theories for A, such as abelian groups, commutative rings or finite fields.

1.3.3 Conditional rewriting

Conditional equalities arise naturally in algebraic specifications of data types, where they provide a way to handle a large class of partial functions and case analysis. The formulas being considered are written

"Γ if $l = r$", where Γ is a conjunction of equalities $(s_1 = t_1 \wedge \ldots \wedge s_n = t_n)$. Γ and $l = r$ are respectively called the *condition* and the *conclusion* of the conditional equality. In a *conditional rewrite rule*, the conclusion is oriented and this denoted as "$t \to s$ if Γ". The orientation is performed with respect to an ordering on terms. There may be variables occurring in the condition but not in the conclusion, called extra-variables. Different abstract reduction systems defined on terms but using different relations have been proposed for these formulas. First, using a conditional rewrite system R, a given rule may be applied to a term if its condition, instantiated by the matching substitution, is satisfied.

Definition 1.3.11 *Given a conditional rewrite system R, a term t* rewrites *to a term t' if there exist a conditional rewrite rule $l \to r$ if Γ of R, a position ω in t, a substitution σ, satisfying $t_{|\omega} = \sigma(l)$, a substitution τ for the extra-variables such that $\tau(\sigma(\Gamma))$ holds. Then $t' = t[\omega \hookleftarrow \tau(\sigma(r))]$.*

When conditional rewriting is used for modelling logic and functional programming, this definition which allows extra-variables in the conditions and right-hand sides of rules, is quite convenient. A conditional rewrite rule is applicable if there exists a substitution for the extra-variables that makes the condition hold.

Example 1.3.11 *Consider the set of conditional rewrite rules with variables x, y, z, x', y':*

$$\begin{array}{rcl} & & append(nil, y) \to y \\ (x = cons(x', y') \wedge append(y', y) = z) & \text{if} & append(x, y) \to cons(x', z) \end{array}$$

With the second rule, $append(cons(a, nil), nil)$ rewrites to $cons(a, nil)$ since the substitution $(x' \mapsto a)(y' \mapsto nil)(z \mapsto nil)$ is a solution of $(cons(a, nil) = cons(u, v) \wedge append(y', nil) = z)$. In this example, there are variables in the right-hand side of the second rule that do not occur in the left-hand side.

Definition 1.3.11 is too general to be efficient, due to the complexity of verifying the condition. This leads us to distinguish more restrictive notions of conditional rewriting relations, according to the kind of evaluation chosen for the conditions. Definition 1.3.11 is modified accordingly.

First it is often required that every variable occurring either in the condition or in the right-hand side of a rule also occurs in the left-hand side, which is denoted by $\mathcal{V}ar\,\Gamma \cup \mathcal{V}ar\,r \subseteq \mathcal{V}ar\,l$. This requirement rules out Example 1.3.11.

Thus in the condition (C) "there exists a substitution τ for the extra-variables such that $\tau(\sigma(\Gamma))$ holds" of Definition 1.3.11, τ is identity and we are left to check that $\sigma(\Gamma)$ holds.

This is done in different ways, according to different kinds of rewriting relations introduced below. Only the replacements of condition (C) in Definition 1.3.11 are mentioned:

1. for *natural* conditional rewriting: $t \longrightarrow_{R^{nat}} t'$ if
   (C^{nat}) there exists a proof $\sigma(s_i) \overset{*}{\longleftrightarrow}_{R^{nat}} \sigma(t_i)$ for each instantiated component of the condition.
2. for *join* conditional rewriting: $t \longrightarrow_{R^{join}} t'$ if
   (C^{join}) $\sigma(s_i) \downarrow_{R^{join}} = \sigma(t_i) \downarrow_{R^{join}}$ for each instantiated component of the condition.
3. for *normal* conditional rewriting: $t \longrightarrow_{R^{norm}} t'$ if
   (C^{norm}) $\sigma(t_i)$ is a normal form of $\sigma(s_i)$, denoted by $\sigma(s_i) \overset{!}{\longrightarrow}_{R^{norm}} \sigma(t_i)$, for each instantiated component of the condition.

(C^{nat}) is very close to equational reasoning with the underlying conditional equalities. But from a rewriting point of view it is not quite satisfactory due to the bidirectional use of rewrite rules. In practice, (C^{join}) has been more studied and is often implemented. (C^{norm}) is quite interesting in relation to logic and functional programming, and when conditions are expressed with boolean functions.

Example 1.3.12 [BK86] *Consider the join conditional rewriting relation for the system:*

$$\begin{array}{rcl} & & f(a) \to a \\ x = f(x) & \text{if} & f(x) \to c \end{array}$$

Since $a \downarrow = f(a) \downarrow$, $f(a) \longrightarrow c$ by applying the second rule. Then $f(f(a)) \longrightarrow c$. But neither c nor $f(c)$ are reducible.

Example 1.3.13 *Consider the normal conditional rewriting relation for the system:*

$$\begin{array}{rcll} & & even(0) & \to & true \\ & & even(s(x)) & \to & odd(x) \\ even(x) = false & \mathsf{if} & odd(x) & \to & true \\ even(x) = true & \mathsf{if} & odd(x) & \to & false \end{array}$$

Then $even(s(0)) \longrightarrow odd(0) \longrightarrow false$ *by applying first the second, then the fourth rule.*

In any case, the rewrite relation $\longrightarrow_R$ associated with a conditional rewrite system has an inductive definition, which is fundamental for establishing properties of conditional rewrite systems.

Definition 1.3.12 *Let* R *be a conditional rewrite system and* R_i *the rewrite system defined for* $i \geq 0$ *as follows:*

$$\begin{array}{rcl} R_0 & = & \{l \to r \mid l \to r \in R\} \\ R_{i+1} & = & R_i \cup \{\sigma(l) \to \sigma(r) \mid l \to r \ \mathsf{if}\ (s_1 = t_1 \wedge \ldots \wedge s_n = t_n) \in R, \\ & & \text{and } \forall j = 1, \ldots, n,\ \sigma(s_j) \equiv_i \sigma(t_j)\} \end{array}$$

where $\equiv_i$ *denotes* $\overset{*}{\longleftrightarrow}_{R_i}$, $\downarrow_{R_i}$ *or* $\overset{!}{\longrightarrow}_{R_i}$. *Then* $t \longrightarrow_R t'$ *if and only if* $t \longrightarrow_{R_i} t'$ *for some* $i \geq 0$.

With this definition, an abstract reduction system can be associated to any conditional rewrite system, according to the evaluation chosen for the conditions. All definitions and properties of abstract relations are thus available for these abstract reduction systems.

Obviously, if R^{join} is confluent, then so is the relation R^{nat}, but the converse does not hold, because not all proofs in the condition of a natural rewriting proof can be transformed into joinability proofs.

Example 1.3.14 [Kap84] *Consider the following system, where* a, b, a', b', a'' *are constants and* g, d *are any terms.*

$$\begin{array}{rcll} & a'' & \to & a \\ & a' & \to & b \\ & b' & \to & a \\ & b' & \to & b \\ a' = a'' \ \mathsf{if} & g & \to & d \end{array}$$

Then $a'' \longleftrightarrow_{R^{nat}} a \longleftrightarrow_{R^{nat}} b' \longleftrightarrow_{R^{nat}} b \longleftrightarrow_{R^{nat}} a'$ *and so* $g \longrightarrow_{R^{nat}} d$. *But* $a'' \downarrow_{R^{join}} a'$ *is false and thus* $g \longrightarrow_{R^{join}} d$ *is false, as well as* $g \overset{*}{\longleftrightarrow}_{R^{join}} d$.

In [Kap83, Kap84], an example is built of a conditional rewrite system such that the relation $\longrightarrow_R$ is not decidable and such that the normal form of a term is not computable. Additional conditions must be added to get back decidable properties.

Definition 1.3.13 *A conditional rewriting relation is* decreasing *if there exists a well-founded extension* $>$ *of the rewriting relation* $\longrightarrow$ *which satisfies two additional properties:*

- $>$ *contains the proper subterm relation.*
- *for each rule* $l \to r$ if $(s_1 = t_1 \wedge \ldots \wedge s_n = t_n)$, $\sigma(l) > \sigma(s_i)$ *and* $\sigma(l) > \sigma(t_i)$, *for all substitutions* σ *and all indices* i, $1 \leq i \leq n$.

The next result is proved by induction on the ordering that makes the conditional rewriting relation decreasing.

Theorem 1.3.5 [DO90] *Whenever* $\longrightarrow_{R^{join}}$ *is decreasing, the relations* $\longrightarrow_{R^{join}}$, $\overset{*}{\longrightarrow}_{R^{join}}$ *and* $\downarrow_{R^{join}}$ *are all decidable.*

The notion of decreasingness of a rewriting relation is not usable in practice. A property that can be checked on the conditional rewrite system is preferable. This leads to introduce the notion of *reductive* system [JW86] that generalises the definition of *simplifying* system in [Kap87].

Definition 1.3.14 [JW86] *A conditional rewrite system is* reductive *if there exists a well-founded reduction ordering* $>$ *such that for each rule* $l \to r$ if $(s_1 = t_1 \wedge \ldots \wedge s_n = t_n)$, *and substitution* σ
(i) $\sigma(l) > \sigma(r)$ *and*
(ii) $\sigma(l) > \sigma(s_i)$ *and* $\sigma(l) > \sigma(t_i)$, *for all indices* i, $1 \leq i \leq n$.

Actually reductive systems capture the finiteness of evaluation of terms, as explained in [DO90] where it is also proved that if R is a reductive conditional rewrite system, R^{join} is decreasing.

1.4 Rewriting as a logic for concurrent computations

While in the previous section, the emphasis was on the Church-Rosser property and on the use of rewriting as a decision procedure for equality in equational theories, another point of view is adopted now which strengthens the computational logic aspect of rewriting. Rewriting logic is presented for instance in [Mes92] as a logic of actions in which logical deduction is identified with concurrent rewriting, i.e. application in one step of rewrite rules at disjoint positions in a term. Rewriting logic provides a general framework for unifying a variety of models of concurrency as argued in [Mes92] but also a logical framework in which other logics can be encoded as proposed in [MM93].

In rewriting logic, the sentences are of the form $t \longrightarrow t'$ where t and t' are terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. More generally, some structural axioms may be taken into account and sequents are then defined as sentences of the form $\langle t \rangle_A \longrightarrow \langle t' \rangle_A$ where $\langle t \rangle_A$ denotes the equivalence class of t modulo A. Then the considered rewrite relation is rewriting modulo A. To support intuition, the reader can think of A as being the empty set of axioms, or the associativity commutativity axioms for some binary symbol. But in theory, A can encode more complex data structures.

Moreover in rewriting logic, proofs are first-order objects and are represented by proof terms. In order to build proof terms and later on strategies, rules are labelled with a set $\mathcal{L}$ of ranked label symbols, i.e. of symbols with a fixed arity. In order to compose proofs, the infix binary operator ";" is introduced. A *proof term* is by definition a term built on function symbols in $\mathcal{F}$, label symbols in $\mathcal{L}$ and the concatenation operator ";". Let $\mathcal{PT}$ denote the set of proof terms. Taking proof terms into account in the sentences leads to consider sequents of the form $\pi : \langle t \rangle_A \longrightarrow \langle t' \rangle_A$ where $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $\pi \in \mathcal{PT}$. The informal meaning of such sentences is that the proof π allows to derive t' from t. The labels are crucial both in the later definition of strategies and in the construction of models for the rewriting logic.

We are now ready to formalise deduction in rewriting logic. We restrict here for simplicity to unconditional rewriting, but everything can be extended to conditional rewriting at the cost of more technical definitions. The reader is referred to [Mes92] for a full treatment of this case.

A *labelled rewrite theory* is a 5-tuple $\mathcal{R} = (\mathcal{X}, \mathcal{F}, A, \mathcal{L}, R)$ where $\mathcal{X}$ is a countably infinite set of variables, $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols, A a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities, and R a set of labelled rewrite rules of the form $\ell : l \to r$ where $\ell \in \mathcal{L}$ and $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Each rule $\ell : l \to r$ has a finite set of variables $\mathcal{V}ar\, l \cup \mathcal{V}ar\, r = \{x_1, \ldots, x_n\}$, and the arity of ℓ is exactly the number of distinct variables in this set. This is recorded in the notation $\ell(x_1, \ldots, x_n) : l \to r$.

A labelled rewrite theory $\mathcal{R}$ entails the sequent $\pi : \langle t \rangle_A \longrightarrow \langle t' \rangle_A$, if the sequent is obtained by the finite application of the deduction rules REW presented in Figure 1.1 on the next page.

Example 1.4.1 *Consider as an example the following specification:*

$$
\begin{array}{lcll}
sorts & & Nat & \\
0 & : & & \mapsto \; Nat \\
s & : & Nat & \mapsto \; Nat \\
+ & : & Nat \times Nat & \mapsto \; Nat
\end{array}
$$

Reflexivity For any $t \in \mathcal{T}(\mathcal{F})$:

$$\frac{}{t : \langle t \rangle_A \longrightarrow \langle t \rangle_A}$$

Congruence For any $f \in \mathcal{F}$ with $arity(f) = n$:

$$\frac{\pi_1 : \langle t_1 \rangle_A \longrightarrow \langle t'_1 \rangle_A \quad \ldots \quad \pi_n : \langle t_n \rangle_A \longrightarrow \langle t'_n \rangle_A}{f(\pi_1, \ldots, \pi_n) : \langle f(t_1, \ldots, t_n) \rangle_A \longrightarrow \langle f(t'_1, \ldots, t'_n) \rangle_A}$$

Replacement For any rewrite rule $\ell(x_1, \ldots, x_n) : l \to r$ of R,

$$\frac{\pi_1 : \langle t_1 \rangle_A \longrightarrow \langle t'_1 \rangle_A \quad \ldots \quad \pi_n : \langle t_n \rangle_A \longrightarrow \langle t'_n \rangle_A}{\ell(\pi_1, \ldots, \pi_n) : \langle l(t_1, \ldots, t_n) \rangle_A \longrightarrow \langle r(t'_1, \ldots, t'_n) \rangle_A}$$

Transitivity

$$\frac{\pi_1 : \langle t_1 \rangle_A \longrightarrow \langle t_2 \rangle_A \qquad \pi_2 : \langle t_2 \rangle_A \longrightarrow \langle t_3 \rangle_A}{\pi_1 ; \pi_2 : \langle t_1 \rangle_A \longrightarrow \langle t_3 \rangle_A}$$

Figure 1.1: REW: The rules of rewrite deduction

A contains the commutativity axiom for $+$:

$$\forall x, y : Nat, \quad x + y = y + x$$

with the following set of labelled rewrite rules R:

$$\begin{array}{lcll} \ell_0 & : & 0 + 0 & \to & 0 \\ \ell_1(x, y) & : & s(x) + y & \to & s(x + y) \end{array}$$

In this rewrite theory, we can prove:

$$\langle 0 + s(0) \rangle_A \longrightarrow \langle s(0) \rangle_A$$

We first use Reflexivity to get

$$0 : \langle 0 \rangle_A \longrightarrow \langle 0 \rangle_A$$

With Replacement using the rewrite rule ℓ_1 and commutativity of $+$:

$$\frac{0 : \langle 0 \rangle_A \longrightarrow \langle 0 \rangle_A \qquad 0 : \langle 0 \rangle_A \longrightarrow \langle 0 \rangle_A}{\ell_1(0, 0) : \langle 0 + s(0) \rangle_A \longrightarrow \langle s(0 + 0) \rangle_A}$$

With Replacement using the rewrite rule ℓ_0

$$\ell_0 : \langle 0 + 0 \rangle_A \longrightarrow \langle 0 \rangle_A$$

With Congruence for the symbol s:

$$\frac{\ell_0 : \langle 0 + 0 \rangle_A \longrightarrow \langle 0 \rangle_A}{s(\ell_0) : \langle s(0 + 0) \rangle_A \longrightarrow \langle s(0) \rangle_A}$$

and with Transitivity

$$\frac{\ell_1(0, 0) : \langle 0 + s(0) \rangle_A \longrightarrow \langle s(0 + 0) \rangle_A \qquad s(\ell_0) : \langle s(0 + 0) \rangle_A \longrightarrow \langle s(0) \rangle_A}{\ell_1(0, 0) ; s(\ell_0) : \langle 0 + s(0) \rangle_A \longrightarrow \langle s(0) \rangle_A}$$

This concludes the proof and builds the associated proof term.

Although the rules of rewriting logic are very close to those of equational logic, the gap between equational logic and rewriting logic relies actually upon the requirement for symmetry. Let us now make the link with the more operational notion of rewriting previously introduced in Section 1.3. For simplicity, let us consider an empty set of structural axioms A and forget proof terms. So sequents are simply written $t \longrightarrow t'$. Concurrent rewriting coincides exactly with deduction in rewriting logic and sequential rewriting appears as a special case.

For a given rewrite system R, a sequent $t \longrightarrow t'$ is:

- a *zero-step R-rewrite*, if it can be derived from R by application of the rules $Reflexivity$ and $Congruence$. In this case t and t' coincide.
- a *one-step concurrent R-rewrite*, if it can be derived from R by application of the rules $Reflexivity$, $Congruence$ and at least one application of $Replacement$. When $Replacement$ is applied exactly once, then the sequent is called a one-step *sequential R-rewriting*.
- a *concurrent R-rewrite* if it can be derived from R by finite application of the rules in REW.

The relation between one-step sequential R-rewriting and the rewrite relation $\longrightarrow_R$ defined on terms in Section 1.3.1 is made precise in the following lemma, proved by induction on the form of the proof of the sequent $t \longrightarrow t'$.

Lemma 1.4.1 *A sequent $t \longrightarrow t'$ is a one-step sequential R-rewriting if and only if there exist a rule $l \to r$ in R, a substitution σ and a position ω of t such that $t \longrightarrow_R^{\omega,\sigma,l\to r} t'$.*

This result extends to any concurrent rewriting derivation and is also a consequence of the form of the proof of the sequent.

Lemma 1.4.2 *For each sequent $t \longrightarrow t'$ computed using the rewriting logic relative to a set of rules R:*

- *either $t = t'$,*
- *or there exists a chain of one-step concurrent R-rewrite:*

$$t = t_0 \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_{n-1} \longrightarrow t_n = t'.$$

Moreover, in this chain, each step $t_i \longrightarrow t_{i+1}$ can be chosen sequential.

This last result shows the equivalence between the operational definition of rewriting and the sequential rewriting relation obtained from rewriting logic. The main advantage in rewriting logic is precisely to get rid of an operational definition of rewriting that needs to handle the notion of positions, matching and replacement.

Interesting investigations for rewriting logic are its use for specification and design of software systems, as well as for semantic integration of different programming paradigms based on a common logic and model theory. In [MM93], it is shown how various logics can be represented in rewriting logic, as for example equational logic, Horn logic, linear logic and the natural semantics [Kah87]. In [KKV95, BKK+96b], rewriting logic is experimented to design constraint solvers, theorem provers, kernel of programming languages, but also to combine computational paradigms in a single logical framework. This is applied for instance to the combination of two constraint solvers, of constraint solving and theorem proving, and of constraint solving with logic programming.

However this kind of applications reveals the need for the definition of strategies. In practice, a strategy is a way to describe which computations the user is interested in, and specifies where a given rule should be applied in the term to be reduced. From a theoretical point of view, a strategy as defined in [KKV95, BKK+96b] characterises a subset of all proof terms that can be built in the current rewrite theory. The application of a strategy to a term results in the (possibly empty) set of all terms derivable from the starting term, using this strategy. A strategy fails when it returns an empty set of terms. To illustrate these ideas, the following example illustrates how strategies are defined and implemented in the language ELAN.

Example 1.4.2 *The application of a rewrite rule in* ELAN *yields, in general, several results. This is first due to equational matching (for instance AC-matching computing several possible substitutions) and second, to the use of strategies that recursively return several possible results on subterms. Thus the language provides a way to handle this non-determinism. This is done using the basic strategy operators:* dont care choose *and* dont know choose. *For a rewrite rule* $\ell : l \to r$ *the strategy* dont care choose(ℓ) *returns* at most one *result which is undeterministically taken among the possible results of the application of the rule. In practice, the current implementation returns the first found one. On the contrary, if the* ℓ *rule is applied using the* dont know choose(ℓ) *strategy, then* all *possible results are computed and returned by the strategy. The implementation handles these several results by an appropriate back-chaining operation. This is extended to the application of several rules: the* dont know choose *strategy results in the application of all sub-strategies and yields the union of all results; the application of the* dont care choose *strategy returns the set of results of the first non-failing strategy. If all sub-strategies fail, then it fails too, i.e. it yields the empty set. Two strategies can be concatenated: this means that the second strategy is applied on all results of the first one. In order to allow the automatic concatenation of the same strategy,* ELAN *offers two iterators* iterate *and* repeat. *The strategy* iterate *corresponds to applying zero, then one, then two, ... n times the strategy to the starting term, until the strategy fails. Thus* $(iterate(s))t$ *returns* $\bigcup_{n=0}^{\infty}(s^n)t$. *Notice that* iterate *returns the results one by one even when an infinite derivation exists. The strategy* repeat *applies the strategy until it fails and returns just the terms resulting of the last unfailing call. It can be defined as* $(repeat(s))t = (s^n)t$ *where* $(s^{n+1})t$ *fails.*

A short illustration of the use of these strategies is provided by the specification of extraction of elements from a list.

$$\begin{array}{rcll} sort & & Elem, NeList & \\ \_ & : & Elem & \mapsto\ NeList \\ \cdot & : & Elem, NeList & \mapsto\ NeList \\ element & : & NeList & \mapsto\ Elem \end{array}$$

The declaration $\_ : Elem \mapsto NeList$ *of the invisible unary operator coerces an element of sort* $Elem$ *to be a non-empty list of sort* $NeList$. *Extraction is defined by three labelled rules:*

$$\begin{array}{l} \forall e : Elem, l : NeList, \\ \qquad \begin{array}{rcll} extract1(e) & : & element(e) & \to\ e \\ extract2(e,l) & : & element(e \cdot l) & \to\ e \\ extract3(e,l) & : & element(e \cdot l) & \to\ element(l) \end{array} \end{array}$$

Let a, b, c *be three constants of sort* $Elem$. *Then*

$$\text{repeat dont know choose}(extract1, extract2, extract3)(a \cdot b \cdot c)$$

yields the set $\{a, b, c\}$.

$$\text{repeat dont care choose}(extract1, extract2, extract3)(a \cdot b \cdot c)$$

yields the set $\{a\}$.

$$\text{iterate dont know choose}(extract1, extract2, extract3)(a \cdot b \cdot c)$$

yields the set $\{element(a \cdot b \cdot c), a, element(b \cdot c), b, element(c), c\}$.

In order to further improve this first language of elementary strategies, a natural idea is to provide the user with the capability to program his own strategies or tactics, as already done in many logical frameworks. The idea of controlling rewriting by rewriting, more precisely by a strategy language based on rewriting is investigated in [CELM96, GSHH92, BKK96a]. Roughly speaking, there are two levels of rewriting: the object (or first-order term) level, and the meta-level that controls the object-level.

Using conditional rewriting as a computational paradigm is simple, elegant, reasonably expressive and efficient, as witnessed for instance by performance of the ELAN compiler [Vit96]. Another advantage is that several properties of rewrite programs can be automatically proved. This is the topic of the second part of this chapter, where we focus on techniques for verifying some properties like confluence or well-typedness, and for proving some inductive properties of programs.

1.5 Superposition and simplification

The concept of critical pairs and the superposition process between two formulas introduced in this section appear in many places in automated deduction, and other examples of their use will be given in the following sections. Their role in the local confluence property is explained in the first part of this section presenting the basic completion procedure of rewrite systems. Completion is based on superposition and simplication by rewriting. In the second part, extensions and applications of these two basic mechanisms are surveyed.

1.5.1 Completion procedure

Results presented in Section 1.2 have stated that the Church-Rosser property of a terminating term rewriting system is equivalent to the local confluence property. This property is important in the context of programming in rewriting logic, for ensuring the unicity of the result computed by a rewrite program. Local confluence in turn can be checked on special patterns computed from pairs of rewrite rules and called *critical pairs*. When a set of rewrite rules fails this test because some critical pair is not joinable, a special rule tailored for this case is added to the system. This is the basic idea of the so-called *completion procedure*, designed by Knuth and Bendix [KB70], building on ideas of Evans [Eva51]. Of course, adding a new rule implies computing new critical pairs and the process is recursively applied. If this process of completion stops, it comes up with a locally confluent term rewriting system and if, in addition, termination has been incrementally checked, the system is also confluent. Two other possibilities can arise: the process can find an equality that cannot be oriented and then terminates in failure. It can also indefinitely discover non joinable critical pairs and hence endlessly add new rewrite rules. Critical pairs are produced by overlaps of two redexes in a same term.

Definition 1.5.1 [Hue80] *Let $(g \to d)$ and $(l \to r)$ be two rules with disjoint sets of variables, such that l overlaps g at a non-variable position ω of g with the most general unifier σ. The* overlapped term *$\sigma(g)$ produces the* critical pair *(p, q) defined by $p = \sigma(g[\omega \hookleftarrow r])$ and $q = \sigma(d)$. The new equality $(p = q)$ is obtained by* superposition *of $(l \to r)$ on $(g \to d)$ at position ω.*
$CP(R)$ will denote the set of all critical pairs between rules in R.

In this definition $(g \to d)$ and $(l \to r)$ may be the same rule up to variable renaming, that is a rewrite rule can overlap itself. Note that a rule always overlaps itself at the outermost position ϵ, producing a trivial critical pair.

Lemma 1.5.1 Critical pairs lemma [KB70, Hue80]
Let t, t', t'' be terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that

$$t' \leftarrow_R^{\omega,\alpha,l\to r} t \longrightarrow_R^{\epsilon,\beta,g\to d} t''$$

with ω being a non-variable position in g. Such a proof is called a peak. *Then, there exists a critical pair*

$$(p, q) = (\sigma(g[\omega \hookleftarrow r]), \sigma(d))$$

of the rule $(l \to r)$ on the rule $(g \to d)$ at position ω such that σ is the most general unifier of l and $g_{|\omega}$, $\beta\alpha(x) = \tau\sigma(x)$ for all $x \in \mathcal{V}ar\, g \cup \mathcal{V}ar\, l$ and for some substitution τ. So $t' = \tau(p)$ and $t'' = \tau(q)$.

Definition 1.5.2 *A critical pair (p, q) is said to be* joinable *(or sometimes convergent), which is denoted by $p \downarrow_R q$, if there exists a rewrite proof $p \xrightarrow{*}_R w \xleftarrow{*}_R q$.*

The next theorem is also called Newman's lemma in the literature.

Theorem 1.5.1 [KB70, Hue80] *A rewrite system R is locally confluent if and only if any critical pair of R is joinable.*

Since finite systems have a finite number of critical pairs, their local confluence is decidable. Moreover, according to Theorem 1.5.1 on the preceding page, a terminating rewrite system R is confluent if and only if all its critical pairs are joinable. This was the idea of the *superposition test* proposed by Knuth and Bendix [KB70]. The completion process was first studied from an operational point of view and several pioneer implementations were surveyed in [HKK89]. A completion process is composed of elementary tasks, such as rewriting a term in a rule or an equality, orienting an equality, computing a critical pair. The efficiency of the general process is related to the strategies chosen for combining these tasks. Each task can be formalised as a transition rule (a rewrite rule in some adequate rewrite theory), that transforms sets of equalities and rules. Strategies express the control for application of these transition rules. Separating control from transition rules makes the proofs of correctness and completeness independent from strategies. As in traditional proof theory, these transition rules are studied by looking at their effect on equational proofs. The completion process then appears as a way to modify the axioms in order to reduce a proof in some normal form, called a rewrite proof. This abstract point of view is adopted here. It is based on the relation between the transition rule system that describes the relation between successive pairs of sets R and P computed by the process on one hand, and a proof transformation that reduces proofs to rewrite proofs. Following [BD87] the completion process is described by a set of transition rules, which actually provides a rewrite theory for completion. Each transition rule transforms a computation state composed of pairs (P, R) where P is a set of equalities and R a set of rewrite rules. These rules are usually followed by a condition that specifies in which case the rule applies. Each step computes (P_{i+1}, R_{i+1}) from (P_i, R_i) using a rewrite relation $\mapsto\!\!\!\!\!\rightarrow$.
Let $R_* = \bigcup_{i\geq 0} R_i$ be the set of all generated rules and $P_* = \bigcup_{i\geq 0} P_i$ the set of all generated equalities.
Let R_∞ and P_∞ be respectively the set of persisting rules and pairs, i.e. the sets effectively generated by completion starting from a set of axioms P_0. Formally:

$$P_\infty = \bigcup_{i\geq 0} \bigcap_{j>i} P_j \quad \text{and} \quad R_\infty = \bigcup_{i\geq 0} \bigcap_{j>i} R_j.$$

Completion is parameterised by a given reduction ordering, denoted $>$, used to order equalities in a decreasing way. Moreover, rewrite rules are compared by the following ordering: $l \to r \gg g \to d$ if

- either a subterm of l is an instance of g and not conversely,
- or l and g are equal up to variable renaming, and $r > d$ in the given reduction ordering.

The completion procedure is expressed by the set of rules presented in Figure 1.2.

$$\begin{array}{lllll}
Orient & P \cup \{p = q\}, R & \mapsto\!\!\!\!\!\rightarrow & P, R \cup \{p \to q\} & \\
 & & & \text{if } p > q & \\
Deduce & P, R & \mapsto\!\!\!\!\!\rightarrow & P \cup \{p = q\}, R & \\
 & & & \text{if } (p, q) \in CP(R) & \\
Simplify & P \cup \{p = q\}, R & \mapsto\!\!\!\!\!\rightarrow & P \cup \{p' = q\}, R & \\
 & & & \text{if } p \longrightarrow_R p' & \\
Delete & P \cup \{p = p\}, R & \mapsto\!\!\!\!\!\rightarrow & P, R & \\
Compose & P, R \cup \{l \to r\} & \mapsto\!\!\!\!\!\rightarrow & P, R \cup \{l \to r'\} & \\
 & & & \text{if } r \longrightarrow_R r' & \\
Collapse & P, R \cup \{l \to r\} & \mapsto\!\!\!\!\!\rightarrow & P \cup \{l' = r\}, R & \\
 & & & \text{if } l \longrightarrow_R^{g\to d} l' \text{ and } l \to r \gg g \to d &
\end{array}$$

Figure 1.2: Standard completion rules

Orient turns an equality $p = q$ such that $p > q$ into a rewrite rule. *Deduce* adds equational consequences derived from overlaps between rules. *Simplify* uses rules to simplify both sides of equalities and could be written more precisely as two rules $Left - Simplify$ and $Right - Simplify$ where $q \longrightarrow_R q'$. *Delete* removes any trivial equality. *Compose* simplifies right-hand sides of rules with respect to other rules. *Collapse* simplifies the left-hand side of a rule and turns the result into a new equality, but only when the simplifying rule $g \to d$ is smaller than the disappearing rule $l \to r$ in the ordering $\gg$. Note that if any equality is

simplified by existing rules before being oriented, the condition of the *Collapse* rule is always satisfied. *Simplify*, *Compose* and *Collapse* are three aspects of *simplification* of formulas by rewriting with current rules in R.

These rules are sound in the following sense: they do not change the equational theory.

Lemma 1.5.2 [Bac91] *If* $(P, R) \mapsto\!\!\!\!\twoheadrightarrow (P', R')$ *then* $\overset{*}{\longleftrightarrow}_{P \cup R}$ *and* $\overset{*}{\longleftrightarrow}_{P' \cup R'}$ *are the same.*

A completion procedure is aimed at transforming an initial set of equalities P_0 and an initial set of rules R_0 most often taken empty, into a rewrite rule system R_∞ that is convergent and inter-reduced.

With a proof-theoretical point of view, let us now consider the set of all provable equalities $(t = t')$. To each (P_i, R_i) is associated a proof of $(t = t')$ using rules in R_i and equalities in P_i. Following [Bac91], to each rule is associated a proof transformation rule on some kind of proofs using rules in R_* and equalities in P_*. The proof transformation relation is denoted by $\Longrightarrow$. Establishing the correctness of a completion procedure involves the following steps:

- Each sequence of proof transformation using the relation $\Longrightarrow$ terminates and decreases the complexity measure of proofs. For the standard completion rules given in Figure 1.2 on the facing page, the complexity measure is defined as follows in [Bac91]. The complexity measure of elementary proof steps by:

$$\begin{array}{lcl} c(s \longleftrightarrow_P t) & = & (\{s, t\}) \\ c(s \overset{l \to r}{\longleftrightarrow}_R t) & = & (\{s\}, l \to r) \end{array}$$

  By convention, the complexity of the empty proof Λ is $c(\Lambda) = (\emptyset)$. Complexities of elementary proof steps are compared using the lexicographic combination, denoted $>_{ec}$ of the multiset extension $>^{mult}$ of the reduction ordering for the first component, and the ordering $\gg$ for the second component. Since both $>$ and $\gg$ are well-founded, so is $>_{ec}$. The complexity $c(\mathcal{P})$ of a non-elementary proof $\mathcal{P}$ is the multiset of the complexities of its elementary proof steps. Complexities of non-elementary proofs are compared using the multiset extension $>_c$ of $>_{ec}$, which is also well-founded. Then if $\mathcal{P} \Longrightarrow \mathcal{P}'$, one can check that $c(\mathcal{P}) >_c c(\mathcal{P}')$.

- Every minimal proof is in the desired normal form, in this case a rewrite proof. For that, a completion procedure must satisfy a fairness requirement, which states that all necessary proof transformations are performed.

The fairness hypothesis states that any proof reducible by $\Longrightarrow$ will eventually be reduced. In other words, no reducible proof is forgotten.

Definition 1.5.3 *A derivation* $(P_0, R_0) \mapsto\!\!\!\!\twoheadrightarrow (P_1, R_1) \mapsto\!\!\!\!\twoheadrightarrow \ldots$ *is* fair *if whenever* $\mathcal{P}$ *is a proof in* $(P_i \cup R_i)$ *reducible by* $\Longrightarrow$*, then there is a proof* $\mathcal{P}'$ *in* $(P_j \cup R_j)$ *at some step* $j \geq i$ *such that* $\mathcal{P} \overset{+}{\Longrightarrow} \mathcal{P}'$.

A sufficient condition to satisfy the fairness hypothesis can be given:

Proposition 1.5.1 *A derivation* $(P_0, R_0) \mapsto\!\!\!\!\twoheadrightarrow (P_1, R_1) \mapsto\!\!\!\!\twoheadrightarrow \ldots$ *is fair if* $CP(R_\infty)$ *is a subset of* P_**,* R_∞ *is reduced and* P_∞ *is empty.*

In practice, the implementation of this abstract fairness condition is often performed through a marking process of rewrite rules whose critical pairs have been computed with all other rules, or which have been normalised. When fairness is ensured, proofs have normal forms with respect to $\Longrightarrow$. One can prove by induction on $\Longrightarrow$ that if a derivation $(P_0, R_0) \mapsto\!\!\!\!\twoheadrightarrow (P_1, R_1) \mapsto\!\!\!\!\twoheadrightarrow \ldots$ is fair, then any proof $t \overset{*}{\longleftrightarrow}_{P_i \cup R_i} t'$ for $i \geq 0$, has a rewrite proof $t \overset{*}{\longrightarrow}_{R_\infty} s \overset{*}{\longleftarrow}_{R_\infty} t'$.

Theorem 1.5.2 *If the derivation* $(P_0, R_0) \mapsto\!\!\!\!\twoheadrightarrow (P_1, R_1) \mapsto\!\!\!\!\twoheadrightarrow \ldots$ *is fair, then* R_∞ *is Church-Rosser and terminating. Moreover* $\overset{*}{\longleftrightarrow}_{P_0 \cup R_0}$ *and* $\overset{*}{\longleftrightarrow}_{R_\infty}$ *coincide on terms.*

The termination property of R_∞ is obvious since the test is incrementally processed for each rule added in R_*, thus in R_∞. The Church-Rosser property results from the fact that any proof has a normal form for $\Longrightarrow$ which is a rewrite proof. Eventually, the fact that $\overset{*}{\longleftrightarrow}_{P_0 \cup R_0} = \overset{*}{\longleftrightarrow}_{R_\infty}$ is a consequence of Lemma 1.5.2 on the page before.

The completion may not terminate and generate an infinite set of rewrite rules.

Example 1.5.1 [Der89] *The theory of idempotent semi-groups (sometimes called bands) is defined by the following set of two axioms:*

$$\begin{aligned} (x * y) * z &= x * (y * z) \\ x * x &= x \end{aligned}$$

The completion diverges and generates an infinite set of rewrite rules:

$$\begin{aligned} (x * y) * z &\to x * (y * z) \\ x * x &\to x \\ x * (x * z) &\to x * z \\ x * (y * (x * y)) &\to x * y \\ x * (y * (x * (y * z))) &\to x * (y * z) \\ &\dots \\ x * (y * (z * (y * (x * (y * (z * x)))))) &\to x * (y * (z * x)) \\ &\dots \end{aligned}$$

Example 1.5.2 *Rewriting and completion techniques have been applied to the study of several calculi defined for handling explicit substitutions in λ-calculus. The λ-calculus was defined by Church in the 30's in order to develop a general theory for computable functions. The λ-calculus contains a rule β, called β-reduction, which consists in replacing some variable occurrences by a term. This substitution is described in the meta-language of λ-calculus and introduces the need for variable renaming operations to take into account the binding rules environment. This was the motivation for the notation introduced by de Bruijn in the 70's for counting the binding depth of a variable. Another improvement has been introduced in the 85's (although the idea was already proposed by de Bruijn in 78) to deal with substitutions: instead of defining them in the meta-language, a new operator is introduced to explicitely denote the substitution to perform. So in addition to a rule (Beta) which starts the substitution, other rules are added to propagate it down to variables. Based on this idea, several calculi, regrouped under the name $\lambda\sigma$-calculi, have been studied and progressively refined in order to obtain suitable properties like termination and confluence.*

The $\lambda\sigma$-calculi of [ACCL91, CHL96] are first-order rewriting systems. They contain the λ-calculus, written in de Bruijn notation, as a proper subsystem, and they differ by their treatment of substitution, which leads to slightly different confluence properties. The chosen example here is the $\lambda\sigma$-calculus of [ACCL91].

A substitution is represented as a list of terms with an operator cons (written $\cdot$) and an operator for the empty list (written id as it represents the identity substitution). The substitution $u_1 \cdot u_2 \cdot \ldots \cdot u_n \cdot id$ replaces $\mathtt{1}$ by u_1, ..., $\mathtt{n}$ by u_n and decrements by n all the other (free) indices in the term. The operator $\uparrow$ can be seen as the infinite substitution $\mathtt{2} \cdot \mathtt{3} \cdot \mathtt{4} \cdot \ldots$. The term $u[s]$ denotes the application of the substitution s to u. The composition operator is denoted $\circ$. For more details see [ACCL91, CHL96].

Let $\mathcal{X}$ be a set of term metavariables, and $\mathcal{Y}$ be a set of substitution metavariables. The set of terms *and of* explicit substitutions *is inductively defined as:*

$$u = \mathtt{1} \mid X \mid (u\ u) \mid \lambda u \mid u[s]$$

$$s = Y \mid id \mid \uparrow \mid u \cdot s \mid s \circ s$$

with $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$.

This is a first-order many-sorted algebra built on the signature:

$$
\begin{array}{lll}
\mathtt{1} & : & \mapsto \mathtt{term} \\
(\_\ \_) & : \mathtt{term}\ \mathtt{term} & \mapsto \mathtt{term} \\
(\lambda\_) & : \mathtt{term} & \mapsto \mathtt{term} \\
\_[\_] & : \mathtt{term}\ \mathtt{substitution} & \mapsto \mathtt{term} \\
id & : & \mapsto \mathtt{substitution} \\
\uparrow & : & \mapsto \mathtt{substitution} \\
\_\cdot\_ & : \mathtt{term}\ \mathtt{substitution} & \mapsto \mathtt{substitution} \\
\_\circ\_ & : \mathtt{substitution}\ \mathtt{substitution} & \mapsto \mathtt{substitution}
\end{array}
$$

The $\lambda\sigma$-calculus is defined as the term rewriting system defined in Figure 1.3. If we drop the rule $(Beta)$, we get the rewrite system that performs application of substitutions.

$$
\begin{array}{lrcl}
(Beta) & (\lambda u)v & \to & u[v \cdot id] \\
(App) & (u\ v)[s] & \to & (u[s]\ v[s]) \\
(VarCons) & \mathtt{1}[u \cdot s] & \to & u \\
(Id) & u[id] & \to & u \\
(Abs) & (\lambda u)[s] & \to & \lambda(u[\mathtt{1} \cdot (s \circ \uparrow)]) \\
(Clos) & (u[s])[t] & \to & u[s \circ t] \\
(IdL) & id \circ s & \to & s \\
(ShiftCons) & \uparrow \circ (u \cdot s) & \to & s \\
(AssEnv) & (s_1 \circ s_2) \circ s_3 & \to & s_1 \circ (s_2 \circ s_3) \\
(MapEnv) & (u \cdot s) \circ t & \to & u[t] \cdot (s \circ t) \\
(IdR) & s \circ id & \to & s \\
(VarShift) & \mathtt{1} \cdot \uparrow & \to & id \\
(SCons) & \mathtt{1}[s] \cdot (\uparrow \circ s) & \to & s
\end{array}
$$

Figure 1.3: A $\lambda\sigma$ term rewriting system

The main properties of this $\lambda\sigma$ term rewriting system are:

1. *The term rewriting system $\lambda\sigma$ is locally confluent on any term [ACCL91].*
2. *$\lambda\sigma$ is confluent on terms without substitution variable [Río93].*
3. *$\lambda\sigma$ is not confluent on terms with term and substitution variables [CHL96].*

1.5.2 Extensions and applications

A large amount of work has been done for extending the completion techniques to handle rewriting modulo an equational theory or conditional rewriting, and to apply superposition and simplification in other contexts of automated deduction. Let us sketch below a few directions. Further references can also be found in [HKLR92, Kir95b].

Unfailing completion.

Completion procedures can abort on an equality that cannot be oriented into a terminating rule. The idea of extending completion by computing equational consequences of non-orientable equalities can be traced back to [Bro75, Lan75]. In 1985, the notion of an *unfailing completion* was proposed: in this framework, orientable instances of an equality are used to perform reductions, even if the equality itself is not orientable. For instance, the commutativity axiom $(x * y = y * x)$ may have an instance $(f(x) * x = x * f(x))$ that can be oriented with a lexicographic path ordering. Such an instance reduces the term $(f(a) * a)$ into $(a * f(a))$. This method requires a reduction ordering $>$ which can be extended to an ordering total on ground terms. Deduction of new equalities needs the definition of *ordered critical pairs* between two equalities of E, obtained

by unifying one term of an equality with a non-variable subterm of the other one. The unfailing completion procedure, also called ordered completion, has only two possible outcomes: either it generates a finite set of equalities, or it diverges. In the first case, it provides a decision procedure for validity of any equational theorem. In the second case, it provides a semi-decision procedure [HR87, Rus87, Bac91, BDP89]. Another use of unfailing completion is a familiar method for mathematical proof, namely proof by contradiction: assume the negation of the formula to be proved and derive a contradiction. To refute this negation, the unfailing completion compute ordered critical pairs, simplifies equalities and eliminates trivial equalities, until it generates a contradiction [BDH86, BDP89]. The unfailing completion has been proved refutationally complete [HR87, Rus87, BDP89, Bac91].

Narrowing and goal solving.

Narrowing is a relation on terms which generalises rewriting by using unification instead of matching, in order to find a rule whose application results in an instantiation followed by one reduction step of the term. This relation has been first introduced in [Fay79, Hul80a, JKK83] in order to perform unification in equational theories presented by a confluent and terminating rewrite system R. Applying a narrowing step on an equation to solve (the goal) in order to deduce new equations (or sub-goals), is again a superposition process of a rule if R into the goal equation. This process is iterated until an equation is found whose both sides are syntactically unifiable. Then the composition of their most general unifier with all the substitutions computed by narrowing yields a unifier modulo R. The narrowing process consists in building all the possible narrowing derivations starting from the equation to be solved, and computes in this way a complete set of unifiers modulo the equational theory defined by R. However, the drawback of such a general method is that it very often diverges and several attempts have been made to restrict the size of the narrowing derivation tree [Hul80a, RKKL85, Rét87, DS88, You89, NRS89, KB91, Chr92, WBK94, LR96]. Another application of narrowing is its use as operational semantics of logic and functional programming languages like BABEL [MR92], EQLOG [GM86] and SLOG [Fri85], among many others. A survey on the integration of functions into logic programming based on narrowing and presenting both theoretical and practical points of view can be found in [Han93].

Deduction with constraints.

Constraints have been introduced in automated deduction since about 1990, although one could find similar ideas in theory resolution [Sti85] and in higher-order resolution [Hue72]. A first motivation for introducing constraints in equational deduction processes is provided by completion modulo a set of axioms A [BD89, JK86a]. A main drawback of this class of completion procedures is an inherent inefficiency, due to the computation of matchers and unifiers modulo A. A natural idea is to use constraints to record unification problems in the theory A and to avoid solving them immediately. Constraints are just checked for satisfiability, which is in general much simpler than finding a complete set of solutions or a solved form, especially in equational theories. Following ideas proposed in [KKR90, MN90, Pet90, BPW89, JM90, Kir95a] on ordering and equality constraints, further investigations have been done in particular to study completeness of deduction with constraints and the impact of constraints on simplification in [BGLS92, NR92a, NR92b, LS93, Lyn94, NR94, Vig94, Lyn95, BGLS95].

First-order theorem proving.

It has been observed since 1975 [Bro75, Lan75, Sla74] that superposition is a restricted form of paramodulation [RW69], while simplification by rewriting is similar to demodulation [WRCS67], used in clausal theorem proving. However, until the eighties, very few theoretical results were known about the effect of simplifying during deduction. An extension of the classical *semantic trees* method was designed in [Pet83, HR86, Rus89] to show the refutational completeness of several refinements of the resolution and paramodulation rules, that allow free interleaving of deduction steps with simplification steps without losing completeness. Another approach with a different proof technique is developed in [BG94], where various refutationally complete calculi for first-order logic with equality allowing elimination of redundant formulas are presented. Ordered completion and completion of equational Horn clauses are derived as special cases.

1.6 Further reading

This chapter is by no means an exhaustive description of rewriting theory and applications. Excellent surveys have been written by G. Huet and D. Oppen [HO80], J.-W. Klop [Klo92], J. Avenhaus and K. Madlener [AM90], N. Dershowitz and J.-P. Jouannaud [DJ90, DJ91], D. Plaisted [Pla93], and the term rewriting approach to automated theorem proving is surveyed for instance in [HKLR92]. Recent advances can be found by looking at proceedings of the Conference on Rewriting Techniques and Applications, in Springer Verlag Lecture Notes number 202, 256, 355, 488, 690, 914, 1103, 1232. Volume 909 in the same series, entitled Term Rewriting is devoted to selected presentations at the French Spring School of Theoretical Computer Science in 1993.

Chapter 2

Introduction to the rewriting calculus

H. Cirstea and C. Kirchner [CK01]

2.1 Introduction

2.1.1 Rewriting, computer science and logic

It is a common claim that rewriting is ubiquitous in computer science and mathematical logic. And indeed the rewriting concept appears from the very theoretical settings to the very practical implementations. Some extreme examples are the mail system under Unix that uses rules in order to rewrite mail addresses in canonical forms (see the `/etc/sendmail.cf` file in the configuration of the mail system) and the transition rules describing the behaviors of tree automata. Rewriting is used in semantics in order to describe the meaning of programming languages [Kah87] as well as in program transformations like, for example, re-engineering of Cobol programs [vdBvDK+96]. It is used in order to compute [Der85], implicitly or explicitly as in Mathematica [Wol99], MuPAD [MuP96] or OBJ [GKK+87], but also to perform deduction when describing by inference rules a logic [GLT89], a theorem prover [JK86] or a constraint solver [JK91]. It is of course central in systems making the notion of rule an explicit and first class object, like expert systems, programming languages based on equational logic [O'D77], algebraic specifications (*e.g.* OBJ [GKK+87]), functional programming (*e.g.* ML [Mil84]) and transition systems (*e.g.* Murphi [DDHY92]).

It is hopeless to try to be exhaustive and the cases we have just mentioned show part of the huge diversity of the rewriting concept. When one wants to focus on the underlying notions, it becomes quickly clear that several technical points should be settled. For example, what kind of objects are rewritten? Terms, graphs, strings, sets, multisets, others? Once we have established this, what is a rewrite rule? What is a left-hand side, a right-hand side, a condition, a context? And then, what is the effect of a rule application? This leads immediately to defining more technical concepts like variables in bound or free situations, substitutions and substitution application, matching, replacement; all notions being specific to the kind of objects that have to be rewritten. Once this is solved one has to understand the meaning of the application of a set of rules on (classes of) objects. And last but not least, depending on the intended use of rewriting, one would like to define an induced relation, or a logic, or a calculus.

In this very general picture, we introduce a calculus whose main design concept is to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule *application* and *result*. We concentrate on *term* rewriting, we introduce a very general notion of rewrite rule and we make the rule application and result explicit concepts. These are the basic ingredients of the *rewriting-* or *ρ-calculus* whose originality comes from the fact that terms, rules, rule application and therefore rule application strategies are all treated at the object level.

2.1.2 How does the rewriting calculus work?

In ρ-calculus we can explicitly represent the application of a rewrite rule, as for example $2 \to s(s(0))$, to a term, *e.g.* the constant 2, as the object $[2 \to s(s(0))](2)$ which evaluates to the singleton $\{s(s(0))\}$. This means that the rule application binary symbol "$[\_](\_)$" is part of the calculus syntax.

As we have seen a rule application can be reduced to a singleton, but it may also fail as in $[2 \to s(s(0))](3)$ that evaluates to the empty set $\emptyset$, or it can be reduced to a set with more than one element as exemplified later in this section and explained in Section 2.2.4. Of course, variables may be used in rewrite rules as in $[x + 0 \to x](4 + 0)$. In this last case the evaluation mechanism of the calculus will reduce the application to $\{4\}$. In fact, when evaluating this expression, the variable x is bound to 4 via a mechanism classically called matching, and the result of the evaluation is obtained by instantiating accordingly the variable x from the right hand side of the rewrite rule. We recover, thus, the classical way term rewriting is acting.

Where this game becomes even more interesting is that "$\_ \to \_$", the rewrite binary operator, is integrally part of the calculus syntax. This is a powerful abstraction operator whose relationship with λ-abstraction [Chu40] could provide a useful intuition: A λ-expression $\lambda x.t$ can be represented in the ρ-calculus as the rewrite rule $x \to t$. Indeed, the β-redex $(\lambda x.t\ u)$ is nothing else than $[x \to t](u)$ (*i.e.* the application of the rewrite rule $x \to t$ to the term u) which reduces to $\{\{x/u\}t\}$ (*i.e.* the application of the substitution $\{x/u\}$ to the term t).

We are aware of other ways to abstract on terms or patterns in lambda-calculus e.g. the works of Colson, Kesner, van Oostrom [Col88, vO90, Kes93] or Peyton-Jones [PJ87]. For example, the λ-calculus with patterns presented in [PJ87] can be given a direct representation in the ρ-calculus. Let us consider, for example, the λ-term $\lambda(PAIR\ x\ y).x$ that selects the first element of a pair and the application $\lambda(PAIR\ x\ y).x\ (PAIR\ a\ b)$ that evaluates to a. The representation in the ρ-calculus of the first λ-term is $PAIR(x, y) \to x$ and the corresponding application $[PAIR(x, y) \to x](PAIR(a, b))$ ρ-evaluates to $\{\{x/a, y/b\}x\}$, that is to $\{a\}$.

Of course we have to make clear what a substitution $\{x/u\}$ is and how it applies to a term. But there is no surprise here and we consider a substitution mechanism that preserves the correct variable bindings via the appropriate α-conversion. In order to make this point clear in the paper, as in [DHK95], we will make a strong distinction between *substitution* (which takes care of variable binding) and *grafting* (that performs replacement directly).

When building abstractions, *i.e.* rewrite rules, there is a priori no restriction. A rewrite rule may introduce new variables as in the rule $f(x) \to g(x, y)$ that when applied to the term $f(a)$ evaluates to $\{g(a, y)\}$, leaving the variable y free. It may also rewrite an object into a rewrite rule as in the application $[x \to (f(y) \to g(x, y))](a)$ that evaluates to the singleton $\{f(y) \to g(a, y)\}$. In this case the variable x is free in the rewrite rule $f(y) \to g(x, y)$ but is bound in the rule $x \to (f(y) \to g(x, y))$. More generally, the object formation in ρ-calculus is unconstrained. Thus, the application of the rule $b \to c$ after the rule $a \to b$ to the term a is written $[b \to c]([a \to b](a))$ and as expected the evaluation mechanism will produce first $[b \to c](\{b\})$ and then $\{c\}$. It also allows us to make use in an explicit and direct way of non-terminating or non-confluent (equational) rewrite systems. For example the application of the rule $a \to a$ to the term a $([a \to a](a))$ terminates, since it is applied only once and does not represent the *repeated* application of the rewrite rule $a \to a$.

So, basic ρ-calculus objects are built from a signature, a set of variables, the abstraction operator "$\to$", the application operator "$[\,](\,)$", and we consider sets of such objects. This gives to the ρ-calculus the ability to handle non-determinism in the sense of sets of results. This is achieved via the explicit handling of reduction result sets, including the empty set that records the fundamental information of rule application failure. For example, if the symbol $+$ is assumed to be commutative then $x + y$ is equivalent modulo commutativity to $y + x$ and thus applying the rule $x + y \to x$ to the term $a + b$ results in $\{a, b\}$. Since there are two different ways to apply (match) this rewrite rule modulo commutativity the result is a set that contains two different elements corresponding to two possibilities. This ability to integrate specific computations in the matching process allows us for example to use the ρ-calculus for deduction modulo purposes as proposed in [DHK98].

To summarize, in ρ-calculus abstraction is handled via the arrow binary operator, matching is used as the parameter passing mechanism, substitution takes care of variable bindings and results sets are handled explicitly.

2.1.3 Rewriting relation versus rewriting calculus

A ρ-calculus term contains all the (rewrite rule) information needed for its evaluation. This is also the case for λ-calculus but it is quite different from the usual way term rewrite *relations* are defined.

The rewrite relation generated by a rewrite system $\mathcal{R} = \{l_1 \to r_1, \ldots, l_n \to r_n\}$ is defined as the smallest transitive relation stable by context and substitution and containing $(l_1, r_1), \ldots, (l_n, r_n)$. For example if $\mathcal{R} = \{a \to f(a)\}$, then the rewrite relation contains $(a, f(a))$, $(a, f(f(a)))$, $(f(a), f(f(a)))$, ... and one says that the derivation $a \to f(a) \to f(f(a)) \to \ldots$ is generated by $\mathcal{R}$.

In ρ-calculus the situation is different since ρ-evaluation will reduce a given ρ-term in which all the rewriting information is explicit. It is customary to say that the rewrite system $a \to a$ is not terminating because it generates the derivation $a \to a \to a \to \ldots$. In ρ-calculus the same infinite derivation should be explicitly built (for example using an iterator) and all the evaluation information should be present in the starting term as in $[a \to a]([a \to a]([a \to a](a)))$ whose evaluation corresponds to the three steps derivation $a \to a \to a \to a$.

There is thus a big difference between the way one can define rewrite derivations generated by a rewrite system and their representation in ρ-calculus: in the first case the derivation construction is implicit and left at the meta-level, in the later case, all rewrite steps should be explicitly built.

2.1.4 Integration of first-order rewriting and higher-order logic

We are introducing a new calculus in a heavily-charged landscape. Why one more? There are several complementary answers that we will make explicit in this work. One of them is the unifying principle of the calculus with respect to algebraic and higher-order theories.

The integration of first-order and higher-order paradigms has been one of the main problems raised since the beginning of the study of programming language semantics and of proof environments. The λ-calculus emerged in the thirties and had a deep influence on the development of theoretical computer-science as a simple but powerful tool for describing programming language semantics as well as proof development systems. Term rewriting for its part emerged as an identified concept in the late sixties and it had a deep influence in the development of algebraic specifications as well as in theorem proving.

Because the two paradigms have a lot in common but have extremely useful complementary properties, many works address the integration of term rewriting with λ-calculus. This has been handled either by enriching first-order rewriting with higher-order capabilities or by adding to λ-calculus algebraic features allowing one, in particular, to deal with equality in an efficient way. In the first case, we find the works on CRS [KvOvR93], XRS [Pag98] and other higher-order rewriting systems [Wol93, NP98], in the second case the works on combination of λ-calculus with term rewriting [Oka89, BT88, GBT89, JO97] to mention only a few.

Our previous works on the control of term rewriting [KKV95, Vit94, BKKR01] led us to introduce the ρ-calculus. Indeed we realized that the tool that is needed in order to control rewriting should be made explicit and could be itself naturally described using rewriting. By viewing the arrow rewrite symbol as an abstraction operator, we strictly generalize the abstraction mechanism of λ-calculus, by making the rule application explicit, we get full control of the rewrite mechanism and as a consequence we obtain with the ρ-calculus a uniform integration of algebraic computation and λ-calculus.

2.1.5 Basic properties and uses of the ρ-calculus

One of the main properties of the calculus we are concentrating on is confluence. We will see that the ρ-calculus is not confluent in the general case. The use of sets for representing the reduction results is the main cause of non-confluence. This comes from the fact that in the definition of a standard rewrite step, a rule is applied only when a successful match is found and in this case the reduced term exists and is unique (even if several matches exist). In ρ-calculus we are in a very different situation since a rule application always yields a unique result consisting either of a non-empty set representing all the possible reduced terms (one per different match) or of an empty set representing the impossibility to apply a standard rewrite step.

The confluence can be recovered if the evaluation rules of ρ-calculus are guided by an appropriate strategy. This strategy should first handle properly the problems related to the propagation of failure over the operators

of the calculus. It should also take care of the correct handling of sets with more than one element in non-linear contexts. We are presenting this strategy whose full details are given in [Cir00].

We will see that the ρ-calculus can be used for representing some simpler calculi as λ-calculus and rewriting even in the conditional case. This is achieved by restricting the syntax and the evaluation rules of the ρ-calculus in order to represent the terms of the two calculi. We then show that for any reduction in the λ-calculus or term rewriting, a corresponding natural reduction in the ρ-calculus can be found.

2.1.6 Structure of this work and paper

The purpose of this chapter is to introduce the ρ-calculus, its syntax and evaluation rules and to show how it can be used in order to naturally encode λ-calculus and standard term rewriting. We also show in [CK01], that it can be used to encode conditional rewriting and that it provides a semantics for the rewrite based language ELAN.

In the next section, we introduce the general ρ_T-calculus, where T is a theory used to internalize specific knowledge like associativity and commutativity of certain operators. We present the syntax of the calculus, its evaluation rules together with examples. We emphasize in particular the important role of the matching theory T. We show in Section 2.3 how ρ-calculus can be used to encode in a uniform way term rewriting and λ-calculus. Then, in Section 2.4, we restrict to the $\rho_\emptyset$-calculus (also shortly denoted ρ-calculus), the calculus where only syntactic matching is allowed (*i.e.* the theory T is assumed to be the trivial one), and we present the confluence properties of this calculus. We assume the reader familiar with the standard notions of term rewriting [DJ90a, Klo90, BN98, KK99] and with the basic notions of λ-calculus [Bar84]. For the basic concepts about rule based constraint solving and *deduction modulo*, we refer respectively to [JK91, KR98b] and [DHK98].

2.2 Definition of the ρ_T-calculus

We assume given in this section a theory T defined equationally or by any other means.

A calculus is defined by the following five components:

1. First its *syntax* that makes precise the formation of the objects manipulated by the calculus as well as the formation of substitutions that are used by the evaluation mechanism. In the case of ρ_T-calculus, the core of the object formation relies on a first-order signature together with rewrite rules formation, rule application and sets of results.

2. The description of the *substitution application* to terms. This description is often given at the meta-level, except for explicit substitution frameworks. For the description of the ρ_T-calculus that we give here, we use (higher-order) substitutions and not grafting, *i.e.* the application takes care of variable bindings and therefore uses α-conversion.

3. The *matching algorithm* used to bind variables to their actual values. In the case of ρ_T-calculus, this is matching modulo the theory T. In practical cases it will be higher-order-pattern matching, or equational matching, or simply syntactic matching or combination of any of these. The matching theory is specified as a parameter (the theory T) of the calculus and when it is clear from the context this parameter is omitted.

4. The *evaluation rules* describing the way the calculus operates. It is the glue between the previous components. The simplicity and clarity of these rules are fundamental for its usability.

5. The *strategy* guiding the application of the evaluation rules. Depending on the strategy employed we obtain different versions and therefore different properties for the calculus.

This section makes explicit all these components for the ρ_T-calculus and comments our main choices.

2.2.1 Syntax of the ρ_T-calculus

Definition 2.2.1 We consider $\mathcal{X}$ a set of variables and $\mathcal{F} = \bigcup_m \mathcal{F}_m$ a set of ranked function symbols, where for all m, $\mathcal{F}_m$ is the subset of function symbols of arity m. We assume that each symbol has a unique arity *i.e.* that the $\mathcal{F}_m$ are disjoint. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the set of first-order terms built on $\mathcal{F}$ using the variables in $\mathcal{X}$. The set of basic ρ-terms, denoted $\varrho(\mathcal{F}, \mathcal{X})$, is the smallest set of objects formed according to the following rules:

- the variables in $\mathcal{X}$ are ρ-terms,
- if $t_1, \ldots, t_n$ are ρ-terms and $f \in \mathcal{F}_n$ then $f(t_1, \ldots, t_n)$ is a ρ-term,
- if $t_1, \ldots, t_n$ are ρ-terms then $\{t_1, \ldots, t_n\}$ is a ρ-term (the empty set is denoted $\emptyset$),
- if t and u are ρ-terms then $[t](u)$ is a ρ-term (application),
- if t and u are ρ-terms then $t \to u$ is a ρ-term (abstraction or rewrite rule).

The set of basic ρ-terms can thus be inductively defined by the following grammar:

$$\rho\text{-terms} \quad t \quad ::= \quad x \mid f(t, \ldots, t) \mid \{t, \ldots, t\} \mid [t](t) \mid t \to t$$

where $x \in \mathcal{X}$ and $f \in \mathcal{F}$. Notice that this syntax does not make use of the theory T.

A term may be viewed as a *finite labeled ordered tree*, the leaves of which are labeled with variables or constants and the internal nodes of which are labeled with symbols of positive arity.

Definition 2.2.2 A *position* (also called *occurrence*) of a term (seen as a tree) is represented as a sequence ω of positive integers describing the path from the root of t to the root of the sub-term at that position. We denote by $t_{\lceil s \rceil_p}$ the term t containing the sub-term s at the position p. The symbol at the position p of a term t is denoted by $t(p)$.

We call *functional position* of a ρ-term t, any occurrence p of the term whose symbol belongs to $\mathcal{F}$, *i.e.* $t(p) \in \mathcal{F}$. The set of all positions of a term t is denoted by $\mathcal{P}os(t)$. The set of all functional positions of a term t is denoted by $\mathcal{FP}os(t)$.

The position of a sub-term in a set ρ-term is obtained by considering one of the possible tree representations of the respective ρ-term.

We adopt a very general discipline for the rewrite rule formation, and we do not enforce any of the standard restrictions often used in the term rewriting community like non-variable left-hand sides or occurrence of the right-hand side variables in the left-hand side. We also consider rewrite rules containing rewrite rules as well as rewrite rule application. For convenience, we consider that the symbols $\{\}$ and $\emptyset$ both represent the empty set. We usually use the notation f instead of $f()$ for a function symbol of arity 0 (*i.e.* a constant). For the terms of the form $\{t_1, \ldots, t_n\}$ we assume, as usually, that the comma is an associative, commutative and idempotent function symbol.

The main intuition behind this syntax is that a rewrite rule is an abstraction, the left-hand side of which determines the bound variables and some contextual information. Having new variables in the right-hand side is just the ability to have free variables in the calculus. We will come back to this later but to support the intuition let us mention that the λ-terms [Bar84] and standard first-order rewrite rules [DJ90a, BN98] are clearly objects of this calculus. For example, the λ-term $\lambda x.(y\ x)$ corresponds to the ρ-term $x \to [y](x)$ and a rewrite rule in first-order rewriting corresponds to the same rewrite rule in the rewriting-calculus.

We have chosen sets as the data structure for handling the potential non-determinism. A set of terms can be seen as the set of distinct results obtained by applying a rewrite rule to a term. Other choices could be made depending on the intended use of the calculus. For example, if we want to provide all the results of an application, including the identical ones, a multi-set could be used. When the order of the computation of the results is important, lists could be employed. Since in this presentation of the calculus we focus on the possible results of a computation and not on their number or order, sets are used. The confluence properties

presented in Section 2.4 are preserved in a multi-set approach. It is clear that for the list approach only a confluence modulo permutation of lists can be obtained.

The following examples show the very expressive syntax that is allowed for ρ-terms.

Example 2.2.1 If we consider $\mathcal{F}_0 = \{a, b, c\}$, $\mathcal{F}_1 = \{f\}$, $\mathcal{F}_2 = \{g\}$, $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2$ and x, y variables in $\mathcal{X}$, some ρ-terms from $\varrho(\mathcal{F}, \mathcal{X})$ are:

- $[a \to b](a)$; this denotes the application of the rewrite rule $a \to b$ to the term a. We will see that evaluating this application results in $\{b\}$.
- $[g(x, y) \to f(x)](g(a, b))$; a classical rewrite rule application.
- $[x \to x + y](a)$; a rewrite rule with a free variable y. We will see later why the result of this application is $\{a + y\}$ where the variable y remains free.
- $[y \to [x \to x+y](b)]([x \to x](a))$; a ρ-term that corresponds to the λ-term $(\lambda y.((\lambda x.x+y)\ b))\ ((\lambda x.x)\ a)$. In the rewrite rule $x \to x + y$ the variable y is free but in the rewrite rule $y \to [x \to x + y](b)$ this variable is bound.
- $[x \to [x](x)](x \to [x](x))$; the well-known $(\omega\omega)$ λ-term. We will see that the evaluation of this term is not terminating.
- $[[(x \to x + 1) \to (1 \to x)](a \to a + 1)](1)$; a more complicated ρ-term without corresponding standard rewrite rule or λ-term.

2.2.2 Grafting versus substitution

Since we are dealing with $\to$ as a binder, like for any calculus involving binders (as the λ-calculus), α-conversion should be used to obtain a correct substitution calculus and the first-order substitution (called here grafting) is not directly suitable for the ρ-calculus. We consider the usual notions of α-conversion and higher-order substitution as defined for example in [DHK95].

This is the reason for introducing an appropriate notion of bound variables renaming in Definition 2.2.4. It computes a variant of a ρ-term which is equivalent modulo α-conversion to the initial term.

Definition 2.2.3 The set of free variables of a ρ-term t is denoted by $FV(t)$ and is defined by:

1. if $t = x$ then $FV(t) = \{x\}$,
2. if $t = f(u_1, \ldots, u_n)$ then $FV(t) = \bigcup_{i=1,\ldots,n} FV(u_i)$,
3. if $t = \{u_1, \ldots, u_n\}$ then $FV(t) = \bigcup_{i=1,\ldots,n} FV(u_i)$,
4. if $t = [u](v)$ then $FV(t) = FV(u) \cup FV(v)$,
5. if $t = u \to v$ then $FV(t) = FV(v) \setminus FV(u)$.

Definition 2.2.4 Given a set $\mathcal{Y}$ of variables, the application $\alpha_{\mathcal{Y}}$ (called α-conversion) is defined by:

- $\alpha_{\mathcal{Y}}(x) = x$,
- $\alpha_{\mathcal{Y}}(f(u_1, \ldots, u_n)) = f(\alpha_{\mathcal{Y}}(u_1), \ldots, \alpha_{\mathcal{Y}}(u_n))$,
- $\alpha_{\mathcal{Y}}(\{t\}) = \{\alpha_{\mathcal{Y}}(t)\}$,
- $\alpha_{\mathcal{Y}}([t](u)) = [\alpha_{\mathcal{Y}}(t)](\alpha_{\mathcal{Y}}(u))$,
- $\alpha_{\mathcal{Y}}(u \to v) = \alpha_{\mathcal{Y}}(u) \to \alpha_{\mathcal{Y}}(v)$, if $FV(u) \cap \mathcal{Y} = \emptyset$,

- $\alpha_{\mathcal{Y}}(u \to v) = (\{x_i \mapsto y_i\}_{x_i \in FV(u)}\ \alpha_{\mathcal{Y}}(u)) \to (\{x_i \mapsto y_i\}_{x_i \in FV(u)}\ \alpha_{\mathcal{Y}}(v))$,
  if $x_i \in FV(u) \cap \mathcal{Y}$ and y_i are "fresh" variables and where $\{x \mapsto y\}$ denotes the replacement of the variable x by the variable y in the term on which it is applied.

This allows us to define the usual substitution and grafting operations:

Definition 2.2.5 A *valuation* θ is a finite binding of the variables $x_1, \ldots, x_n$ to the terms $t_1, \ldots, t_n$, *i.e.* a finite set of couples $\{(x_1, t_1), \ldots, (x_n, t_n)\}$.

From a given valuation θ we can define the following two notions of substitution and grafting:

- the *substitution* extending θ is denoted $\Theta = \{x_1/t_1, \ldots, x_n/t_n\}$,
- the *grafting* extending θ is denoted $\bar{\Theta} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.

Θ and $\bar{\Theta}$ are structurally defined by:

$-\ \Theta(x) = u$, if $(x, u) \in \theta$	$-\ \bar{\Theta}(x) = u$, if $(x, u) \in \theta$
$-\ \Theta(f(t_1 \ldots t_n)) = f(\Theta(t_1) \ldots \Theta(t_n))$	$-\ \bar{\Theta}(f(t_1 \ldots t_n)) = f(\bar{\Theta}(t_1) \ldots \bar{\Theta}(t_n))$
$-\ \Theta(\{t_1, \ldots, t_n\}) = \{\Theta(t_1), \ldots, \Theta(t_n)\}$	$-\ \bar{\Theta}(\{t_1, \ldots, t_n\}) = \{\bar{\Theta}(t_1), \ldots, \bar{\Theta}(t_n)\}$
$-\ \Theta([t](u)) = [\Theta(t)](\Theta(u))$	$-\ \bar{\Theta}([t](u)) = [\bar{\Theta}(t)](\bar{\Theta}(u))$
$-\ \Theta(u \to v) = \Theta(u') \to \Theta(v')$	$-\ \bar{\Theta}(u \to v) = \bar{\Theta}(u) \to \bar{\Theta}(v)$

where we consider that z_i are fresh variables (*i.e.* $\theta z_i = z_i$), the z_i do not occur in u and v and for any $y \in FV(u)$, $z_i \notin FV(\theta y)$, and u', v' are defined by:
$u' = \{y_i \mapsto z_i\}_{y_i \in FV(u)}\ \alpha_{FV(u) \cup \mathcal{V}ar\,\theta}(u)$,
$v' = \{y_i \mapsto z_i\}_{y_i \in FV(u)}\ \alpha_{FV(u) \cup \mathcal{V}ar\,\theta}(v)$.
using the following notations: The set of variables $\{x_1, \ldots, x_n\}$ is called the domain of the substitution Θ or of the grafting $\bar{\Theta}$ and is denoted by $\mathcal{D}om(\Theta)$ or $\mathcal{D}om(\bar{\Theta})$ respectively. The set of all the variables from Θ is $\mathcal{V}ar\,\Theta = \cup_{x \in \mathcal{D}om(\Theta)} \Theta(x) \cup \mathcal{D}om(\Theta)$.

Recall that $\{x_1/t_1, \ldots, x_n/t_n\}$ is the simultaneous substitution of the variables $x_1, \ldots, x_n$ by the terms $t_1, \ldots, t_n$ and not the composition $\{x_1/t_1\} \ldots \{x_n/t_n\}$.

There is nothing new in the definition of substitution and grafting except that the abstraction works here on terms and not only on variables. The burden of variable handling could be avoided by using an explicit substitution mechanism in the spirit of [CHL96]. We sketched such an approach in [CK99] and this is detailed in [Cir00].

2.2.3 Matching

Computing the matching substitutions from a ρ-term t to a ρ-term t' is an important parameter of the ρ_T-calculus. We first define matching problems in a general setting:

Definition 2.2.6 For a given theory T over ρ-terms, a *T-match-equation* is a formula of the form $t \ll^?_T t'$, where t and t' are ρ-terms. A substitution σ is a solution of the T-match-equation $t \ll^?_T t'$ if $T \models \sigma(t) = t'$. A *T-matching system* is a conjunction of T-match-equations. A substitution is a solution of a T-matching system P if it is a solution of all the T-match-equations in P. We denote by $\mathbf{F}$ a T-matching system without solution. A T-matching system is called *trivial* when all substitutions are solution of it.
We define the function *Solution* on a T-matching system $\mathcal{S}$ as returning the set of all T-matches of $\mathcal{S}$ when $\mathcal{S}$ is not trivial and $\{\mathbb{ID}\}$, where $\mathbb{ID}$ is the identity substitution, when $\mathcal{S}$ is trivial.

Notice that when the matching system has no solution the function *Solution* returns the empty set.

Since in general we could consider arbitrary theories over ρ-terms, T-matching is in general undecidable, even when restricted to first-order equational theories [JK91]. In order to overcome this undecidability problem, one can think of using constraints as in constrained higher-order resolution [Hue73] or constrained deduction [KKR90]. But we are interested here in the decidable cases. Among them we can mention higher-order-pattern matching that is decidable and unitary as a consequence of the decidability

of pattern unification [Mil91, DHKP96], higher-order matching which is known to be decidable up to the fourth order [Pad96, Pad00, Dow94, HL78] (the decidability of the general case being still open), many first-order equational theories including associativity, commutativity, distributivity and most of their combinations [Nip89, Rin96].

For example when T is empty, the syntactic matching substitution from t to t', when it exists, is unique and can be computed by a simple recursive algorithm given for example by G. Huet [Hue76]. It can also be computed by the following set of rules $SyntacticMatching$ where $f, g \in \mathcal{F}$ and the symbol $\wedge$ is assumed to be associative and commutative.

$Decomposition$	$(f(t_1, \ldots, t_n) \ll^?_\emptyset f(t'_1, \ldots, t'_n)) \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\bigwedge_{i=1\ldots n} t_i \ll^?_\emptyset t'_i \wedge P$
$SymbolClash$	$(f(t_1, \ldots, t_n) \ll^?_\emptyset g(t'_1, \ldots, t'_m)) \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ if $f \neq g$
$MergingClash$	$(x \ll^?_\emptyset t) \wedge (x \ll^?_\emptyset t') \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ if $t \neq t'$
$VariableClash$	$(f(t_1, \ldots, t_n) \ll^?_\emptyset x) \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ if $x \in \mathcal{X}$

Figure 2.1: $SyntacticMatching$ - Rules for syntactic matching

Proposition 2.2.1 The normal form by the rules in $SyntacticMatching$ of any matching problem $t \ll^?_\emptyset t'$ exists and is unique. After removing from the normal form any duplicated match-equation and the trivial match-equations of the form $x \ll^?_\emptyset x$ for any variable x, if the resulting system is:

1. $\mathbf{F}$, then there is no match from t to t' and $\mathcal{S}olution(t \ll^?_\emptyset t') = \mathcal{S}olution(\mathbf{F}) = \emptyset$,
2. of the form $\bigwedge_{i\in I} x_i \ll^?_\emptyset t_i$ with $I \neq \emptyset$, then the substitution $\sigma = \{x_i/t_i\}_{i\in I}$ is the unique match from t to t' and $\mathcal{S}olution(t \ll^?_\emptyset t') = \mathcal{S}olution(\bigwedge_{i\in I} x_i \ll^?_\emptyset t_i) = \{\sigma\}$,
3. empty, then t and t' are identical and $\mathcal{S}olution(t \ll^?_\emptyset t) = \{\mathbb{ID}\}$.

See [KK99].

Example 2.2.2 If we consider the matching problem $(h(x, g(x, y)) \ll^?_\emptyset h(a, g(a, b))$, first we apply the matching rule $Decomposition$ and we obtain the system with the two match-equations $(x \ll^?_\emptyset a)$ and $(g(x, y) \ll^?_\emptyset g(a, b))$. When we apply the same rule once again for the second equation we obtain $(x \ll^?_\emptyset a)$ and $(y \ll^?_\emptyset b)$ and thus, the initial match-equation is reduced to the system $(x \ll^?_\emptyset a) \wedge (x \ll^?_\emptyset a) \wedge (y \ll^?_\emptyset b)$ and $\mathcal{S}olution(h(x, g(x, y)) \ll^?_\emptyset h(a, g(a, b)) = \{\{x/a, y/b\}\}$.

For the matching problem $(g(x, x) \ll^?_\emptyset g(a, b))$ we apply, as before, $Decomposition$ and we obtain the system $(x \ll^?_\emptyset a) \wedge (x \ll^?_\emptyset b)$. This latter system is reduced by the matching rule $MergingClash$ to $\mathbf{F}$ and thus, $\mathcal{S}olution(g(x, x) \ll^?_\emptyset g(a, b)) = \emptyset$.

This syntactic matching algorithm has an easy and natural extension when a symbol $+$ is assumed to be commutative. In this case, the previous set of rules should be completed with

$CommDec$	$(t_1 + t_2) \ll^?_{C(+)} (t'_1 + t'_2) \wedge P$	$\mapsto\!\!\!\!\rightarrow$
	$((t_1 \ll^?_{C(+)} t'_1 \wedge t_2 \ll^?_{C(+)} t'_2) \vee (t_1 \ll^?_{C(+)} t'_2 \wedge t_2 \ll^?_{C(+)} t'_1)) \wedge P$	

where disjunction should be handled in the usual way. In this case of course the number of matches could be exponential in the size of the initial left-hand sides.

Example 2.2.3 When matching modulo commutativity the term $x + y$, with $+$ defined as commutative, against the term $a + b$, the rule $CommDec$ leads to

$$((x \ll^?_{C_{(+)}} a \wedge y \ll^?_{C_{(+)}} b) \vee (x \ll^?_{C_{(+)}} b \wedge y \ll^?_{C_{(+)}} a))$$

and thus, we obtain two substitutions as solution for the initial matching problem, *i.e.* $\mathcal{S}olution(x+y \ll^?_{C_{(+)}} a+b) = \{\{x/a, y/b\}, \{x/b, y/a\}\}$.

Matching modulo associativity-commutativity (AC) is often used. It could be defined either in a rule based way as in [AK92, KR98b] or in a semantic way as in [Eke95]. A restricted form of associative matching called *list matching* is used in the ASF+SDF system [vD96]. In the Maude system any combination of the associative, commutative and idempotency properties is available [Eke96].

2.2.4 Evaluation rules of the ρ_T-calculus

Assume we are given a theory T over ρ-terms having a decidable matching problem. The use of constraints would allow us to drop this last restriction, but we have chosen here to stick to this simpler situation.

As mentioned above, in the general case, the matching is not unitary and thus we should deal with (empty, finite or infinite) sets of substitutions. We consider a substitution application at the meta-level of the calculus represented by the operator "$\_\langle\!\langle\_\rangle\!\rangle$" whose behavior is described by the meta-rule $Propagate$:

$$Propagate \quad r\langle\!\langle\{\sigma_1, \ldots, \sigma_n, \ldots\}\rangle\!\rangle \quad \rightsquigarrow \quad \{\sigma_1 r, \ldots, \sigma_n r, \ldots\}$$

Notice that since this rule operates at the meta-level of the calculus, it is different from the evaluation rules like $Fire$ and its arrow is denoted differently. A version of the calculus can also be given using explicit substitution [Cir00].

The result of the application of a set of substitutions $\{\sigma_1, \ldots, \sigma_n, \ldots\}$ to a term r is the set of terms $\sigma_i r$, where $\sigma_i r$ represents the result of the (meta-)application of the substitution σ_i to the term r as detailed in Definition 2.2.5 on page 37. Notice that when n is 0, *i.e.* the set of substitutions is empty, the resulting set of instantiated terms is also empty.

The evaluation rules of the ρ_T-calculus describe the application of a ρ-term on another one and specify the behavior of the different operators of the calculus when some arguments are sets. Following their specifications they are described in Figure 2.2 to 2.5 on page 42.

Applying rewrite rules

The application of a rewrite rule at the root position of a term is accomplished by matching the left-hand side of the rewrite rule on the term and returning the appropriately instantiated right-hand side. It is described by the evaluation rule $Fire$ in Figure 2.2. The rule $Fire$, like all the evaluation rules of the calculus, can be applied at any position of a ρ-term.

$$Fire \quad [l \to r](t) \quad \Longrightarrow \quad r\langle\!\langle \mathcal{S}olution(l \ll^?_T t)\rangle\!\rangle$$

Figure 2.2: The evaluation rule $Fire$ of the ρ_T-calculus

The central idea is that applying a rewrite rule $l \to r$ at the root (also called top) occurrence of a term t, written as $[l \to r](t)$, consists in replacing the term r by $r\langle\!\langle\Sigma\rangle\!\rangle$ where Σ is the set of substitutions obtained by T-matching l on t (*i.e.* $\mathcal{S}olution(l \ll^?_T t)$). Therefore, when the matching yields a failure represented by an empty set of substitutions, the result of the application of the rule $Propagate$ and thus of the rule $Fire$ is the empty set.

One can notice that the rule $Fire$ can be expressed without using the meta-rule $Propagate$:

$$Fire \quad [l \to r](t) \quad \rightsquigarrow \quad \{\sigma_1 r, \ldots, \sigma_n r, \ldots\} \text{ where } \{\sigma_1, \ldots, \sigma_n, \ldots\} = Solution(l \ll^?_T t)$$

but we preferred the previous version for a smoother transition to the explicit version of the calculus.

We should point out that, as in λ-calculus, an application can always be evaluated. But, unlike in λ-calculus, the set of results can be empty. More generally, when matching modulo a theory T, the set of resulting matches may be empty, a singleton (as in the empty theory), a finite set (as for associativity-commutativity) or infinite (see [FH83]). We have thus chosen to represent the result of a rewrite rule application to a term as a set. An empty set means that the rewrite rule $l \to r$ fails to apply to t in the sense of a matching failure between l and t.

We denote by $\longrightarrow_{Fire}$ the relation induced by the evaluation rule $Fire$.

Example 2.2.4 Some examples of the application of the evaluation rule $Fire$ are:

- $[a \to b](a) \longrightarrow_{Fire} \{b\}$
- $g(x, [x \to c](a)) \longrightarrow_{Fire} g(x, \{c\})$
- $[a \to b](c) \longrightarrow_{Fire} \emptyset$

Applying operators

In order to push rewrite rule application deeper into terms, we introduce the two $Congruence$ evaluation rules of Figure 2.3. They deal with the application of a term of the form $f(u_1, \ldots, u_n)$ (where $f \in \mathcal{F}_n$) to another term of a similar form. When we have the same head symbol for the two terms of the application $[u](v)$ the arguments of the term u are applied on those of the term v argument-wise. If the head symbols are not the same, an empty set is obtained.

$$\begin{array}{lllll} Cong & [f(u_1, \ldots, u_n)](f(v_1, \ldots, v_n)) & \Longrightarrow & \{f([u_1](v_1), \ldots, [u_n](v_n))\} \\ CongFail & [f(u_1, \ldots, u_n)](g(v_1, \ldots, v_m)) & \Longrightarrow & \emptyset \end{array}$$

Figure 2.3: The evaluation rules $Congruence$ of the ρ_T-calculus

Remark 2.2.1 The $Congruence$ rules are redundant with respect to the evaluation rule $Fire$ modulo an appropriate transformation of the initial term. Indeed, one could notice that the application of a term $f(u_1, \ldots, u_n)$ to another ρ-term t (*i.e.* the ρ-term $[f(u_1, \ldots, u_n)](t)$) evaluates, using the rules $Cong$ and $CongFail$, to the same term as the application of the ρ-term $f(x_1, \ldots, x_n) \to f([u_1](x_1), \ldots, [u_n](x_n))$ on the same term t (*i.e.* the ρ-term $[f(x_1, \ldots, x_n) \to f([u_1](x_1), \ldots, [u_n](x_n))](t)$) using the evaluation rule $Fire$. Although we can express the same computations by using only the evaluation rule $Fire$, we prefer to keep the evaluation rules $Congruence$ in the calculus for an explicit use of these rules and thus, a more concise representation of terms.

Handling sets in the ρ_T-calculus

The reductions describing the behavior of terms containing sets are described by the evaluation rules in Figure 2.4 on the next page:

- The rules $Distrib$ and $Batch$ describe the interaction between the application and the set operators,
- The rules $Switch_L$ and $Switch_R$ describe the interaction between the abstraction and the set operators,

- The rule $OpOnSet$ describe the interaction between the symbols of the signature and the set operators.
- The rule describing the interaction between set operators will be described in the next section.

$Distrib$	$[\{u_1, \ldots, u_n\}](v)$	$\Longrightarrow$	$\{[u_1](v), \ldots, [u_n](v)\}$
$Batch$	$[v](\{u_1, \ldots, u_n\})$	$\Longrightarrow$	$\{[v](u_1), \ldots, [v](u_n)\}$
$Switch_L$	$\{u_1, \ldots, u_n\} \to v$	$\Longrightarrow$	$\{u_1 \to v, \ldots, u_n \to v\}$
$Switch_R$	$u \to \{v_1, \ldots, v_n\}$	$\Longrightarrow$	$\{u \to v_1, \ldots, u \to v_n\}$
$OpOnSet$	$f(v_1, \ldots, \{u_1, \ldots, u_m\}, \ldots, v_n)$	$\Longrightarrow$	$\{f(v_1, \ldots, u_1, \ldots, v_n), \ldots, f(v_1, \ldots, u_m, \ldots, v_n)\}$

Figure 2.4: The evaluation rules Set of the ρ_T-calculus

The set representation for the results of the rewrite rule application has important consequences concerning the behavior of the calculus. We can notice, in particular, that the number of set symbols is unchanged by the evaluation rules $Distrib$, $Batch$, $Switch_L$, $Switch_R$ and $OpOnSet$. This way, for a derivation involving only terms that do not contain empty sets, the number of set symbols in a term counts the number of rules $Fire$ and $Congruence$ that have been applied for its evaluation.

The application of the set of rewrite rules $\{a \to b, a \to c\}$ to the term a (*i.e.* the ρ-term $[\{a \to b, a \to c\}](a)$) is reduced, by using the evaluation rule $Distrib$, to the set containing the application of each rule to the term a (*i.e.* the ρ-term $\{[a \to b](a), [a \to c](a)\}$). It is in particular useful when simulating ordinary term rewriting by a *set* of rewrite rules. Moreover, we can factor a set of rewrite rules having the same left-hand side and use the ρ-term $a \to \{b, c\}$ which is reduced, by applying the evaluation rule $Switch_R$, to $\{a \to b, a \to c\}$. Thus, we can say that the ρ-term $[a \to \{b, c\}](a)$ describes the non-deterministic choice between the application of the rule $a \to b$ to the term a and the application of the rule $a \to c$ to the same term and this application is reduced to the set containing the results of the two applications, *i.e.* $\{\{b\}, \{c\}\}$.

Let us consider the ρ-term $[f(a \to b)](f(a))$ which is reduced, by using the rules $Cong$ and $Fire$, to $\{f(\{b\})\}$ and then, by using the rule $OpOnSet$ to $\{\{f(b)\}\}$. The two set symbols corresponding to the two applications of the evaluation rules $Fire$ and $Cong$ are thus preserved by the application of the rule $OpOnSet$.

A result of the form $\{\}$ (*i.e.* $\emptyset$) represents the failure of a rule application and such failures are *strictly* propagated in ρ-terms by the Set rules. For instance, the ρ-term $g([a \to b](c), \{a\})$ is reduced to $g(\emptyset, \{a\})$ and then, by using the rule $OpOnSet$, to $\emptyset$. One should notice that in this case, the information on the number of $Fire$ and $Congruence$ rules used in the reduction of the sub-term $\{a\}$ is lost.

The rewrite relation generated by the evaluation rules $Fire$, $Congruence$ and the Set rules is finer (*i.e.* contains more elements) than the standard one (without sets) and is obviously non-confluent. A reason for the non-confluence is the lack of a similar evaluation rule for the propagation of sets on sets.

Flattening sets in the ρ_T-calculus

We usually care about the set of results obtained by reducing the redexes and not about the exact trace of the reduction leading to these results. In what follows we present the way this behavior is described in the ρ-calculus.

We use the evaluation rule $Flat$ in Figure 2.5 on the following page that flattens the sets and eliminates the (nested) set symbols. In this case, the information on the number of reduction steps is lost. Notice that this implies that failure (the empty set) is *not* strictly propagated on sets.

$$Flat \quad \{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\} \implies \{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\}$$

Figure 2.5: The evaluation rules $Flat$ of the ρ_T-calculus

The same behavior can be described by two distinct evaluation rules: one that would just flatten the sets and thus preserve the number of set braces, and another one that would eliminate the nested set symbols.

This behavior of the calculus could be summarized by stating that failure propagation by the Set rules is strict on all operators but sets. We will see later that $Fire$ may induce non-strict propagations in some particular cases (see Example 2.4.4 on page 51 on page 51).

The design decision to use sets for representing reduction results has another important consequence concerning the handling of sets with respect to matching. Indeed, sets are just used to store results and we do not wish to make them part of the theory. We are thus assuming that the matching operation used in the $Fire$ evaluation rule is *not* performed modulo the set axioms. As a consequence, this requires in some cases to use a strategy that pushes set braces outside the terms whenever possible.

Every time a ρ-term is reduced using the rules $Fire$ and $Congruence$ of the ρ_T-calculus, a set is generated. These evaluation rules are the ones that describe the application of a rewrite rule at the top level or deeper in a term. The set obtained when applying one of the above evaluation rules can trigger the application of the other evaluation rules of the calculus. These evaluation rules deal with the (propagation of) sets and compute a "set-normal form" for the ρ-terms by pushing out the set braces and flattening the sets.

Therefore, we consider that the evaluation rules of the ρ_T-calculus consist of a set of *deduction* rules ($Fire$, $Cong$, $CongFail$) and a set of *computation* rules ($Distrib$, $Batch$, $Switch_L$, $Switch_R$, $OpOnSet$, $Flat$) and that the reduction behaves as in deduction modulo [DHK98]. This means that we can consider the computation rules as describing a congruence modulo which the deduction rules are applied. In such an approach we say that $[f(a \to b)](f(a))$ reduces to $\{f(\{b\})\}$ which is equivalent to $\{f(b)\}$.

Using the ρ_T-calculus

The aim of this section is to make concrete the concepts we have just introduced by giving a few examples of ρ-terms and ρ-reductions. Many other examples could be found on the ELAN web page [Pro01].

The ρ_T-calculus using syntactic matching (*i.e.* an empty matching theory) is denoted $\rho_\emptyset$-calculus or simply ρ-calculus when there is no ambiguity. We denote by ρ_C-calculus, ρ_A-calculus and ρ_{AC}-calculus the ρ_T-calculus with a matching theory commutative, associative and associative-commutative respectively.

Simple functional programming Let us start with the functional part of the calculus and give the ρ-terms representing some λ-terms. For example, the λ-abstraction $\lambda x.(y\ x)$, where y is a variable, is represented as the ρ-rule $x \to [y](x)$. The application of the above term to a constant a, $(\lambda x.(y\ x)\ a)$ is represented in the ρ-calculus by the application $[x \to [y](x)](a)$. This application reduces, in the λ-calculus, to the term $(y\ a)$ while in the ρ-calculus the result of the reduction is the singleton $\{[y](a)\}$. When a functional representation $f(x)$ is chosen, the λ-term $\lambda x.f(x)$ is represented by the ρ-term $x \to f(x)$ and a similar result is obtained for its application. One should notice that for ρ-terms of this form (*i.e.* that have a variable as a left-hand side) the syntactic matching performed in the ρ-calculus is trivial, *i.e.* it never fails and gives only one result.

There is no difficulty to represent more elaborate λ-terms in the ρ-calculus. Let us consider the term $\lambda x.f(x)\ (\lambda y.y\ a)$ with the following β-derivation: $\lambda x.f(x)\ (\lambda y.y\ a) \longrightarrow_\beta \lambda x.f(x)\ a \longrightarrow_\beta f(a)$. The same derivation can be recovered in the ρ-calculus for the corresponding ρ-term: $[x \to f(x)]([y \to y](a)) \longrightarrow_{Fire} [x \to f(x)](\{a\}) \longrightarrow_{Batch} \{[x \to f(x)](a)\} \longrightarrow_{Fire} \{\{f(a)\}\} \longrightarrow_{Flat} \{f(a)\}$. Of course, several reduction strategies can be used in the λ-calculus and reproduced accordingly in the ρ-calculus. Indeed, we will see in Section 2.3.1 that the ρ-calculus strictly embeds the λ-calculus.

Rewriting Now, if we introduce contextual information in the left-hand sides of the ρ-rules we obtain classical rewrite rules as $f(a) \to f(b)$ or $f(x) \to g(x, x)$. When we apply such a rewrite rule, the matching

can fail and consequently, the application of the rewrite rule can fail. As we have already insisted in the previous sections, the failure of a rewrite rule is not a meta-property in the ρ-calculus but is represented by an empty set (of results). For example, in standard term rewriting we say that the application of the rule $f(a) \to f(b)$ to the term $f(c)$ fails and therefore the term is unchanged. On the contrary, in the ρ-calculus the corresponding term $[f(a) \to f(b)](f(c))$ evaluates to $\emptyset$.

Since, in the ρ-calculus, there is no restriction on the rewrite rules construction, a rewrite rule may use a variable as left-hand side, as in $x \to x + 1$, or it may introduce new variables, as in $f(x) \to g(x, y)$. The free variables of the rewrite rules from the ρ-calculus allow us to dynamically build classical rewrite rules. For example, in the application $[y \to (f(x) \to g(x, y))](a)$, the variable y is free in the rewrite rule $f(x) \to g(x, y)$ but bound in the rule $y \to (f(x) \to g(x, y))$. The above application is reduced to the set $\{f(x) \to g(x, a)\}$ containing a classical rewrite rule.

By using free variables in the right-hand side of a rewrite rule we can also "parameterize" the rules by "strategies", as in the term $y \to [f(x) \to [y](x)](f(a))$ where the term to be applied to x is not explicit in the rule $f(x) \to [y](x)$. When reducing the application $[y \to [f(x) \to [y](x)](f(a))](a \to b)$, the variable y from the rewrite rule is instantiated to $a \to b$ and thus, the result of the reduction is $\{b\}$.

Non-determinism When the matching is done modulo an equational theory we obtain interesting behaviors.

An associative matching theory allows us, for example, to express the fact that an expression can be parenthesized in different ways. Take, for example, the list operator $\circ$ that appends two lists with elements of a given sort *Elem*. Any object of sort *Elem* represents a list consisting of this only object. If we define the operator $\circ$ as associative, the rewrite rule describing the decomposition of a list can be written in the associative ρ_A-calculus $l \circ l' \to l$. When applying this rule to the list $a \circ b \circ c \circ d$ we obtain as result the ρ-term $\{a, a \circ b, a \circ b \circ c\}$. If the operator $\circ$ had not been defined as associative, we would have obtained as the result of the same rule application one of the singletons $\{a\}$ or $\{a \circ b\}$ or $\{a \circ (b \circ c)\}$ or $\{(a \circ b) \circ c\}$, depending on the way the term $a \circ b \circ c \circ d$ is parenthesized.

A commutative matching theory allows us, for example, to express the fact that the order of the arguments is not significant. Let us consider a commutative operator $\oplus$ and the rewrite rule $x \oplus y \to x$ that selects one of the elements of the tuple $x \oplus y$. In the commutative ρ_C-calculus, the application $[x \oplus y \to x](a \oplus b)$ evaluates to the set $\{a, b\}$ that represents the set of non-deterministic choices between the two results. In standard rewriting, the result is not well defined; should it be a or b?

We can also use an associative-commutative theory like, for example, when an operator describes multi-set formation. Let us go back to the $\circ$ operator, but this time we define it as associative-commutative and we use the rewrite rule $x \circ x \circ L \to L$ that eliminates doubleton from lists of sort *Elem*. Since the matching is done modulo associativity-commutativity, this rule eliminates the doubleton no matter what is their position in the structure built using the $\circ$ operator. For instance, in the ρ_{AC}-calculus the application $[x \circ x \circ L \to L](a \circ b \circ c \circ a \circ d)$ evaluates to $\{b \circ c \circ d\}$: the search for the two equal elements is done thanks to associativity and commutativity.

Another facility is due to the use of sets for handling non-determinism. This allows us to easily express the non-deterministic application of a set of rewrite rules to a term. Let us consider, for example, the operator $\otimes$ as a syntactic operator. If we want the same behavior as before for the selection of each element of the couple $x \otimes y$, two rewrite rules should be non-deterministically applied as in the following reduction: $[\{x \otimes y \to x, x \otimes y \to y\}](a \otimes b) \longrightarrow_{Distrib} \{[x \otimes y \to x](a \otimes b), [x \otimes y \to y](a \otimes b)\} \longrightarrow_{Fire} \{\{a\}, \{b\}\} \longrightarrow_{Flat} \{a, b\}$.

2.2.5 Evaluation strategies for the ρ_T-calculus

The last component of a calculus, *i.e.* the strategy $\mathcal{S}$ guiding the application of its evaluation rules, is crucial for obtaining good properties for the ρ-calculus. For example, the main property analyzed for the ρ-calculus is confluence and we will see that if the rule $Fire$ is applied under no conditions at any position of a ρ-term, confluence does not hold.

Let us now define formally the notion of strategy. We specialize here to the ρ-calculus, and the general definition can be found in [KKV95].

Definition 2.2.7 An *evaluation strategy* in the ρ-calculus is a subset of the set of all possible derivations.

For example, the $\mathcal{ALL}$ strategy is the set of *all* derivations, *i.e.* it imposes no restrictions. The empty strategy does not allow any reduction. Standard strategies are call by value or by name, leftmost innermost or outermost, lazy, needed.

The reasons for the non-confluence of the calculus are explained in Section 2.4 and a solution is proposed for obtaining a confluent calculus. The confluent strategy can be given explicitly or as a condition on the application of the rule $Fire$.

2.2.6 Summary

Starting from the notions introduced in the previous sections we give the definition of the ρ_T-calculus.

Definition 2.2.8 Given a set $\mathcal{F}$ of function symbols, a set $\mathcal{X}$ of variables, a theory T on $\varrho(\mathcal{F},\mathcal{X})$ terms having a decidable matching problem, we call ρ_T-calculus (or generically rewriting calculus) a calculus defined by:

1. a non-empty subset $\varrho_{-}(\mathcal{F},\mathcal{X})$ of the $\varrho(\mathcal{F},\mathcal{X})$ terms,
2. the (higher-order) substitution application to terms as defined in Section 2.2.2,
3. the theory T,
4. the set of evaluation rules $\mathcal{E}$: $Fire$, $Cong$, $CongFail$, $Distrib$, $Batch$, $Switch_L$, $Switch_R$, $OpOnSet$, $Flat$,
5. an evaluation strategy $\mathcal{S}$ that controls the application of the evaluation rules. The set $\varrho_{-}(\mathcal{F},\mathcal{X})$ should be stable under the strategy controlled application of the evaluation rules.

We use the notation $\rho_T = (\varrho_{-}(\mathcal{F},\mathcal{X}), T, \mathcal{S})$ to make apparent the main components of the rewriting calculus under consideration.

When the parameters of the general calculus are replaced with some specific values, different variants of the calculus are obtained. The remainder of this paper will be devoted, mainly, to the study of a specific instance of the ρ_T-calculus: the ρ-calculus.

2.2.7 Definition of the ρ-calculus

We define the ρ-calculus as the ρ_T-calculus where the matching theory T is restricted to first-order syntactic matching. As an instance of Definition 2.2.8 we get:

Definition 2.2.9 The ρ-calculus is the calculus defined by:

- the subset $\varrho_\emptyset(\mathcal{F},\mathcal{X})$ of $\varrho(\mathcal{F},\mathcal{X})$ whose rewrite rules are restricted to be of the form $u \to v$ where $u \in \mathcal{T}(\mathcal{F},\mathcal{X})$, *i.e.* u is a first-order term and thus does not contain any set, application or abstraction symbol,
- the higher-order substitution application to terms,
- the matching theory $T = \emptyset$, *i.e.* first-order syntactic matching,
- the set of evaluation rules $\mathcal{R}$ presented in Figure 2.6 on the next page (*i.e.* all the rules of the ρ-calculus but $Switch_L$),
- the evaluation strategy $\mathcal{ALL}$ that imposes no conditions on the application of the evaluation rules.

The ρ-calculus is therefore defined as the calculus $\rho_\emptyset = (\varrho_\emptyset(\mathcal{F},\mathcal{X}), \emptyset, \mathcal{ALL})$.

Example 2.2.5 With the exception of the last term, all the ρ-terms from Example 2.2.1 on page 36 are $\rho_\emptyset$-terms.

$Fire$	$[l \to r](t)$	$\Longrightarrow$	$\{\sigma r\}$ where $\{\sigma\} = \mathcal{S}olution(l \ll^?_T t)$
$Cong$	$[f(u_1, \ldots, u_n)](f(v_1, \ldots, v_n))$	$\Longrightarrow$	$\{f([u_1](v_1), \ldots, [u_n](v_n))\}$
$CongFail$	$[f(u_1, \ldots, u_n)](g(v_1, \ldots, v_m))$	$\Longrightarrow$	$\emptyset$
$Distrib$	$[\{u_1, \ldots, u_n\}](v)$	$\Longrightarrow$	$\{[u_1](v), \ldots, [u_n](v)\}$
$Batch$	$[v](\{u_1, \ldots, u_n\})$	$\Longrightarrow$	$\{[v](u_1), \ldots, [v](u_n)\}$
$Switch_R$	$u \to \{v_1, \ldots, v_n\}$	$\Longrightarrow$	$\{u \to v_1, \ldots, u \to v_n\}$
$OpOnSet$	$f(v_1, \ldots, \{u_1, \ldots, u_m\}, \ldots, v_n)$	$\Longrightarrow$	$\{f(v_1, \ldots, u_1, \ldots, v_n), \ldots, f(v_1, \ldots, u_m, \ldots, v_n)\}$
$Flat$	$\{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\}$	$\Longrightarrow$	$\{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\}$

Figure 2.6: The evaluation rules of the ρ-calculus

The following remarks should be made with respect to the restrictions introduced in the ρ-calculus:

- Since first-order syntactic matching is unitary (*i.e.* the match, when it exists, is unique) the meta-rule *Propagate* from Section 2.2.4 gives always as result either the singleton $\{\sigma r\}$ or the empty set. Hence, the evaluation rule $Fire$ can be replaced by the following simpler two rules:

$$\begin{array}{lllll} Fire' & [l \to r](\sigma l) & \Longrightarrow & \{\sigma r\} & \\ Fire'' & [l \to r](t) & \Longrightarrow & \emptyset & \\ & & & \text{if there exists no } \sigma \text{ s.t. } \sigma l = t & \end{array}$$

- The evaluation rule $Switch_L$ can never be used in the ρ-calculus due to the restricted syntax imposed on $\rho_\emptyset$-terms.
- For a specific instance of the ρ_T-calculus, there is a strong relationship between the terms allowed on the left-hand side of the rule and the theory T. Intuitively, the theory T should be powerful enough to fire rule applications in a way consistent with the intended rewriting. For instance, it seems more interesting to use higher-order matching instead of syntactic or equational matching when the left-hand sides of rules contain abstractions and applications. This explains the restriction imposed in the ρ-calculus for the formation of left-hand sides of rules.
- The term restrictions are made only on the left-hand sides of rewrite rules and not on the right-hand side and this clearly leads to more terms than in λ-calculus or in term rewriting.
- The ρ-calculus is not terminating as $[\omega](\omega)$ is a ρ-term (see Example 2.2.1 on page 36).

The case of decidable finitary equational theories will induce more technicalities but is conceptually similar to the case of the empty theory. The case of theories with infinitary or undecidable matching problems could be treated using constraint ρ-terms in the spirit of [KKR90], and will be studied in forthcoming works.

2.3 Encoding λ-calculus and term rewriting in the ρ-calculus

The aim of this section is to show in detail how the ρ-calculus can be used to give a natural encoding of the λ-calculus and term rewriting.

2.3.1 Encoding the λ-calculus

We briefly present some of the notions used in the λ-calculus, such as β-redex and β-reduction, that will be used in this part of the paper. The reader should refer to [HS86] and [Bar84] for a detailed presentation.

Let $\mathcal{X}$ be a set of variables, written x, y, etc. The terms of the λ-calculus are inductively defined by:

$$a ::= x \mid (a\ a) \mid \lambda x.a$$

Definition 2.3.1 The β-reduction is defined by the rule:

$$Beta \quad (\lambda x.M\ N) \quad \rightsquigarrow \quad \{x/N\}M$$

Any term of the form $(\lambda x.M)N$ is called a *β-redex*, and the term $\{x/N\}M$ is traditionally called its *contractum*. If a term P contains a redex, P can be *β-contracted* into P' which is denoted:

$$P \ \longrightarrow_\beta \ P'.$$

If Q is obtained from P by a finite (possibly empty) number of β-contractions we say that P *β-reduces* to Q and we denote:

$$P \ \xrightarrow{*}_\beta \ Q.$$

Let us consider a restriction of the set of ρ-terms, denoted $\mathcal{F}_\lambda$, and inductively defined as follows:

$$\rho_\lambda\text{-terms} \quad t \quad ::= \quad x \mid \{t\} \mid [t](t) \mid x \to t$$

where $x \in \mathcal{X}$.

Definition 2.3.2 The ρ_λ-calculus is the ρ-calculus defined by:

- the $\mathcal{F}_\lambda$ terms,
- the higher-order substitution application to terms,
- the (matching) theory $T = \emptyset$,
- the set of evaluation rules of the ρ-calculus,
- the evaluation strategy $\mathcal{ALL}$ that imposes no conditions on the application of the evaluation rules.

Compared to the syntax of the general ρ-calculus, the rewrite rules allowed in the ρ_λ-calculus can only have a variable as left-hand side. Additionally, all the sets are singletons, hence one could consider an encoding not using sets. For uniformity purposes, we chose to stick to the same encoding approach.

Because of the syntactic restrictions we have just imposed, the evaluation rules of the ρ-calculus specialize to the ones described in Figure 2.7.

$$\begin{array}{lllll} Fire_\lambda & [x \to r](t) & \Longrightarrow & \{r(x/t)\ \} \\ Distrib_\lambda & [\{u\}](v) & \Longrightarrow & \{[u](v)\} \\ Batch_\lambda & [v](\{u\}) & \Longrightarrow & \{[v](u)\} \\ Switch_\lambda & x \to \{v\} & \Longrightarrow & \{x \to v\} \\ Flat_\lambda & \{\{v\}\} & \Longrightarrow & \{v\} \end{array}$$

Figure 2.7: The evaluation rules of the ρ_λ-calculus

The evaluation rule $Fire_\lambda$ initiates in the ρ-calculus (as the β-rule in the λ-calculus) the application of a substitution to a term. The rules $Congruence$ are not used and the rules Set and $Flat$ can be specialized to singletons and describe how to push out the set braces.

An immediate consequence of the restricted syntax of the ρ_λ-calculus is that the matching performed in the evaluation rule $Fire_\lambda$ always succeeds and the solution of the matching equation that is necessarily of the form $x \ll^?_\emptyset t$ is always the singleton $\{\{x/t\}\}$.

At this moment we can notice that any λ-term can be represented by a ρ-term. The function φ that transforms terms in the syntax of the λ-calculus into the syntax of the ρ_λ-calculus is defined by the following transformation rules:

$$\begin{array}{lcl} \varphi(x) & = & x, \text{ if } x \text{ is a variable} \\ \varphi(\lambda x.t) & = & x \to \varphi(t) \\ \varphi(t\ u) & = & [\varphi(t)](\varphi(u)) \end{array}$$

A similar translation function can be used in order to transform terms in the syntax of the ρ_λ-calculus into the syntax of the λ-calculus:

$$\begin{array}{lcl} \delta(x) & = & x, \text{ if } x \text{ is a variable} \\ \delta(\{t\}) & = & \delta(t) \\ \delta([t](u)) & = & (\delta(t)\ \delta(u)) \\ \delta(x \to t) & = & \lambda x.\delta(t) \end{array}$$

The reductions in the λ-calculus and in the ρ_λ-calculus are equivalent modulo the notations for the application and the abstraction and the handling of sets:

Proposition 2.3.1 Given two λ-terms t and t', if $t \longrightarrow_\beta t'$ then $\varphi(t) \stackrel{*}{\longrightarrow}_{\rho_\lambda} \{\varphi(t')\}$.
Given two ρ_λ-terms u and u', if $u \longrightarrow_{\rho_\lambda} u'$ then $\delta(u) \stackrel{*}{\longrightarrow}_\beta \delta(u')$.

We use an induction on $\longrightarrow_\beta$ and $\longrightarrow_{\rho_\lambda}$ respectively:

- If t is a variable x, then $t' = x$ and $\varphi(t) = \varphi(t') = x$.
- If $t = \lambda x.u$ then $t' = \lambda x.u'$ with $u \longrightarrow_\beta u'$ and we have $\varphi(t) = x \to \varphi(u)$. By induction, we have $\varphi(u) \stackrel{*}{\longrightarrow}_{\rho_\lambda} \{\varphi(u')\}$, and thus

$$\varphi(t) = x \to \varphi(u) \quad \stackrel{*}{\longrightarrow}_{\rho_\lambda} \quad x \to \{\varphi(u')\} \quad \longrightarrow_{Switch_\lambda} \quad \{x \to \varphi(u')\} = \{\varphi(t')\}$$

- If $t = (u\ v)$ then we have either $t' = (u'\ v)$ with $u \longrightarrow_\beta u'$, or $t' = (u\ v')$ with $v \longrightarrow_\beta v'$, or $t = \lambda x.u\ v$ and $t' = u(x/v)$.

  In the first case, we apply induction and we obtain

$$\varphi(t) = [\varphi(u)](\varphi(v)) \stackrel{*}{\longrightarrow}_{\rho_\lambda} [\{\varphi(u')\}](\varphi(v)) \longrightarrow_{Distrib_\lambda} \{[\varphi(u')](\varphi(v))\} = \{\varphi(t')\}.$$

  The second case is similar,

$$\varphi(t) = [\varphi(u)](\varphi(v)) \stackrel{*}{\longrightarrow}_{\rho_\lambda} [\{\varphi(u)\}](\varphi(v')) \longrightarrow_{Distrib_\lambda} \{[\varphi(u)](\varphi(v'))\} = \{\varphi(t')\}.$$

  In the third case $\varphi(t) = [x \to \varphi(u)](\varphi(v))$ and

$$\varphi(t) = [x \to \varphi(u)](\varphi(v)) \longrightarrow_{Fire_\lambda} \{\varphi(x/\varphi(v))(u)\} = \varphi(u(x/v)) = \varphi(t').$$

  Since the application of a substitution is the same in the λ-calculus and the ρ-calculus, we have, due to the definition of φ, $\varphi(u(x/v)) = \varphi(x/\varphi(v))(u)$ and thus, the property is verified.

Since in the ρ_λ-calculus we can have only singletons and the δ transformation strips off the set symbols, the application of the evaluation rules $Distrib_\lambda$, $Batch_\lambda$, $Switch_\lambda$ and $Flat_\lambda$ corresponds to the identity in the λ-calculus.

- If $t = [\{u\}](v)$ then we have $t \longrightarrow_{Distrib_\lambda} \{[u](v)\}$. Since $\delta([\{u\}](v)) = \delta(u)\ \delta(v)$ and $\delta(\{[u](v)\}) = \delta(u)\ \delta(v)$, the property is verified.

- If $t = [x \to u](v)$ then $t \longrightarrow_{Fire_\lambda} \{u(x/v)\}$. We have

$$\delta(t) = \lambda x.\delta(u)\ \delta(v) \longrightarrow_\beta \delta(x/\delta(v))(u)\} = \delta(u(x/v)) = \delta(t').$$

The other cases are very similar to the first one and to their correspondents from the first part.

Example 2.3.1 We consider the three combinators $I = \lambda x.x$, $K = \lambda xy.x$ and $S = \lambda xyz.xz(yz)$ and their representation in the ρ-calculus:

- $I = x \to x$,
- $K = x \to (y \to x)$,
- $S = x \to (y \to (z \to [[x](z)]([y](z))))$.

and, as expected, to a reduction $SKK \xrightarrow{*}_\beta I$ in the λ-calculus it corresponds the ρ_λ-reduction $[[S](K)](K) \xrightarrow{*}_{\rho_\lambda} \{I\}$.

$$\begin{array}{l}
[[S](K)](K) = [[x \to (y \to (z \to [[x](z)]([y](z))))](x \to (y \to x))](x \to (y \to x)) \longrightarrow_{\rho_\lambda} \\
\quad [\{y \to (z \to [[x \to (y \to x)](z)]([y](z)))\}](x \to (y \to x)) \longrightarrow_{\rho_\lambda} \\
\quad \{[y \to (z \to [[x \to (y \to x)](z)]([y](z)))](x \to (y \to x))\} \longrightarrow_{\rho_\lambda} \\
\quad \{[y \to (z \to [\{y \to z\}]([y](z)))](x \to (y \to x))\} \longrightarrow_{\rho_\lambda} \\
\quad \{\{[y \to (z \to [y \to z]([y](z)))](x \to (y \to x))\}\} \longrightarrow_{\rho_\lambda} \\
\quad \{\{[y \to (z \to \{z\})](x \to (y \to x))\}\} \longrightarrow_{\rho_\lambda} \\
\quad \{\{\{[y \to (z \to z)](x \to (y \to x))\}\}\} \longrightarrow_{\rho_\lambda} \\
\quad \{\{\{\{z \to z\}\}\}\} \longrightarrow_{\rho_\lambda} \\
\quad \{z \to z\} = \{I\}
\end{array}$$

The need for adding a set symbol comes from the fact that in the ρ-calculus we are mainly interested in the application of terms to some other terms. From this point of view, the application of a term t to another term u reduces to the same thing as the application of the term $\{t\}$ to the same term u.

In the ρ_λ-calculus, we could have introduced an evaluation rule eliminating all set symbols. But as soon as failure, represented by the empty set, and non-determinism, represented by sets with more than one element, are introduced such an evaluation rule will not be meaningful anymore.

The confluence of the λ-calculus holds for any complete reduction strategy (*i.e.* a strategy that does not leave any redex un-reduced) and we would expect the same result for its ρ-representation. As we have already noticed, since in the ρ_λ-calculus all the rewrite rules are left-linear and all the sets are singletons, the confluence conditions that will be presented in Section 2.4.2 are always satisfied. Therefore, the evaluation rule $Fire_\lambda$ can be used on any ρ_λ-application without losing the confluence of the ρ_λ-calculus.

Proposition 2.3.2 The ρ_λ-calculus is confluent.

Notice finally that using the same technique, the λ-calculus with patterns of [PJ87] can be encoded as a sub-calculus of the ρ-calculus.

2.3.2 Encoding finite rewrite sequences

As far as it concerns term rewriting, we just recall the basic notions that are consistent with [DJ90a, BN98] to which the reader is referred for a more detailed presentation.

A *rewrite theory* is a 4-tuple $\mathcal{R} = (\mathcal{X}, \mathcal{F}, E, R)$ where $\mathcal{X}$ is a given countably infinite set of variables, $\mathcal{F}$ a set of ranked function symbols, E a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities, and R a set of rewrite rules of the form $l \to r$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ satisfying $\mathcal{V}ar\, r \subseteq \mathcal{V}ar\, l$.

In what follows we consider $E = \emptyset$ but we conjecture that all the results concerning the encoding of rewriting in ρ-calculus can be smoothly extended to any equational theory E.

Since the rewrite rules are trivially ρ-terms, the representation of rewrite sequences in the ρ-calculus is quite simple. We consider a restriction of the ρ-calculus where the right-hand sides of rewrite rules are terms

of $\mathcal{T}(\mathcal{F},\mathcal{X})$. The rewrite rules are trivially translated in the ρ-calculus and the application of a rewrite rule at the top position of a term is represented using the ρ-operator $[\_](\_)$.

We want to show that for any derivation in a rewriting theory, a corresponding reduction can be found in the ρ-calculus. If we consider that a sub-term w of a term t is reduced to w' by applying some rewrite rule $(l \to r)$ and thus,

$$t_{\lceil w \rceil_p} \longrightarrow_{\mathcal{R}} t_{\lceil w' \rceil_p}$$

then, we can build immediately the ρ-term $t_{\lceil [l\to r](w) \rceil_p}$ with the reduction:

$$t_{\lceil [l\to r](w) \rceil_p} \longrightarrow_{\rho} t_{\lceil \{w'\} \rceil_p} \xrightarrow{*}_{\rho} \{t_{\lceil w' \rceil_p}\}.$$

The above construction method for the ρ-term with a ρ-reduction similar to that of the term t according to the rule $l \to r$ is very easy but allows us to find the correspondence for only one rewrite step. It is not easy to extend this representation for an unspecified number of reduction steps w.r.t. a set of rewrite rules and a systematic method for the construction of the corresponding ρ-term is desirable.

Proposition 2.3.3 Given a rewriting theory $\mathcal{T}_{\mathcal{R}}$ and two first order ground terms $t, t' \in \mathcal{T}(\mathcal{F})$ such that $t \xrightarrow{*}_{\mathcal{R}} t'$. Then, there exist the ρ-terms $u_1, \ldots, u_n$ built using the rewrite rules in $\mathcal{R}$ and the intermediate steps in the derivation $t \xrightarrow{*}_{\mathcal{R}} t'$ such that we have $[u_n](\ldots[u_1](t)\ldots) \xrightarrow{*}_{\rho_\emptyset} \{t'\}$.

We use induction on the length of the derivation $t \xrightarrow{*}_{\mathcal{R}} t'$.

The base case: $t \xrightarrow{0}_{\mathcal{R}} t$ (derivation in 0 steps)

We have immediately $[x \to x](t) \xrightarrow{0}_{\rho_\emptyset} \{t\}$.

Induction: $t \xrightarrow{n}_{\mathcal{R}} t'$ (derivation in n steps)

We consider that the rewrite rule $l \to r$ is applied at position p of the term $t'_{\lceil w \rceil_p}$ obtained after $n-1$ reduction steps,

$$t \quad \xrightarrow{n-1}_{\mathcal{R}} \quad t'_{\lceil w \rceil_p} \quad \longrightarrow_{l\to r,p} \quad t'_{\lceil \theta r \rceil_p}$$

where θ is the grafting such that $\theta l = w$.

By induction, there exist the ρ-terms $u_1, \ldots, u_{n-1}$ such that we have the reduction $[u_{n-1}](\ldots[u_1](t)\ldots) \xrightarrow{*}_{\rho_\emptyset} \{t'_{\lceil w \rceil_p}\}$. We consider the ρ-term $u_n = t'_{\lceil l\to r \rceil_p}$ and we obtain the reduction

$[u_n](\ldots[u_1](t)\ldots) \xrightarrow{*}_{\rho_\emptyset} [t'_{\lceil l\to r \rceil_p}](\{t'_{\lceil w \rceil_p}\}) \longrightarrow_{Batch} \{[t'_{\lceil l\to r \rceil_p}](t'_{\lceil w \rceil_p})\}$
$\xrightarrow{*}_{Congruence} \{\{t'_{\lceil [l\to r](w) \rceil_p}\}\} \longrightarrow_{Fire} \{\{t'_{\lceil \{\theta' r\} \rceil_p}\}\} \xrightarrow{*}_{OpOnSet} \{\{\{t'_{\lceil \theta' r \rceil_p}\}\}\}$
$\xrightarrow{*}_{Flat} \{t'_{\lceil \theta' r \rceil_p}\}$

where the substitution θ' is such that $\{\theta'\} = \mathcal{S}olution(l \ll^?_\emptyset w)$.

Since $\theta = \theta'$ and in this case substitution and grafting are identical, we obtain $t'_{\lceil \theta' r \rceil_p} = t'_{\lceil \theta r \rceil_p}$.

Until now we have used the evaluation rule *Cong* for constructing the reduction

$$[t^n{}_{\lceil l_n \to r_n \rceil_{p_n}}](\ldots[t^2{}_{\lceil l_2\to r_2 \rceil_{p_2}}]([t^1{}_{\lceil l_1\to r_1 \rceil_{p_1}}](t))\ldots) \quad \xrightarrow{*}_{\rho} \quad \{t'\}$$

that corresponds, in the ρ-calculus, to the reduction, in the rewrite theory,

$$t = t^1{}_{\lceil w_1 \rceil_{p_1}} \quad \longrightarrow_{l_1\to r_1,p_1} \quad t^2{}_{\lceil w_2 \rceil_{p_2}} \quad \longrightarrow_{l_2\to r_2,p_2} \quad \ldots \quad \longrightarrow_{l_n\to r_n,p_n} \quad t^n{}_{\lceil w_n \rceil_{p_n}} = t'$$

As explained in Section 2.2.4, to any reduction performed using the rule *Cong* corresponds a reduction that is done using the rule *Fire*. Starting from the term u corresponding to a reduction in n (*Cong*) steps we build the term u' that reduces to the same term as u but using *Fire* reductions:

$$[t^n{}_{\lceil l_n \rceil_{p_n}} \to t^n{}_{\lceil r_n \rceil_{p_n}}](\ldots([t^1{}_{\lceil l_1 \rceil_{p_1}} \to t^1{}_{\lceil r_1 \rceil_{p_1}}](t))\ldots) \quad \xrightarrow{*}_{\rho} \quad \{t'\}$$

Remark 2.3.1 One can notice that the terms u_i used in the proof above are similar to the proof terms used in labeled rewriting logic [Mes92]. Indeed we can see the ρ-terms as a generalization of such proof terms where the ";" is used as a notation for the composition of terms, *i.e.* $[u]([v](t))$ is denoted $[v;u](t)$.

2.4 The confluence of ρ-calculus

It is easy to see, and we provide typical examples just below, that the ρ-calculus is non-confluent. The main reason for the confluence failure comes from the introduction in the syntax of the new function symbols for denoting sets, abstraction and application. It results in a conflict between the use of syntactic matching and the set representation for the reductions results. This leads, on one hand, to undesirable matching failures due to terms that are not completely evaluated or not instantiated. On the other hand, we can have sets with more than one element that can lead to undesirable results in a non-linear context or empty sets that are not strictly propagated. In this section, we summarize the results of [Cir00] to which the reader is referred for full details. In particular we show on typical examples the confluence problems and we give a sufficient condition on the evaluation strategy of the ρ-calculus that allows to restore confluence.

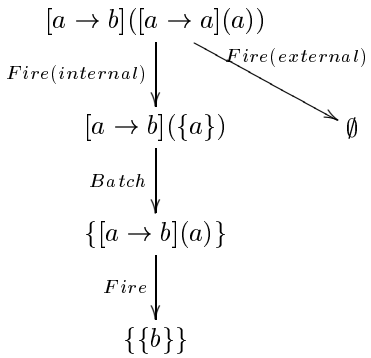
2.4.1 The raw ρ-calculus is not confluent

Let us begin to show typical examples of confluence failure. A first such situation occurs when reducing a (sub-)term of the form $u = [l \to r](t)$ by matching l and t and when either t contains a redex, or u is redex.

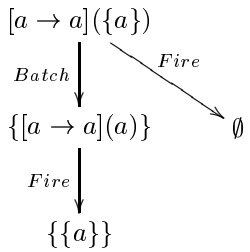
In Example 2.4.1.a the non-confluence is obtained when a matching failure results from a non-reduced sub-term of t but succeeds when the sub-term is reduced. A similar situation is obtained when the evaluation rule $Fire$ gives the $\emptyset$ result due to a matching failure but the application of another evaluation rule before the rule $Fire$ leads to a non-empty set as in Example 2.4.1.b.

Example 2.4.1

a.

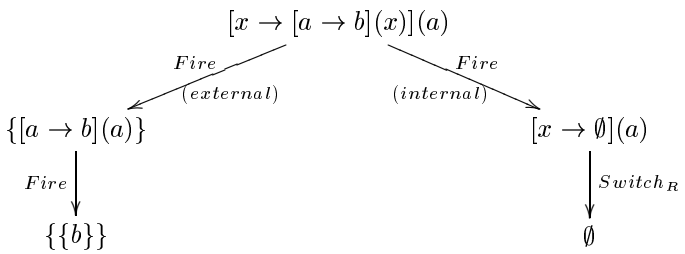


b.



In Example 2.4.2 one can notice that a term can be reduced to an empty set because of a matching failure implying its bound variables. The result can be different from the empty set if the reductions of the sub-terms containing the respective variables are carried out only after the instantiation of these variables.

Example 2.4.2



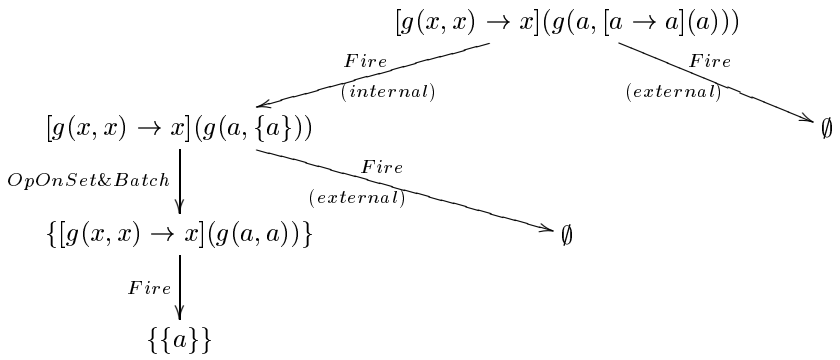
In order to avoid this kind of situation we should prevent the reduction of an application $[l \to r](t)$ if the matching between the terms l and t fails due to the matching rules $VariableClash$ (Example 2.4.2) or $SymbolClash$ (Example 2.4.1.a, 2.4.1.b) and either some variables are not instantiated or some of the terms are not reduced, or the term t is a set.

The matching rules $VariableClash$ and $SymbolClash$ would be never applied if the set of functional positions of the term l was a subset of the set of functional positions of the term t. This is not the case in Example 2.4.2 where, in the term $[a \to b](x)$, a is a functional position and the corresponding position

in the argument of the rewrite rule application is the variable position x. In Example 2.4.1 on the facing page.a and Example 2.4.1 on the preceding page.b a functional position in the left-hand side of the rewrite rule corresponds to an abstraction and set position respectively and thus, the condition is not satisfied.

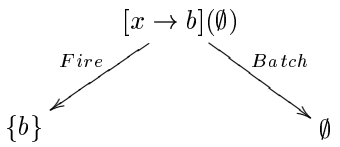
Therefore, we could consider that the evaluation rule $Fire$ is applied only when the condition on the functional positions is satisfied. Unfortunately, such a condition will not suffice for avoiding a non-appropriate matching failure due to the application of the rule $MergingClash$. As shown in Example 2.4.3, such a situation can be obtained if the left-hand side of the rewrite rule to be applied is not linear.

Example 2.4.3



Another pathological case arises when the term t contains an empty set or a sub-term that can be reduced to the empty set. Indeed, the application of the rule $Fire$ can lead to the non-propagation of the failure and thus, to non-confluence as in the next example:

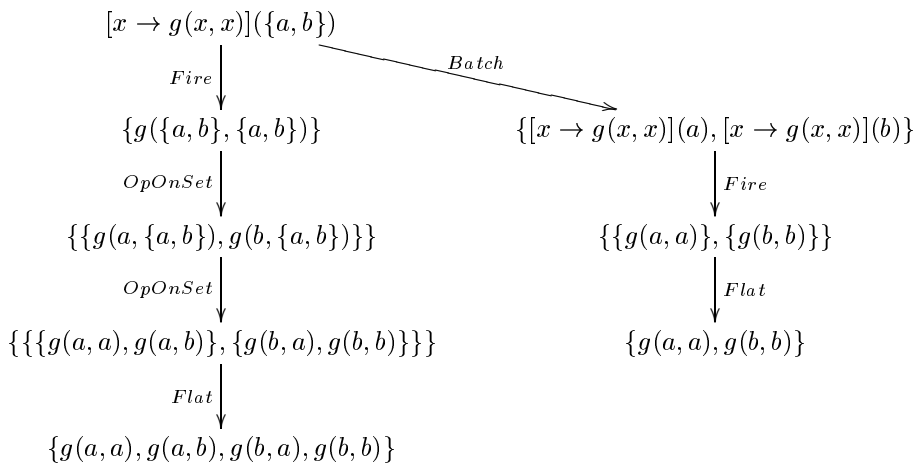
Example 2.4.4



We mention that a rewrite rule is *quasi-regular* if the set of variables of the left-hand side is included in the set of variables of the right-hand side. In Section 2.4.2 we give a formal definition for the notion of quasi-regular rewrite rule that takes into consideration all the operators of the ρ-calculus. We have already seen in Example 2.4.4 that the non-propagation of the failure is obtained when non-quasi-regular rewrite rules are applied to a term containing $\emptyset$. When a quasi-regular rewrite rule is applied to a term containing $\emptyset$, the empty set is present in the term resulting from the application of a substitution of the form $\{x/\emptyset\}$ to the right-hand side of the rewrite rule (unlike in Example 2.4.4) and thus, the appropriate propagation of the $\emptyset$ is guaranteed.

Another nasty situation, well known, in particular in graph rewriting, is obtained due to uncontrolled copies of terms. When applying a non-right-linear rewrite rule to a term that contains sets with more than one element, or terms that can be reduced to such sets, we obtain undesirable results as in Example 2.4.5.

Example 2.4.5



To sum-up, the non-confluence is due to the application of the evaluation rule $Fire$ too early in a derivation and the typical situations that we want to avoid consist in using the rule $Fire$ for reducing an application:

- containing non-instantiated variables,
- containing non-reduced terms,
- containing a non-left-linear rewrite rule,
- of a non-right-linear rewrite rule to a term containing sets with more than one element,
- of a non-quasi-regular rewrite rule to a term containing empty sets.

We can notice that if we assume the computation rules (see Section 2.2.4) to be applied eagerly, then some, but unfortunately not all of the above confluence problems vanish. In particular, non-confluence examples involving sets, as Example 2.4.4 on the preceding page and Example 2.4.5 on the page before, are overcome by an eager application of the computation rules.

2.4.2 Enforcing confluence using strategies

As we have just seen in the previous section, the possibility of having empty sets or sets with more than one element leads immediately to non-confluent reductions implying the evaluation rules $Fire$ and $Congruence$. But the confluence could be restored under an appropriate evaluation strategy and, in particular, this strategy should guarantee a strict failure propagation and an appropriate handling of the sets with more than one element.

A first possible approach consists in reducing a ρ-term by initially applying all the rules handling the sets ($Distrib$, $Batch$, $Switch_L$, $Switch_R$, $OpOnSet$, $Flat$), *i.e.* the computation rules, and only when none of these rules can be applied, apply one of the rules $Fire$, $Cong$, $CongFail$, *i.e.* the deduction rules, to the terms containing no sets.

But an application can be reduced, by using the rule $Fire$, to an empty set or to a set containing several elements and thus, this strategy can still lead, as previously, to non-confluent reductions. Another disadvantage of this approach is that for no restriction of the ρ-calculus the proposed strategy is reduced to the trivial strategy $\mathcal{ALL}$.

Since the sets (empty or having more than one element) are the main cause of the non-confluence of the calculus, a natural strategy consists in reducing the application of a rewrite rule by respecting the following steps: instantiate and reduce the argument of the application, push out the resulting set braces by distributing them in the terms and only when none of the previous reductions is possible, use the evaluation rule $Fire$. We can easily express this strategy by imposing a simple condition for the application of the evaluation rule $Fire$.

Definition 2.4.1 We call *ConfStratStrict* the strategy which consists in applying the evaluation rule $Fire$ to a redex $[l \to r](t)$ only if the term t is a first order ground term.

Proposition 2.4.1 When using the evaluation strategy *ConfStratStrict*, the ρ-calculus is confluent.

We consider the parallelization of the relation induced by the evaluation rules $Fire$ and $Congruence$ on one hand and the relation induced by the other rules of the calculus on the other hand. We show the confluence of the two relations and then use Yokouchi's Lemma [YH90] to prove the strong confluence of the relation obtained by combining the former relations. This latter relation is the transitive closure of the relation induced by the evaluation rules $Fire$ and $Congruence$, and the evaluation rules handling sets.

The Yokouchi Lemma can be easily proved due to the strict conditions on the application of the rule $Fire$ and thus to the absence of interaction between the evaluation rules of the calculus.

The strategy *ConfStratStrict* is quite restrictive and we would like to define a general strategy that becomes trivial (*i.e.* imposes no restriction) when restricted to some simpler calculi, as the λ-calculus.

A confluent strategy emerges from the above counterexamples and allows the application of the evaluation rule $Fire$ only if a possible failure in the matching is preserved by the subsequent ρ-reductions and if the argument of the application cannot be reduced to an empty set or to a set having more than one element. Such a generic strategy consists in applying the evaluation rule $Fire$ to a redex $[l \to r](t)$ only if:

- $t \in \mathcal{T}(\mathcal{F})$ is a first order ground term

or

- the term t is such that if the matching $l \ll^?_\emptyset t$ fails then, for all term t' obtained by instantiating or reducing t, the matching $l \ll^?_\emptyset t'$ fails, and
- the term t cannot be reduced to an empty set or to a set having more than one element.

If we consider an instance of the ρ-calculus such that all the sets are singletons and all the applications are of the form $[x \to u](v)$ then, all the above conditions are always satisfied. Hence, we can say that in this case the previous strategy is equivalent to the strategy $\mathcal{ALL}$, *i.e.* it imposes no restriction on the reductions. One can notice that the ρ_λ-terms satisfy the previous conditions and thus, such a strategy imposes no restrictions on the reductions of this instance of the ρ-calculus.

The conditions imposed for the generic strategy when the term t is not a first order ground term are clearly not appropriate for an implementation of the ρ-calculus and thus, we must define operational strategies guaranteeing the confluence of the calculus. These strategies will impose some decidable conditions that correspond to (and imply) the ones proposed above.

We introduce in what follows a more operational and more restrictive strategy definition guaranteeing the matching *"coherence"* by imposing structural conditions on the terms l and t involved in a matching problem $l \ll^?_\emptyset t$. In order to ensure the matching failure preservation by the ρ-reductions, the failure must be generated only by different first order symbols in the corresponding positions of the two terms l and t. This property is always verified if the two terms are first order terms but an additional condition must be imposed if the term t contains ρ-calculus specific operators, as the abstraction or the application.

Definition 2.4.2 A ρ-term l *weakly subsumes* a ρ-term t if

$$\forall p \in \mathcal{FP}os(l) \cap \mathcal{P}os(t) \Rightarrow t(p) \in \mathcal{F}$$

Thus, a ρ-term l *weakly subsumes* a ρ-term t if for any functional position of the term l, either this position is not a position of the term t, or it is a functional position of the term t.

Remark 2.4.1 If $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ weakly subsumes t then, for any non-functional position (*i.e.* the position of a variable, an application, an abstraction or a set) in t, the corresponding position in l, if it exists, is a variable position. Thus, if the top position of t is not a functional position, then l is a variable.
One can notice that if a first order term l subsumes t, then l weakly subsumes t.

Example 2.4.6 The term $h(a, y, c)$ weakly subsumes the term $g(b, [x \to x](c))$ and the term $f(a)$ weakly subsumes the term $g(b, [x \to x](c))$. The term $g(a, y)$ weakly subsumes the term $g(b, [x \to x](c))$ while the term $f(a)$ does not weakly subsumes $f([x \to x](c))$.

Definition 2.4.3 We call *ConfStrat* the strategy which consists in applying the evaluation rule $Fire$ to a redex $[l \to r](t)$ only if:

- $t \in \mathcal{T}(\mathcal{F})$ is a first order ground term

or

- the term $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is linear and l weakly subsumes t, and
- the term t contains no set with more than one element and no empty set, and
- for all sub-term $[u \to w](v)$ of t, u subsumes v, and
- the term t contains no sub-term of the form $[u](v)$ where u is not an abstraction.

One should notice that the conditions imposed by the strategy *ConfStrat* are decidable even if the term t is not a first order ground term. One can clearly decide if a term is of the form $[u](v)$ or $[u \to w](v)$ as well as the number of elements of a finite set. The condition that l weakly subsumes t is simply a condition on the symbols on the same positions of the two terms and since matching is syntactic, then the subsumption condition is also decidable. Consequently, all the conditions used in the strategy *ConfStrat* are decidable.

The condition forbidding sub-terms of t of the form $[u](v)$ if u is not a rewrite rule is imposed in order to prevent the application of the evaluation rule $CongFail$ leading to an empty set result. If one considers a version of the ρ-calculus without the evaluation rules *Congruence* then, this last condition is no longer necessary in the strategy *ConfStrat*. Hence, all the terms of the representation of the λ-calculus in the ρ-calculus trivially satisfy the above conditions and in this case the strategy *ConfStrat* is equivalent to the strategy $\mathcal{ALL}$.

Proposition 2.4.2 When using the evaluation strategy *ConfStrat*, the ρ-calculus is confluent.

Starting from the evaluation rule $Fire$ expressed as a conditional rule guarded by the conditions defined in the strategy *ConfStrat* we define the relation $FireCong$ induced by this latter rule and the *Congruence* rules. The other evaluation rules of the calculus induce a second relation called Set.

We denote by $\longrightarrow_F$ and $\longrightarrow_S$ respectively, the compatible (context) closures of these two relations, and by $\xrightarrow{*}_S$ the reflexive and transitive closure of $\longrightarrow_S$.

We prove the confluence of the relation $\xrightarrow{*}_S \longrightarrow_F \xrightarrow{*}_S$ and we use an approach similar to the one followed in [CHL96] for proving the confluence of $\lambda_{\Uparrow}$.

Thus, we have to prove the strong confluence of the relation $\longrightarrow_F$, the confluence and termination of $\longrightarrow_S$ and the compatibility between the two relations (*i.e.* Yokouchi's Lemma.

Using a polynomial interpretation we show that $\longrightarrow_S$ terminates and by analyzing the induced critical pairs we obtain the local confluence and consequently, the confluence of this relation.

The relation $\longrightarrow_F$ is not strongly confluent but we define the parallel version of this relation in the style of *Tait & Martin-Löf*. We denote this relation by $\longrightarrow_{F_\parallel}$ and we show that is strongly confluent.

The Yokouchi Lemma is proved using the conditions imposed on the application of the rule $Fire$. We obtain thus the strong confluence of the relation $\xrightarrow{*}_S \longrightarrow_{F_\parallel} \xrightarrow{*}_S$ and since this latter relation is the transitive closure of the relation $\xrightarrow{*}_S \longrightarrow_F \xrightarrow{*}_S$ we deduce the confluence of the calculus.

The proof is presented in full detail in [Cir00].

The relatively restrictive conditions imposed in strategy *ConfStrat* can be relaxed at the price of the simplicity of the strategy. The conditions that we want to weaken concern on one hand, the number of elements of the sets and on the other hand, the form of the rewrite rules.

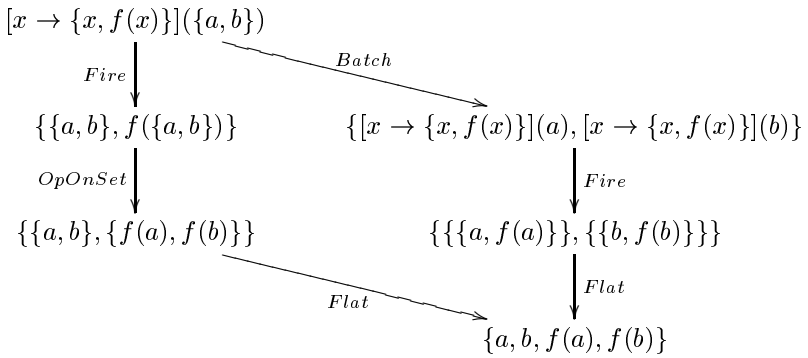
First, the absence of sets having more than one element is necessary in order to guarantee a good behavior for the non-right-linear rewrite rules. The *right-linearity* of a rewrite rule is defined as the linearity of the right-hand side w.r.t. the variables of the left-hand side. For example, $x \to g(x, y)$ is right-linear, but

$x \to g(x,x)$ is not right-linear. Moreover, the right-linearity can be imposed only to the operators different from the set symbols ($\{\_\}$) and thus, the rewrite rule $x \to \{f(x), f(x)\}$ can be considered right-linear. Intuitively, we do not need to impose right-linearity for sets since, due to the evaluation rule $Flat$, they do not lead to non-convergent reductions as in Example 2.4.5 on page 51.

Definition 2.4.4 The rewrite rule $l \to r$ is *hereditary right-linear* if any sub-term of r that is not a set is linear w.r.t. the free variables of l and any rewrite rule of r is hereditary right-linear.

The application of a rewrite rule which is not hereditary right-linear to a set with more than one element can lead to non-convergent reductions, as shown in Example 2.4.5 on page 51, but this is not the case if the applied rewrite rule is hereditary right-linear:

Example 2.4.7



On another hand, in order to guarantee the strict propagation of the failure, we impose that the evaluation rule $Fire$ is applied only if the argument of the application is not an empty set and it cannot lead to an empty set. In Example 2.4.4 on page 51 we can notice that the free variables of the left-hand side of the rewrite rule are not preserved in the right-hand side of the rule. If the rewrite rule $l \to r$ of the application preserves the variables of the left-hand side in the right-hand side (*e.g.* $x \to x$), the application of a substitution replacing one of these variables with an empty set (*e.g.* $\{x/\emptyset\}$) to r leads to a term containing $\emptyset$ and thus, which is possibly reduced to $\emptyset$.

We define thereafter more formally the rewrite rules preserving the variables and we present a new strategy defined using this property. First, we introduce a concept similar to that of free variable but, by considering this time the not-deterministic nature of the sets.

Definition 2.4.5 The set of *present variables* of a ρ-term t is denoted by $PV(t)$ and is defined by:

1. if $t = x$ then $PV(t) = \{x\}$,
2. if $t = \{u_1, \ldots, u_n\}$ then $PV(t) = \bigcap_{i=1\ldots n} PV(u_i)$, $(PV(\emptyset) = \mathcal{X})$,
3. if $t = f(u_1, \ldots, u_n)$ then $PV(t) = \bigcup_{i=1\ldots n} PV(u_i)$, $(PV(c) = \emptyset \; if \; c \in \mathcal{T}(\mathcal{F}))$,
4. if $t = [u](v)$ then $PV(t) = PV(u) \cup PV(v)$,
5. if $t = u \to v$ then $PV(t) = PV(v) \setminus FV(u)$.

The set of *free variables* of a set of ρ-terms is the union of the sets of free variables of each ρ-term while the set of *present variables* of a set of ρ-terms is the intersection of the sets of free variables of each ρ-term. We can say that a variable is *present* in a set only if it is present in all the elements of the set. For example, $PV(\{x,y,x\}) = \emptyset$ and $PV(\{x, g(x,y)\}) = \{x\}$.

Definition 2.4.6 We say that the ρ-rewrite rule $l \to r$ is quasi-regular if $FV(l) \subseteq PV(r)$ and any rewrite rule of r is quasi-regular.

Intuitively, to each free variable of the left-hand side of a quasi-regular rewrite rule corresponds, in a deterministic way, a free variable in the right-hand side of the rule. For any set ρ-term in the right-hand side, the correspondence with the free variables of the left-hand side should be verified for each element of the set.

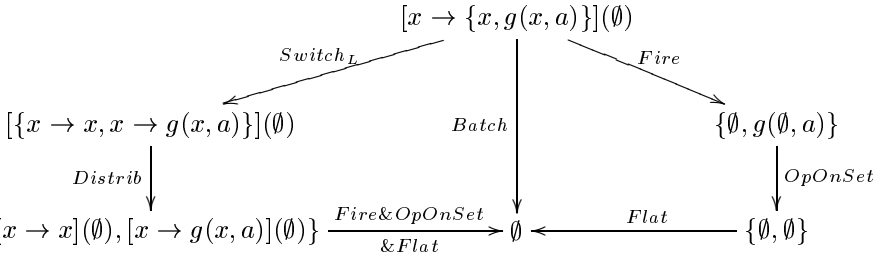
Example 2.4.8 The rewrite rule $x \to g(x, y)$ is quasi-regular while the rewrite rule $x \to \{x, y\}$ is non-quasi-regular.

The rewrite rule $\{f(x), g(x, x)\} \to x$ is quasi-regular while $\{f(x), g(x, y)\} \to x$ is non-quasi-regular. If the definition of quasi-regular rewrite rules had asked for the condition $PV(l) \subseteq PV(t)$ instead, then the second rewrite rule would have become quasi-regular as well. This is not desirable since the rewrite rule $\{f(x), g(x, y)\} \to x$ reduces to $\{f(x) \to x, g(x, y) \to x\}$ and only the first one is quasi-regular.

In the particular case of the ρ-calculus, since the left-hand side of a rewrite rule $l \to r$ must be a first-order term (*i.e.* $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$), we have $FV(l) = PV(l) = \mathcal{V}ar\, l$ and thus the condition from Definition 2.4.6 on the page before can be changed to $\mathcal{V}ar\, l \subseteq PV(t)$.

Let us consider the application a quasi-regular rewrite rule $l \to r$ to a term t giving as result the term $\{\sigma r\}$, where σ is the matching substitution between l and t. If $\emptyset$ is a sub-term of t and if l weakly subsumes r, then $\emptyset$ is in σ. Since the rewrite rule is quasi-regular, we have $\mathcal{D}om(\sigma) \subseteq PV(r)$ and thus, we are sure that $\emptyset$ is a sub-term of σr. Furthermore, if $\emptyset$ instantiated a variable of a set in σr then it is present in all the elements of the set and thus, we avoid non-confluent results as the ones in Example 2.4.4 on page 51.

Example 2.4.9 A quasi-regular rule applied to $\emptyset$ gives only one result:



while a non-quasi-regular one yields two different results as shown in Example 2.4.4 on page 51.

One should notice that if a rewrite rule $l \to r$ is reduced by the evaluation rule $Switch\_R$ to a set of rewrite rules, each of these rules is quasi-regular and thus the strict propagation of the empty set is ensured on all the right-hand sides of the obtained rewrite rules.

Definition 2.4.7 We call *ConfStratLin* the strategy which consists in applying the evaluation rule $Fire$ to a redex $[l \to r](t)$ only if $t \in \mathcal{T}(\mathcal{F})$ is a first order ground term *or*:

- the term $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is linear and l weakly subsumes t,

  and

- *either*
  - $l \to r$ is quasi-regular

    or
  - the term t contains no empty set, and
  - for all sub-term $[u \to w](v)$ of t, u subsumes v, and
  - the term t contains no sub-term of the form $[u](v)$ where u is not an abstraction.

  and

- *either*

- $l \to r$ is hereditary right-linear

or

- the term t contains no set with more than one element.

Compared to the strategy *ConfStrat* we added the possibility to test either the quasi-regular condition on the rewrite rule $l \to r$ or the conditions on the reducibility of the term t to an empty set. Moreover, if the rewrite rule is hereditary right-linear we allow arguments containing sets having more than one element. Since one can clearly decide if a rule is quasi-regular or hereditary right-linear, all the conditions used in the strategy *ConfStratLin* are decidable.

Proposition 2.4.3 When using the evaluation strategy *ConfStratLin*, the ρ-calculus is confluent.

The same approach as for the strategy *ConfStrat* is used but some additional diagrams corresponding to the reductions that where not possible before are considered. These new cases are mainly introduced in the proof of Yokouchis's Lemma. The proof is detailled in [Cir00].

When using a calculus integrating reduction modulo an equational theory (*e.g.* associativity and commutativity), as explained in Section 2.2.4, the overall confluence proof is different but uses lemmas similar to the ones of the former case. Therefore, we conjecture that Proposition 2.4.2 on page 54 and Proposition 2.4.3 can be extended to a ρ_E-calculus modulo a specific decidable and finitary equational matching theory E.

2.5 Conclusion

We have presented the ρ_T-calculus together with some of its variants obtained as instances of the general framework. By making explicit the notion of rule, rule application and application result, the ρ_T-calculus allows us to describe in a simple yet very powerful and uniform manner algebraic and higher-order capabilities. This provides therefore a simple and natural framework for their combination.

In the ρ_T-calculus the non-determinism is handled by using sets of results and the rule application failure is represented by the empty set. Handling sets is a delicate problem and we have seen that the raw ρ-calculus, where the evaluation rules are not guided by a strategy, is not confluent. When an appropriate but rather natural generalized call-by-value evaluation strategy is used, the calculus is confluent.

The ρ-calculus is both conceptually simple as well as quite expressive. This allows us to represent the terms and reductions from λ-calculus and rewriting. We conjecture that, following the lines of [Vir96], it is also simple to encode other calculi of interest like the π-calculus.

Chapter 3

Compilation of rule-based programs

H. Kirchner and P.-E. Moreau [KM01]

3.1 Introduction

Rewrite rules are pairs of terms (the left and right-hand sides) with variables, that describe a transformation on given expressions. A rewrite rule may be guarded by a condition which is a boolean expression. A rewrite system is a set of rewrite rules. Rewrite rules are frequent in many areas of Computer Science, but languages based on rewriting are not so common. Let us cite for instance the first-order languages OBJ [GW88], ASF+SDF [Kli93], Maude [CELM96], Cafe-OBJ [FN97] and ELAN [BKK+98b]. In these languages, programs are sets of rewrite rules, called rewrite programs, and a query is an expression to evaluate according to these rules. Evaluation is performed by applying rewrite rules. Informally, rewriting an expression consists of selecting a rule whose left-hand side (also called pattern) matches the current expression, or a sub-expression, computing a substitution that gives the values of rule variables, checking that the condition evaluates to true under this substitution, and applying it to the right-hand side of the selected rule to build the reduced term. In general the evaluation may not terminate, or terminate with different results according to which rules are applied. So evaluation by rewriting is essentially non-deterministic and backtracking is used to generate all results. In this chapter, we address compilation techniques for languages based on rewriting.

First-order languages based on rewrite rules share many features with functional languages [BW88] such as CAML [CM98, WL93], Clean [BvEvL+87, PvE93], Erlang [AVWM96, Arm97], Gofer [Jon94], Haskell [Jon96, PJ96], or ML [CPH+85, LM93]. They provide the same capability of writing specifications which can be actually executed, tested and debugged. Such a specification is the first prototype of the final program. Re-usability is encouraged through language features such as modules, polymorphism, algebraic types and predefined types. Both classes of languages share concepts like pattern matching (first-order versus higher-order), (tree or graph) rewriting, guards (or conditions), sometimes "where" blocks and "let" expressions. All these programs tend to be concise, easy to understand, and relatively easy to maintain because the code is short, clear, with no side effects or unforeseen interactions. They are strongly typed, eliminating a huge class of errors at compile time. In such languages, the programmer is relieved of the storage management burden. Store is allocated and initialised implicitly, and reclaimed by the garbage collector. Compared to imperative languages, programs are easier to design, write and maintain, but the language offers the programmer less control over the machine.

However, contrary to functional languages, λ-abstraction and higher-order matching are not used in first-order languages based on rewrite rules. Higher-order functions provide a powerful abstraction mechanism, since a function can be freely passed to other functions, returned as a result of a function, stored in a data structure and so-on. This is not possible in most rewriting based languages. However, we will present later the ELAN system, where the notion of strategy is indeed similar to an higher-order function applied with an explicit application operator. Moreover, although no λ-expression can be written in ELAN, there is an

extension of the language, based on ρ-calculus [CK99], which provides a uniform integration of λ-calculus and first-order rewriting.

The loss of abstraction due to the first-order restriction is balanced by the ability to build equational theories in the matching and rewriting mechanism. In order to illustrate how expressivity and conciseness are gained with this approach, let us consider the formalisation of sorting a list of elements by applying only one rule:

$$L_1 \cdot x \cdot L_2 \cdot y \cdot L_3 \to L_1 \cdot y \cdot L_2 \cdot x \cdot L_3 \text{ if } x > y$$

where L_1, L_2, L_3 are variables of type $List[Element]$, x, y are variables of sort $Element$ and where the "$\cdot$" list concatenation operator is associative and has a unit element which is the empty list nil. Thanks to these built-in properties, the matching algorithm applied to the list $(3 \cdot 5 \cdot 1)$ will discover several solutions, among which for instance: $L_1 = nil, x = 3, L_2 = nil, y = 5, L_3 = (1)$ and $L_1 = (3), x = 5, L_2 = nil, y = 1, L_3 = nil$. Since the first match does not satisfy the condition $x > y$, the second has to be chosen to apply the rewrite rule and to get the list $(3 \cdot 1 \cdot 5)$. A second rule application yields the now irreducible result $(1 \cdot 3 \cdot 5)$.

Beyond lists, other data structures that can benefit from this kind of rewriting are sets and multisets, where the union operator is associative and commutative (AC for short). AC operators provide a high level of abstraction: the programmer may use set data structures without knowing how they are represented. Moreover, their implementation may be more efficient than a list implementation, simply because some optimisations are performed at compile time. In order to illustrate the practical interest of AC operators, let us consider the well-known N-queens problem, that will be further developed in Section 3.5. The problem consists of setting a queen on each row of the chessboard and assigning a unique column number (taken in the set of integers $\{1, \ldots, N\}$) such that the queens do not attack each other. We represent a solution by a list of integers and we use a set to represent columns that may be assigned to a queen. As we will see in Example 3.5.1 on page 74, the program may be expressed in ELAN with three rules and a strategy. The first of these three rules is the following one, where i and S are variables of respective types *Integer* and *Set*:

$$(i) \cup S \to [i, S]$$

The $\_ \cup \_$ operator (where $\_$ stands for a place-holder) is an AC infix operator, which allows expressing constructions such as $(1) \cup (2) \cup (3)$. The $[\_, \_]$ operator is a constructor used to represent pairs for example. When this rule is applied on a given term, say $(1) \cup (2) \cup (3)$, the first result is $[1, (2) \cup (3)]$, when the variable i is instantiated to 1 (i.e. (i) matches (1)). Another possibility is to match i with 2, and in this case, the result is $[2, (1) \cup (3)]$. Thus, the application of an AC rule returns a multiset of results, and behaves like a *generator* over a set data structure. This feature will be used in Example 3.5.1 on page 74 to enumerate all possibilities to set a queen on a chessboard.

Let us explain in more detail the properties of first-order languages based on rewrite rules.

The evaluation mechanism relies primarily on a matching algorithm that must be carefully designed. When programming with rewrite rules, it frequently happens that programs contain several rules whose left-hand sides begin with the same top function symbol. This often corresponds to a case definition of a function, as usual in functional programming languages. In this context, many-to-one pattern matching is quite relevant. The idea is to group all these rules together and search for a candidate rewrite rule in this group when the subject to be reduced begins with the same top function symbol.

When the program involves algebraic structures with axioms stating commutativity of function symbols, these axioms cannot be oriented as a terminating rewrite system. As mentioned above, they instead can be handled implicitly by working with congruence classes of terms. For practical implementation purposes, representatives of these congruence classes are chosen, and the matching step of term rewriting is performed by special matching algorithms, specific to the equational theories in use. An important case in practice is the case of associative and commutative theories. However, AC matching has a high computational complexity, as analysed in [BKN87, HK95]. Moreover, since an AC matching problem may have several minimal solutions, if the first match which is found does not satisfy the condition of the rule, or if all possibilities are looked for, backtracking is used to get the next solutions. With this additional non-determinism, rewriting in such theories becomes computationally difficult and it is a real challenge to identify subclasses of programs for which it is possible to provide an efficient compiler for the language.

Although it is essential to have a good matching algorithm to get an efficient rewriting engine, matching is not the only operation involved in a normalisation process. To compute the reduced term, a global

consideration of the whole process is crucial: all data structures and sub-algorithms have to be well-designed to cooperate without introducing any bottleneck. Optimisations go through the careful combination of several algorithms and data structures. In this chapter, we show how we have reused and improved existing techniques, and invented new ones, in order to build a compiler for AC rewriting.

As already mentioned, there are different sources of non-determinism in the evaluation process. If the rewrite system is not confluent, different normal forms, when they exist, may be considered as relevant results of the computation. In presence of AC function symbols, several matching solutions have to be considered and lead to different results. We explain in this chapter how to use strategy constructors to handle sets of results, how to perform a determinism analysis at compile time and the benefits of this analysis for the performance of the compiled evaluation process.

This chapter is an extension of previous work published in [MK98, KM98]. After a short introduction of notations and classical definitions in Section 3.2, an algorithm for many-to-one AC matching is presented in Section 3.3. This algorithm works efficiently for a restricted class of patterns, and other patterns are transformed to fit into this class. A refined compact bipartite graph data structure allows encoding all matching problems relative to a set of rewrite rules. A few optimisations concerning the construction of the substitution and of the reduced term are described in Section 3.4. In Section 3.5, we turn to the problem of non-determinism and show how to handle it through the concept of strategies. All these sections are independent from the rewriting language and the described techniques may be useful to build a compiler for any language based on non-deterministic rewriting in associative and commutative theories. Section 3.6 briefly introduces the ELAN system and its compiler. The conclusion in Section 3.7 points out a few directions where there seems to be yet some room for further improvements.

3.2 Preliminary concepts

We assume the reader familiar with basic definitions of term rewriting given in particular in [DJ90b, BN98], and associative commutative theories handled for instance in [PS81, JK86]. We briefly recall or introduce notations for a few concepts that will be used along this chapter.

$\mathcal{T}(\mathcal{F},\mathcal{X})$ is the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables, denoted $x, y, z, \ldots$. Positions in a term are represented as sequences of integers and denoted by greek letters ϵ, ν. The empty sequence ϵ denotes the position associated to the root and so it is the position of the top symbol. The subterm of t at position ν is denoted $t_{|\nu}$. The replacement at position ν of the subterm $t_{|\nu}$ by t' is written $t[\nu \hookleftarrow t']$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms. A term t is said to be *linear* if no variable occurs more than once in t. A *substitution* is an assignment from a finite subset of $\mathcal{X}$ to $\mathcal{T}(\mathcal{F},\mathcal{X})$, written $\sigma = \{y_1 \mapsto t_1, \ldots, y_k \mapsto t_k\}$.

In a first part of this chapter, we focus on theories and rewrite systems in which there is at least one binary function symbol F, that satisfies the following set AC of associativity and commutativity axioms:

$$\forall x, y, z,\ F(x, F(y, z)) = F(F(x, y), z) \quad \text{and} \quad \forall x, y,\ F(x, y) = F(y, x).$$

Such symbols are called AC function symbols and are always in uppercase in the following. On the other hand, $\mathcal{F}_\emptyset$ is the subset of $\mathcal{F}$ made of function symbols which are not AC, and are called *free* function symbols. In the following, we consider that a function symbol is either free or AC. A term is said to be *syntactic* if it contains only free function symbols. We write $s =_{AC} t$ to indicate that the two terms s and t are equivalent modulo associativity and commutativity.

Definition 3.2.1 *A rewrite rule is a pair of terms denoted $l \to r$ such that $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The term l is called the left-hand side or pattern and r is the right-hand side.*

In order to get a better control on the application of the rewrite rules, conditions can be added. In this chapter, we consider an enriched notion of condition, called *matching condition*, as used for instance in ASF+SDF and ELAN.

Definition 3.2.2 *A* conditional rewrite rule *denoted* $l \to r$ ***where*** $p := c$ *is such that* $l, r, p, c \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\mathcal{V}ar(p) \cap \mathcal{V}ar(l) = \emptyset$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p)$ *and* $\mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$. *When the term* p *is just the boolean constant true, the condition is usually written* ***if*** c.

The notion of conditional rewrite rule can be generalised with a sequence of conditions, as in $l \to r$ **where** $p_1 := c_1 \ldots$ **where** $p_n := c_n$ where:

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$ and
- $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$.

A (conditional) rewrite rule is said to be *syntactic* if the left-hand side is a syntactic term.

To apply a syntactic rule $l \to r$ on a term t at some position ν, one looks for a matching, i.e. a substitution σ satisfying $l\sigma = t_{|\nu}$. Note that t is always a ground term. The algorithm which provides the unique substitution σ, whenever it exists, is called *syntactic matching*. Once a substitution σ is found, the application of the rewrite rule consists of building the *reduced term* $t' = t[\nu \leftarrow r\sigma]$. Computing the normal form of a term t w.r.t. a rewrite system R consists of successively applying the rewrite rules of R, at any position, until no more applies. The existence and uniqueness of normal forms require the rewrite system R to be respectively terminating and confluent.

To apply a syntactic conditional rule $l \to r$ **where** $p := c$ on a term t, the satisfiability of the condition **where** $p := c$ has to be checked before building the reduced term. Let σ be the matching substitution from l to $t_{|\nu}$. Checking the matching condition **where** $p := c$ consists first of using the rewrite system R to compute a normal form c' of $c\sigma$, whenever it exists, and then verifying that p matches the ground term c'. If there exists a matching μ, such that $p\mu = c'$, the composed substitution $\sigma\mu$ is used to build the reduced term $t' = t[\nu \leftarrow r\sigma\mu]$. Otherwise the application of the conditional rule fails. For usual boolean conditions of the form **if** c, μ is the identity when the normal form of c is *true*. In cases where $c\sigma$ has no normal form, the application of the rule does not terminate.

When the conditional rule is of the form $l \to r$ **where** $p_1 := c_1 \ldots$ **where** $p_n := c_n$, the matching substitution is successively composed with each matching μ_i from p_i to a normal form of $c_i\sigma\mu_1 \ldots \mu_{i-1}$, for $i = 1, \ldots, n$, when it exists. If one of these μ_i does not exist, the application of the conditional rule fails. If one of the $c_i\sigma\mu_1 \ldots \mu_{i-1}$ for $i = 1, \ldots, n$ has no normal form, the application of the rule does not terminate.

When the left-hand side of the (conditional) rule contains AC function symbols, AC matching is invoked from l to $t_{|\nu}$. The term l is said to AC match the term $t_{|\nu}$ if there exists a substitution σ such that $l\sigma =_{AC} t_{|\nu}$. In general, AC matching can return several solutions, which introduces a need for backtracking for conditional rules: as long as there is a solution to the AC matching problem for which the matching condition is not satisfied, another solution has to be extracted. Similarly, if the pattern p contains AC function symbols, an AC matching procedure is called. Only when all solutions have been tried unsuccessfully, the application of this conditional rule fails. When the rule contains a sequence of matching conditions, failing to satisfy the i-th condition causes a backtracking to the previous one.

So, in our case, conditional rewriting requires AC matching problems to be solved in a particular way: the first solution has to be found as fast as possible, and the others have to be provided "on request". AC matching has already been extensively studied, for instance in [Hul80, BKN87, KL91, BCR93, LM94, Eke95]. In this chapter, we borrow from these works some techniques for the compilation of AC-matching, but we go further and address the more global problem of its integration in a normalisation procedure.

One of the first problems encountered for an efficient implementation is to choose an adequate term representation. It is well known that terms involving associative function symbols can be converted to a normal form by grouping associative operators to the right (or left). Equivalently, and more usefully for machine representation, we may flatten such terms by replacing nested occurrences of the same associative operator by a single variadic operator (i.e. an operator with a variable arity). The same may be done with AC operators. However this does not give a unique normal form, since the commutativity axiom may be used to arbitrarily permute the arguments of a variadic operator. Nevertheless, we obtain a unique normal form by regarding the arguments of a variadic operator as a multiset of terms (since the same term may

occur as an argument more than once). A canonical form [Hul80, Eke95] corresponds to an implementation of this idea where a unique syntactic representation of this multiset of arguments is obtained by sorting and grouping. Canonical form computation can be seen as a function $\mathcal{CF}$ on $\mathcal{T}(\mathcal{F},\mathcal{X})$ that returns, for each term t, a unique representative of its congruence class. Let $>$ be a total ordering on the set of symbols $\mathcal{F}\cup\mathcal{X}$.[1] For terms that are just variables or constants, $\mathcal{CF}$ is the identity function and the total ordering is the given ordering $>$ on symbols. For terms in canonical form with different top symbols, the total ordering is given by the ordering on their top symbols. This ordering on canonical forms is also denoted $>$. Considering that AC function symbols can be variadic, the canonical form is obtained by flattening nested occurrences of the same AC function symbol, recursively computing the canonical forms and sorting the subterms, and replacing α identical subterms by a single instance of the subterm with multiplicity α, denoted by t^α. A formal definition can be found in [Eke95]. An algorithm that performs a bottom-up computation of $\mathcal{CF}$ is given later on, in Section 3.4.1. At this stage, canonical form computation is easily understandable by an example. Consider the term $t = F(F(t_1,t_2),\ t_3,\ F(t_4,t_5))$ where no t_i has the AC function symbol F as top symbol. Assuming that $\mathcal{CF}(t_1) = \mathcal{CF}(t_5)$, $\mathcal{CF}(t_3) = \mathcal{CF}(t_4)$ and $\mathcal{CF}(t_1) > \mathcal{CF}(t_2) > \mathcal{CF}(t_3)$, then $\mathcal{CF}(t) = F(\mathcal{CF}(t_1)^2, \mathcal{CF}(t_2), \mathcal{CF}(t_3)^2)$.

A term in canonical form is said to be *almost linear* if the term obtained by forgetting the multiplicities of variable subterms is linear. For instance, the term $t = F(x^3, y^2, g(a))$ is almost linear.

For a term t in canonical form, the *syntactic top layer* $\hat{t}$ is obtained from t by removing subterms below the first AC symbol in each branch and considering remaining AC symbols as constants. For instance the top layer of the term $f(g(a), F(f(a,x), f(y,g(b))))$ is $f(g(a), F)$, if $f,g,a,b \in \mathcal{F}_\emptyset$. Note that $f(g(a),F)$ is a syntactic term if we consider F as a constant. A formal definition of the top layer and a few properties can be found in [BCR93, Mor99].

Another theoretical difficulty related to AC rewriting is to ensure its completeness with respect to equality in congruence classes: given a rewrite system R, how to ensure that for any terms t and t', t and t' are equivalent in the theory defined by R and AC, if and only if t and t' AC rewrite respectively to terms u and u' such that $u =_{AC} u'$. The interested reader can refer to [JK86] which provides an extensive study of this question. For the purpose of this chapter, it is enough to know that to achieve this completeness property, we must already ensure the following property of coherence with AC congruence classes: if a term is reducible, any AC equivalent term is reducible too. In order to illustrate the problem and its solution, let us consider the associative-commutative union operator: $\cup$, and the rewrite rule that removes identical elements of a multiset:

$$x \cup y \to x \textbf{ if } x = y$$

When applied on $a \cup a$, the result is a, but when applied on $(a \cup b) \cup (a \cup c)$, the result is not $a \cup b \cup c$ because the rule cannot be applied directly to $(a \cup a)$. To perform the expected rewrite step on subterms of equivalent terms such as the subterm $(a \cup a)$ of $(a \cup a) \cup (b \cup c)$ in this example, a new rule with an extension variable z' has to be added:

$$z' \cup (x \cup y) \to z' \cup x \textbf{ if } x = y$$

The variable z' matches the context and allow to perform rewrite steps in subterms. In order to get this coherence property of AC rewriting, some rules called extensions are automatically added. The extension of the rule $F(l_1,\ldots,l_n) \to r$, already put in flattened form, where F is an AC function symbol, is of the form $F(z', l_1,\ldots,l_n) \to F(z', r)$, where z' is called an extension variable. This well-known technique was introduced in [PS81] and generalised in [JK86] where a more detailed justification of these extensions can be found. Extensions are not needed for rules of the form $F(l_1,\ldots,l_n) \to r$, where F is AC, and where one of the $l_1,\ldots,l_n$ is a variable with multiplicity 1 which does not occur also in the condition of the rule. Indeed this variable can capture the context as well as an extension variable. Note that an extension is always necessary for rules where $l_1,\ldots,l_n$ are non-variable subterms. In addition, from the implementation point of view, the reduction with a rule $F(l_1,\ldots,l_n) \to r$ can be simulated with $F(z', l_1,\ldots,l_n) \to F(z', r)$ only, by allowing the extension variable z' to be instantiated to an empty context.

Example 3.2.1 *In order to support intuition throughout this chapter, we choose the following running example of two rewrite rules with the same AC top symbol F and whose right-hand sides are irrelevant. The*

[1] Any ordering can be chosen but, once chosen, it is fixed, since it determines the uniqueness of representation.

first rule has a boolean condition ***if*** $z = x$ *which is equivalent to the matching condition* ***where*** $true := (z = x)$ *and we assume that the boolean function* $=$ *is completely defined by rewrite rules.*

$$\begin{array}{lcll} F(z, f(a,x), g(a)) & \to & r_1 & \textbf{\textit{if}} \quad z = x \\ F(f(a,x), f(y, g(b))) & \to & r_2 & \end{array}$$

Their respective extensions are:

$$\begin{array}{lcll} F(z', z, f(a,x), g(a)) & \to & F(z', r_1) & \textbf{\textit{if}} \quad z = x \\ F(z', f(a,x), f(y, g(b))) & \to & F(z', r_2) & \end{array}$$

Both extensions are necessary and for reduction, only extensions are sufficient. In all following examples related to these four rules, we assume that x, y, z, z' *are variables,* f, g, a, b *are free function symbols, and* F, G *are AC function symbols. We also assume that the following total order is used to compute canonical forms:*

$$z' > x > y > z > f > g > a > b$$

3.3 Many-to-one AC matching

An AC matching problem is said to be *one-to-one* when only one pattern p and only one subject s are involved: the problem consists of finding substitutions σ such that $p\sigma =_{AC} s$. By extension, an AC many-to-one matching is the following problem: given a set of terms $P = \{p_1, \ldots, p_n\}$, called patterns, and a ground term s, called subject, find one (or more) patterns in P that AC-matches s. Patterns and subject are assumed to be in canonical form. Efficient many-to-one matching algorithms (both in the syntactic case and in AC theories) are based on the general idea of factoring patterns to produce a matching automaton. The discrimination net approach [Grä91, McC92, Chr93, Vor95, NWE97, Mor99] is one of this kind: it is a variant of the data structure used to index dictionaries. Given a set of patterns, the idea is to partition terms based upon their structure. As presented in [Chr93], a tree is formed and at each point where two terms have different symbols, a separate branch for each term is added.

Figure 3.1 shows a discrimination net associated to the set of patterns $P = \{f(a,x), f(y,g(b)), g(a)\}$. At the end of each path in the tree, there is a list of terms sharing the same structure. Since at this stage of the matching algorithm, we are dealing only with linearized terms, all variables are different and may be treated as a single wildcard symbol ω. A discrimination net is said to be deterministic when no backtracking is needed to compute the maximal set of terms that match a given subject. To build such a discrimination net, several algorithms exist and are fully detailed in [Grä91, Chr93, NWE97, Mor99]. In our implementation we use the algorithm presented in [Mor99] which is incremental and produces compact discrimination nets (i.e. with a reduced number of nodes).

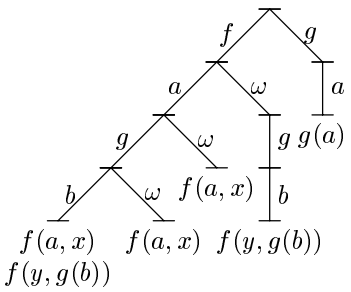


Figure 3.1: Deterministic discrimination net associated to $f(a,x)$, $f(y,g(b))$ and $g(a)$

Given a deterministic discrimination net D, a matching automaton A is associated: the nodes of D are the states of A, the root being the initial state and the leaves being the final states. On a given input term t, the automaton scans t in preorder and makes a transition $u \to v$ if D contains an edge (u, v) labelled by the current symbol of t or by ω. In the latter case, the current subterm of t is skipped and used to

build the substitution. Given the ground term $f(g(c), g(b))$, the matching automaton associated to the previous discrimination net recognises the pattern $f(y, g(b))$ (following the path $f \to \omega \to g \to b$) where y is instantiated to the subterm $g(c)$.

In the case of AC theories, the matching problems are decomposed according to the syntactic top layers and the different AC symbols occurring in the patterns. This decomposition gives rise to a hierarchically structured collection of standard discrimination nets, called an AC discrimination net [BCR93, Gra96]. Given a set of patterns $P_i = P = \{p_1, \ldots, p_n\}$, the construction of such a structure is performed in four steps:

1. computation of the syntactic top layer $\hat{P}_i = \{\hat{p_{i_1}}, \ldots, \hat{p_{i_n}}\}$;
2. construction of the matching automaton A_i associated to $\hat{P}_i$ (where all AC symbols are considered as constants);
3. recursive application of the algorithm to P_{i+1} (the set of "removed" terms during the computation of $\hat{P}_i$);
4. construction of a special edge between AC symbols appearing in A_i and the top automaton A_{i+1} associated to the sub-AC matching structure built during the recursive application of the algorithm.

Let us consider again the set of patterns in our running example, where F is the unique AC function symbol:

$$P_1 = P = \{F(z', z, f(a, x), g(a)), F(z', f(a, x), f(y, g(b)))\}$$

We have $\hat{P}_1 = \{F, F\}$ and the set of removed terms is $P_2 = \{z', z, f(a, x), f(y, g(b)), g(a)\}$. This decomposition leads us to build the AC discrimination net presented in Figure 3.2.

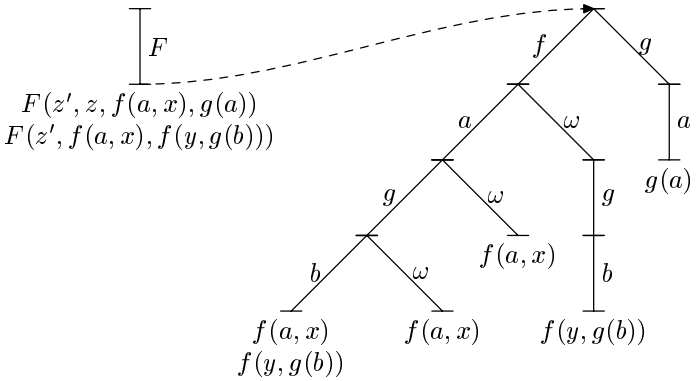


Figure 3.2: Example of AC discrimination net

The resulting net is composed of two parts:

- a discrimination net for the top layers used to determine which rules can be applied, and a link to the sub-automaton used to match subterms of AC function symbols (here F);
- the sub-automaton itself that implements a many-to-one syntactic matching algorithm [Grä91, Chr93, NWE97, Mor99] for the set of syntactic subterms: $f(a, x), f(y, g(b))$ and $g(a)$.

3.3.1 Description of the algorithm

The skeleton of our many-to-one AC matching algorithm is similar to the algorithm presented in [BCR93]. Given a set of patterns P,

1. Transform rules to fit into a specific class of patterns. In particular, patterns are converted to their canonical forms, each rule with non-linear left-hand side is transformed into a conditional rule with a linear left-hand side and a condition expressing equality between variables (of the form $x = y$).

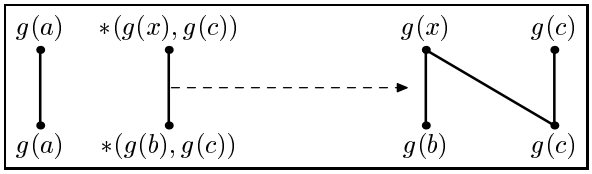


Figure 3.3: To illustrate the notion of hierarchy of bipartite graphs, we consider the following AC matching problem: a pattern $+(g(a), *(g(x), g(c)))$ and a subject $+(g(a), *(g(b), g(c)))$, where $+$ and $*$ are AC symbols, x is a variable and a, b, c, f are syntactic. Following the main algorithm, a recursive call of the AC matching algorithm is needed to build the edge between $*(g(x), g(c))$ and $*(g(b), g(c))$. This call leads to the construction of a sub-bipartite graph whose satisfiability has to be checked: if the sub-graph has a solution, the edge between $*(g(x), g(c))$ and $*(g(b), g(c))$ can be built. Solving a hierarchy of bipartite graphs means that sub-graphs are solved in a bottom-up way.

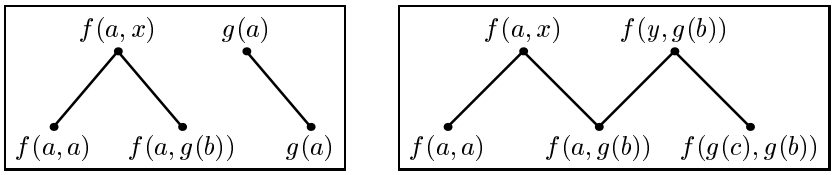


Figure 3.4: Examples of bipartite graphs

2. Compute the AC discrimination net associated to $P = \{p_1, \ldots, p_n\}$ and the corresponding matching automata (as in Figure 3.2 on the preceding page).

The previous steps only depend on the set of rewrite rules. They can be performed once and for all at compile time.

At run-time the subject $s = F(s_1, \ldots, s_p)$ is known and the matching automata are used to build bipartite graphs, where an edge between p_i and s_j is added if the subpattern p_i matches the subterm s_j. Given a ground term $s = F(s_1, \ldots, s_p)$ in canonical form, the following steps are performed.

3. Build a hierarchy of bipartite graphs according to the given subject s in canonical form. For each subterm $p_{i|_\nu}$, where ν is a position of an AC function symbol in $\hat{p_i}$, an associated bipartite graph is built. Let us consider an AC matching problem from $F(t_1, \ldots, t_m)$ to $F(s_1, \ldots, s_p)$, where $t_1, \ldots, t_m$ come from subterms of $p_{i|_\nu}$, and where for some k, $0 \leq k \leq m$, no $t_1, \ldots, t_k$ is a variable, and all $t_{k+1}, \ldots, t_m$ are variables. The associated bipartite graph is $BG = (V_1 \cup V_2, E)$ whose sets of vertices are $V_1 = \{s_1, \ldots, s_p\}$ and $V_2 = \{t_1, \ldots, t_k\}$, and whose set of edges E consists of all pairs $[s_i, t_j]$ such that $t_j\sigma$ and s_i are equal modulo AC for some substitution σ.
This construction is done recursively for each subterm of $p_{i|_\nu}$ whose root is an AC symbol. An example of recursive construction which leads to a hierarchy of bipartite graphs is given in Figure 3.3.

In the case of our running example, given the ground term $s = F(f(a, a), f(a, g(b)), f(g(c), g(b)), g(a))$, the compiled matching automaton is used to build the two bipartite graphs given in Figure 3.4 (one for each rule);

4. Find a set of solutions to the hierarchy of bipartite graphs and construct a linear Diophantine system which encodes the constraints on the remaining unbound variables (which appear directly under an AC symbol): in order to match m variables $x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}$ to n remaining subterms $s_1^{\beta_1}, \ldots, s_n^{\beta_n}$ one looks for non-negative integer solutions of the system $\bigwedge_{i=1\ldots n} \beta_i = \alpha_1 X_i^1 + \cdots + \alpha_m X_i^m$ with the additional constraint $\Sigma_{i=1}^n X_i^j \geq 1$.

5. Solve the linear Diophantine system to solutions of the form $x_k = F(s_1^{X_1^k}, \ldots, s_n^{X_n^k})$ for $k = 1 \ldots m$. This completes the definition of the matching substitution.
As an example, consider the pattern $+(x_1, x_2^3)$ and the subject $+(a^3, b^2, c^5)$ where $+$ is an AC operator.

Intuitively, this matching problem has three solutions:

$$\begin{aligned} S_1 &= \{x_1 \mapsto +(a^3, b^2, c^2), \quad x_2 \mapsto c\} \\ S_2 &= \{x_1 \mapsto +(b^2, c^5), \quad x_2 \mapsto a\} \\ S_3 &= \{x_1 \mapsto +(b^2, c^2), \quad x_2 \mapsto +(a, c)\} \end{aligned}$$

These solutions are found by solving the following linear Diophantine system:

$$\begin{aligned} \beta_1 = 3 &= 1 \times X_1^1 + 3 \times X_1^2 \\ \beta_2 = 2 &= 1 \times X_2^1 + 3 \times X_2^2 \\ \beta_3 = 5 &= \underbrace{1 \times X_3^1}_{\Sigma X_i^1 > 1} + \underbrace{3 \times X_3^2}_{\Sigma X_i^2 > 1} \end{aligned}$$

This linear Diophantine system has three solutions:

$$\begin{aligned} &X_1^1 = 3 \wedge X_1^2 = 0 \wedge X_2^1 = 2 \wedge X_2^2 = 0 \wedge X_3^1 = 2 \wedge X_3^2 = 1 \\ &X_1^1 = 0 \wedge X_1^2 = 1 \wedge X_2^1 = 2 \wedge X_2^2 = 0 \wedge X_3^1 = 5 \wedge X_3^2 = 0 \\ &X_1^1 = 0 \wedge X_1^2 = 1 \wedge X_2^1 = 2 \wedge X_2^2 = 0 \wedge X_3^1 = 2 \wedge X_3^2 = 1 \end{aligned}$$

By computing $x_1 = +(a^{X_1^1}, b^{X_2^1}, c^{X_3^1})$ and $x_2 = +(a^{X_1^2}, b^{X_2^2}, c^{X_3^2})$, the first assignment leads to the first solution:

$$S_1 = \{x_1 \mapsto +(a^3, b^2, c^2), x_2 \mapsto c\}$$

Starting from this quite general algorithm, our goal was to improve its efficiency by lowering the cost of some steps, such as traversing the levels of the hierarchy of discrimination nets, building bipartite graphs, or solving linear Diophantine systems. The idea is to apply these costly steps on specific patterns for which they can be designed efficiently, or to simply skip these steps when they are useless. We identified classes of patterns, presented in the following section, for which the general algorithm described above can be efficiently implemented. These patterns already cover a large class of rewrite programs, and for the other cases, we propose a pre-processing of the rewrite program, that transforms it into a semantically equivalent program that belongs to the restricted classes of patterns. This is explained in Section 3.3.5.

3.3.2 Classes of patterns

The classes of compiled patterns are defined on the whole set of rules together with their extensions, automatically added as described at the end of Section 3.2. All terms in the pattern classes are assumed to be in canonical form and almost linear. The pattern classes C_0, C_1, C_2, defined below, contain respectively linear terms with no AC function symbol, at most one and at most two levels of AC function symbols with a maximum of two variables rooted by an AC function symbol. The motivation to select these patterns was first based on an empirical study of rewrite rules systems used in different applications. It appeared that in practice, these restrictions are sufficiently weak to describe a large class of patterns occurring in specifications based on rewriting. On the other hand, they are sufficiently strong to allow us to design a specialised and efficient AC normalisation algorithm.

Definition 3.3.1 *Let $\mathcal{F}_\emptyset$ be the set of free function symbols, $\mathcal{F}_{AC}$ the set of AC function symbols and $\mathcal{X}$ the set of variables.*

- *The pattern class C_0 consists of linear terms $t \in \mathcal{T}(\mathcal{F}_\emptyset, \mathcal{X}) \backslash \mathcal{X}$.*
- *The pattern class C_1 is the smallest set of almost linear terms in canonical form that contains C_0, all terms t of the form $t = F(x_1, x_2^{\alpha_2}, t_1, \ldots, t_n)$, with $F \in \mathcal{F}_{AC}$, $0 \leq n$, $t_1, \ldots, t_n \in C_0$, $x_1, x_2 \in \mathcal{X}$, $\alpha_2 \geq 0$, and all terms t of the form $f(t_1, \ldots, t_n)$, with $f \in \mathcal{F}_\emptyset$, $t_1, \ldots, t_n \in C_1 \cup \mathcal{X}$.*
- *The pattern class C_2 is the smallest set of almost linear terms in canonical form that contains C_1, all terms of the form $t = F(x_1, x_2^{\alpha_2}, G(x_3, x_4^{\alpha_4}))$ with $F, G \in \mathcal{F}_{AC}$, $x_1, x_2, x_3, x_4 \in \mathcal{X}$, $\alpha_2 \geq 0$, $\alpha_4 > 0$, and all terms t of the form $f(t_1, \ldots, t_n)$, with $f \in \mathcal{F}_\emptyset$, $t_1, \ldots, t_n \in C_2 \cup \mathcal{X}$.*

In our example, the patterns $F(z, f(a, x), g(a))$ and $F(f(a, x), f(y, g(b)))$, have been extended into patterns $F(z', z, f(a, x), g(a))$ and $F(z', f(a, x), f(y, g(b)))$ where z' is an extension variable. Only these two last patterns have to be considered for reduction, and they belong to the class C_1.

3.3.3 Many-to-one AC matching using Compact Bipartite Graphs

Let us first emphasize that the AC matching techniques described in this section are restricted to the class of patterns presented in Section 3.3.2, which leads to several improvements of the general approach described in Section 3.3.1.

- Thanks to the restriction put on patterns, the hierarchy of bipartite graphs has at most two levels, and the second one is degenerate. Thus, the construction can be done without recursion.
- We use a new compact representation of bipartite graphs, which encodes, in only one data structure, all matching problems relative to the given set of rewrite rules.
- No linear Diophantine system is generated since there are at most two variables, with (restricted) multiplicity, under an AC function symbol in the patterns. Instantiating these variables can be done in a simple and efficient way.
- A preliminary syntactic analysis of rewrite rules can determine that only one solution of an AC matching problem has to be found to apply some rule. This is the case for unconditional rules or for rules whose conditions do not depend on a variable that occurs under an AC function symbol in the left-hand side. Taking advantage of the structure of compact bipartite graphs, a refined algorithm is presented to handle those particular (but frequent) cases.

Compact Bipartite Graph

Given a set of patterns with the same syntactic top layer, all subterms with the same AC top function symbol are grouped (at compiled time) to build (at runtime) a particular bipartite graph called a *Compact Bipartite Graph* described below. Given a subject, the compact bipartite graph encodes all matching problems relative to the given set of rewrite rules. All bipartite graphs that the general algorithm would have to construct can be generated from this compact data structure. In general, the syntactic top layer may be not empty and several AC function symbols may occur. In this case, a compact bipartite graph has to be associated to each AC function symbol. Each graph is solved and the solutions have to be combined to solve the matching problem.

Such a decomposition leads us to focus our attention on sets of patterns $p_1, \ldots, p_n$ defined as follows:

$$\begin{array}{lcccccc} p_1 & = F(& p_{1,1} & , \ldots , & p_{1,m_1} &) \\ \vdots & & \vdots & & \vdots & \\ p_n & = F(& p_{n,1} & , \ldots , & p_{n,m_n} &) \end{array}$$

where for some k_j, $0 \leq k_j \leq m_j$, no $p_{j,1}, \ldots, p_{j,k_j}$ is a variable, and all $p_{j,k_j+1}, \ldots, p_{j,m_j}$ are variables. Syntactic subterms $p_{j,k}$ are grouped together and given a subject $s = F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$, a discrimination net that encodes a many-to-one syntactic matching automaton is built. This automaton is used to find pairs of terms $[s_i, p_{j,k}]$ that match each other and to build the associated Compact Bipartite Graph, defined as follows:

$CBG = (V_1 \cup V_2, E)$ where $V_1 = \{s_1, \ldots, s_p\}$, $V_2 = \{p_{j,k} \mid 1 \leq j \leq n, 1 \leq k \leq k_j\}$, and E consists of all pairs $[s_i, p_{j,k}]$ such that $p_{j,k}\sigma = s_i$ for some substitution σ.

Solving a Compact Bipartite Graph

Finding a pattern that matches the subject usually consists of selecting a pattern p_j, building the associated bipartite graph BG_j, finding a maximum bipartite matching [HK73, FM89] and finding assignments to remaining unbound variables. Instead of building a new bipartite graph BG_j each time a new pattern p_j is tried, in our approach, the bipartite graph BG_j is extracted from the compact bipartite graph $CBG = (V_1 \cup V_2, E)$ as follows:

$$BG_j = (V_1 \cup V_2', E') \text{ where } \left\{ \begin{array}{lcl} V_2' & = & \{p_{j,k} \mid p_{j,k} \in V_2 \text{ and } 1 \leq k \leq k_j\} \\ E' & = & \{[s_i, p_{j,k}] \mid [s_i, p_{j,k}] \in E \text{ and } p_{j,k} \in V_2'\} \end{array} \right.$$

The set V_2' contains only vertices associated to the pattern p_j and E' is the set of edges that consists of pairs $[s_i, p_{j,k}]$ matched by p_j.
Given the subject $s = F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ and a fixed j, S_j is a *solution* of BG_j if:

$$\begin{cases} S_j \subseteq E' \text{ and } \forall k \in \{1, \ldots, k_j\}, \exists!\ [s_i, p_{j,k}] \in S_j \\ card(\{[s_i, p_{j,k}] \in S_j \mid 1 \leq i \leq p\}) \leq \alpha_i \end{cases}$$

Roughly speaking, S_j is a solution of the bipartite graph BG_j if all patterns $p_{j,1}, \ldots, p_{j,k_j}$ match a different ground subterm of $\{s_1, \ldots, s_p\}$ (according to multiplicities $\alpha_1, \ldots, \alpha_p$).
This solution corresponds to a maximum bipartite matching for BG_j. If S_j does not exist, the next bipartite graph BG_{j+1} (associated to p_{j+1}) has to be extracted. Note that common syntactic subterms are matched only once, even if they appear in several rules, since the information is saved once for all in the compact bipartite graph.

The main advantage of the many-to-one approach is that it is no longer necessary to inspect the subject more than once to build the compact bipartite graph: given a ground term s_i, the compiled version of the matching automaton traverses positions of s_i only once, and returns a list of $p_{j,k}$ that match s_i. This list of pairs $[s_i, p_{j,k}]$ is directly used to build the compact bipartite graph. Moreover, extraction can be performed efficiently with an adapted data structure: the compact bipartite graph can be represented by a set of bit vectors. A bit vector is associated to each subterm $p_{j,k}$ and the i^{th} bit is set to 1 if $p_{j,k}$ matches to s_i. Encoding compact bipartite graphs by a list of bit vectors has two main advantages: the memory usage is low and the bipartite graph extraction operation is extremely cheap, since only selections of bit vectors are performed.

Considering our running example, an analysis of subterms with the same AC top function symbol F gives three distinct non-variable subterms up to variable renaming: $p_{1,1} = f(a, x)$ and $p_{1,2} = g(a)$ for $F(z', z, f(a, x), g(a)) \to F(z', r_1)$ **if** $z = x$, and $p_{2,1} = f(a, x)$, $p_{2,2} = f(y, g(b))$ for $F(z', f(a, x), f(y, g(b))) \to F(z', r_2)$.

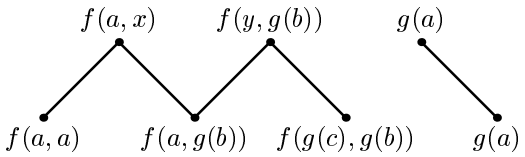
Variable subterms (z, z' in this example) are not involved in the compact bipartite graph construction. They are instantiated later in the substitution construction phase described in Section 3.3.4.

Let us consider the subject:

$$s = F(f(a, a), f(a, g(b)), f(g(c), g(b)), g(a)).$$

The matching automaton, presented in Figure 3.2 on page 65 is successively applied on $f(a, a)$, $f(a, g(b))$, $f(g(c), g(b))$ and $g(a)$ to find the pairs: $[f(a, a), p_{1,1} = p_{2,1}]$, $[f(a, g(b)), p_{1,1} = p_{2,1}]$, $[f(a, g(b)), p_{2,2}]$, $[f(g(c), g(b)), p_{2,2}]$ and $[g(a), p_{1,2}]$.

The matching automaton is used to build the following compact bipartite:



The compact bipartite graph is exploited as follows. A rule has to be selected in order to normalise the subject, for instance $F(z', f(a, x), f(y, g(b))) \to F(z', r_2)$. The bipartite graph that should have been created by the general method can be easily constructed by extracting edges that join $f(a, x)$ and $f(y, g(b))$ to subject subterms, which gives the "classical" bipartite graph presented in the right part of Figure 3.4 on page 66. To check if the selected rule can be applied, a maximum bipartite matching has to be found. The bipartite graph has three solutions:

$$\begin{array}{lll} S = & \{[f(a, a), f(a, x)], & [f(a, g(b)), f(y, g(b))]\} \\ S' = & \{[f(a, a), f(a, x)], & [f(g(c), g(b)), f(y, g(b))]\} \\ S'' = & \{[f(a, g(b)), f(a, x)], & [f(g(c), g(b)), f(y, g(b))]\} \end{array}$$

The given example of compact bipartite graph is represented by only three bit vectors: `1100`, `0110` and `0001`. The first one: `1100`, means that the corresponding pattern $f(a, x)$ matches the two first subterms:

$f(a,a)$ and $f(a,g(b))$, and similarly for the other ones. Extracting the bipartite graph is done by only selecting bit vectors associated to $f(a,x)$ and $f(y,g(b))$: `1100` and `0110`.

In this example, the bipartite graph has three solutions $\{S, S', S''\}$. With the first solution S, the rewrite rule $F(z', f(a,x), f(y,g(b))) \to F(z', r_2)$ can be applied, by instantiating z' to unbounded terms ($f(g(c),g(b))$ and $g(a)$.

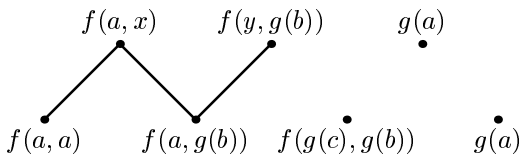
Eager matching

An AC matching problem usually has more than one solution. However, for applying a rule without conditions or whose conditions do not depend on a variable that occurs under an AC function symbol of the left-hand side, there is no need to compute a set of solutions: the first match which is found is used to apply the corresponding rule. Those rules are called *eager rules*. This remark leads us to further specialise our algorithm to get an *eager matching* algorithm, which tries to find a match for eager rules, as fast as possible. The idea consists of incrementally building the whole compact bipartite graph and adding to each step a test to check whether a bipartite graph associated to an eager rule has a solution. This test is not performed for non-eager rules and no check is necessary on a bipartite graph if no modification has occurred since the last applied satisfiability test (i.e. no edge has been added). Using those two remarks, the number of checked bipartite graphs is considerably reduced.

Let us consider our running example. With the first method presented above (called the main algorithm), four matching attempts were done to completely build the compact bipartite graph (corresponding to its five edges). Only after this building phase, bipartite graphs are extracted and solved. Assuming that subterms are matched from left to right, it is sufficient to match only two subterms (with the eager algorithm), to find the first suitable solution:

$$S = \{[f(a,a), f(a,x)], [f(a,g(b)), f(y,g(b))]\}.$$

This solution is found as soon as the following partial compact bipartite graph is built:



In practice, eager matching considerably reduces the number of matching attempts and there is only a small time overhead, due to the test, when no eager rule is applied. In the main algorithm, the number of matching attempts is linear in the number of subterms of the subject. In the eager algorithm, this number also depends on the pattern structure.

Note however that eager matching is not compatible with the concept of priority rewriting, since the eager rule chosen by the eager matching algorithm may not correspond to the first applicable rule in the set of rules ordered by the programmer.[2]

3.3.4 Construction of substitutions

Once matching is performed, the remaining tasks are to instantiate variables and to build the reduced term. In the construction of the reduced term, it is usually possible to reuse parts of the left-hand side to construct the instantiated right-hand side. At least, instances of variables that occur in the left-hand side can be reused. More details can be found in [Vit96] for the syntactic case. Similar techniques have been developed in our compiler, but we do not discuss them here, and rather focus on the construction of substitutions. At this stage of description of the compiler, two problems have to be addressed: how to instantiate the remaining variables in AC patterns? How to optimise the substitution construction?

[2] In Section 3.5, we introduce strategy constructors (called `first` and first_one) that apply rules in a specific order: in this case, the eager matching algorithm cannot be used.

Variable instantiation

Variables that occur in patterns just below an AC function symbol are not handled in the previously described phases of the compiler. This problem is delayed until the construction of substitutions. When only one or two distinct variables (with multiplicity) appear directly under each AC function symbol, their instantiation does not need to construct a linear Diophantine system. Several cases can be distinguished according to the syntactic form of patterns.

- For $F(x_1, t_1, \ldots, t_n)$, once $t_1, \ldots, t_n$ are matched, all the unmatched subject subterms are captured by x_1.
- For $F(x_1, x_2^{\alpha_2}, t_1, \ldots, t_n)$, let us first consider the case where $\alpha_2 = 1$. Then once $t_1, \ldots, t_n$ are matched, the remaining subject subterms are partitioned into two non-empty classes in all possible ways. One class is used to build the instance of x_1, the other for x_2.

  If $\alpha_2 > 1$, once $t_1, \ldots, t_n$ are matched, one tries to find in all possible ways α_2 identical remaining subjects to match x_2 and then, all the remaining unmatched subject subterms are captured by x_1.

Considering our running example, and the rule $F(z', z, f(a, x), g(a)) \to F(z', r_1)$ **if** $z = x$. Once matching have been performed, and the two solutions for x (a and $g(b)$) have been found, the variables z and z' can be instantiated to $F(f(a, g(b)), f(g(c), g(b)))$ or $F(f(a, a), f(g(c), g(b)))$ or subterms directly under the two F symbols. The condition $z = x$ is never satisfied with those substitutions, so application of this rule fails.

Compiling the substitution construction

In the syntactic case, the matching substitution is easily performed by the discrimination net, since there is at most one solution. In the AC case, there may be many different instantiations for each variable. It would be too costly to store them in a data structure for possible backtracking. Furthermore, the construction of this dynamic data structure is not necessary when the first selected rule is applied, because all computed substitutions are deleted. Our approach consists of computing the substitution only when a solution of the bipartite graph is found. For each subterm $p_{j,k}$, variable positions are known at compile time and used to construct an access function $access\_p_{j,k}$ from terms to lists of terms. This function takes a ground term as argument and returns the list of instances of variables of $p_{j,k}$ as a result. Given $S_j = \{[s_i, p_{j,k}]\}$ a solution of BG_j, the set $I_j = \{access\_p_{j,k}(s_i) \mid [s_i, p_{j,k}] \in S_j\}$ of variable instantiations can be computed.
Given the rule $F(z', f(a, x), f(y, g(b))) \to F(z', r_2)$, the functions $access\_f(a, x)(t) = t_{|2}$ and $access\_f(y, g(b))(t) = t_{|1}$ are defined. Starting from $S_2 = \{[f(a, a), f(a, x)], [f(a, g(b)), f(y, g(b))]\}$, the set $I_2 = \{a, a\}$ is easily computed, and we get the substitution $\sigma = \{x \mapsto a, y \mapsto a\}$.

3.3.5 Handling other patterns

In order to handle rules whose patterns are not in C_2, a program transformation is applied. It transforms these rules into equivalent ones whose left-hand sides are in the class C_2.

Any rule can be transformed into a conditional rule with matching conditions and satisfying our pattern restrictions. The transformation preserves the semantics in the sense that a term is reducible by the initial rule if and only if it is reducible by the transformed rule.

Let l be a left-hand side of rule which does not belong to C_2, and Λ be an abstraction function that replaces non-variable subterms of l, say u_j, by new variables, say x_j, in such a way that $l' = \Lambda(l)$ is in the class C_2. Let k be the number of abstracted subterms. The new rule

$$\begin{aligned} l' \to r \quad & \textbf{where } u_1 := x_1 \\ & \vdots \\ & \textbf{where } u_k := x_k \end{aligned}$$

is equivalent to $l \to r$.

When using such a transformation approach, the efficiency of the resulting system partially depends on the one-to-one AC matching algorithm used for the matching conditions: simple rules that belong to the

presented pattern classes are efficiently compiled, and complex rules that were not in C_2 are transformed and compiled.[3]

Example 3.3.1 *Let $\cup$, Eq be two AC operators, e, solve and $\mathsf{simplify}$ be three syntactic operators, and $r(x_1, x_2, x_3)$ any term where the variables x_1, x_2, x_3 occur. Let us consider the following rule:*

$$\mathsf{solve}(\mathsf{simplify}(x_1 \cup Eq(\mathsf{e}(x_2), \mathsf{e}(x_3)))) \to r(x_1, x_2, x_3)$$

A transformation has to be applied because the left-hand side of the rule does not belong to C_2. Let $\Lambda = \{\mathsf{e}(x_2) \mapsto y_2, \mathsf{e}(x_3) \mapsto y_3\}$ be the abstraction function. The following rule now belongs to the class C_2:

$$\mathsf{solve}(\mathsf{simplify}(x_1 \cup Eq(y_2, y_3))) \to r(x_1, x_2, x_3) \quad \textbf{where } \mathsf{e}(x_2) := y2 \quad \textbf{where } \mathsf{e}(x_3) := y3$$

It is worth recalling that when computing such matching conditions, only one-to-one matching problems occur. If a pattern u_j contains an AC function symbol, a general one-to-one AC matching procedure, such as the one described in [Eke95], is called. In the worst case, our many-to-one AC matching is not used and the program transformation builds a rewrite rule system where AC problems are solved with a one-to-one AC matching procedure in the **where** parts, helped by a full indexing for the topmost free function symbol layer. This is also a frequently implemented matching technique, used in Maude [CELM96] for instance.

3.4 Optimisations

Before concluding the compilation of the normalisation process, let us mention a few useful optimisations.

3.4.1 Maintaining canonical forms.

The compact bipartite graph construction (and thus the matching phase) assumes that both pattern and subject are in canonical form. Instead of re-computing the canonical form after each right-hand side construction, one can maintain this canonical form during the reduced term construction. Whenever a new term t is added as a subterm of $s = F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$, if an equivalent subterm s_i already exists, its multiplicity is incremented, else, the subterm t (which is in canonical form by construction) is inserted in the list $s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}$ at a position compatible with the chosen ordering. If t has the same AC top symbol F, a flattening step is done and the two subterm lists are merged with a merge sort algorithm.

Definition 3.4.1 *Let us define the function* mcf *taking as arguments two terms $s = F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ and $t = G(t_1^{\beta_1}, \ldots, t_m^{\beta_m})$ in canonical form, as follows:*

- *case $F \neq G$ (s and t have different top symbol)*
  – if there exists i in $[1 \ldots p]$ such that $s_i = t$ the multiplicity α_i is incremented:
  $\mathrm{mcf}(F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}), t) = F(s_1^{\alpha_1}, \ldots, s_i^{\alpha_i+1}, \ldots, s_p^{\alpha_p})$
  – else, there exists i in $[1 \ldots p]$ such that $\forall j \leq i, s_j > t$ and $\forall j > i, t > s_j$:
  $\mathrm{mcf}(F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}), t) = F(s_1^{\alpha_1}, \ldots, s_i^{\alpha_i}, t, s_{i+1}^{\alpha_{i+1}}, \ldots, s_p^{\alpha_p})$
- *case $F = G$ (s and t have same top symbol)*
  $\mathrm{mcf}(F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}), t) = F(u_1^{\gamma_1}, \ldots, u_k^{\gamma_k})$ *such that $(u_1^{\gamma_1}, \ldots, u_k^{\gamma_k})$ is the merged sort (without multiple occurrences) of $(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ and $(t_1^{\beta_1}, \ldots, t_m^{\beta_m})$*

From the definition of mcf, it is easy to get the following result: let $s = F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ and t be two terms in canonical form. The function mcf applied to s and t returns the canonical form of $F(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}, t)$. As a consequence, the canonical form of a term can be obtained by a bottom-up construction using the mcf function.

[3] In the current version of the ELAN compiler, we use the algorithm presented in [Eke95] which is also used in C*i*ME [Mar96] and in the ELAN interpreter.

3.4.2 Normalised substitutions.

In the case of the leftmost-innermost reduction strategy, nested function calls are such that before a matching phase, each subterm is in normal form w.r.t. the rewrite rule system. In syntactic rewriting, when a pattern p matches a ground term s, all variables of p are assigned to a subterm of s which is irreducible by construction.

This is no longer the case in AC rewriting because variables that appear directly under an AC top symbol may be instantiated to a reducible combination of irreducible subterms. For instance, in our running example, the variable z can be instantiated to $F(f(a, g(b)),\ f(g(c),\ g(b)))$ which is reducible by the rule $F(z', f(a, x),\ f(y, g(b))) \to F(z', r_2)$.

To ensure that instances of variables occurring immediately under an AC top function symbol are irreducible, these instances are normalised before using them to build the right-hand side. Moreover, if the considered rule has a non-linear right-hand side, this normalisation step allows reducing the number of further rewrite steps: the irreducible form is computed only once. Without this optimisation, normalisation of identical terms frequently occurs even if a shared data structure is used, because flattening can create multiple copies.

3.4.3 Using colors to avoid unnecessary normalised substitutions.

Normalising reducible instances of variables considerably reduces the number of applied rules, but renormalising a term already in normal form involves extra work that introduces an overhead. Let us note that all subterms of the subject are in normal form by construction and that an instance of a variable that appears immediately under an AC top function symbol is irreducible if this instance is a subterm of an irreducible term. This remark leads to further improve the algorithm used to build ground reduced terms:

- whenever a term rooted by an AC symbol is built with the mcf function, a different "color" is assigned to all its immediate subterms.
- whenever an irreducible term rooted by an AC symbol is reached by normalisation, a same "color" is assigned to all its immediate subterms.
- in the algorithm that computes the canonical form of a term, if α identical subterms t appear, they are replaced by a single instance of the subterm with multiplicity α and a special color, say *bicolor*, is assigned to this subterm.

Coming back to the original problem of checking whether the term s assigned by the matching substitution to the variable x is irreducible, it is now possible to inspect the colors of immediate subterms of s: if all subterms have the same color and none of them is *bicolor*, the term s is a subterm of the subject (irreducible by construction), so it is not necessary to normalise s again.

3.5 Determinism analysis

Let us now turn to the problem of non-determinism. The fact that a computation may have several results can be taken into account either by introducing explicitly sets of results or by a backtracking capability to enumerate the elements of this set. We adopt here the second approach and we introduce strategy constructors to specify whether a function call returns several, at least one or only one result.

For implementation of backtracking, two functions are usually required: the first one, to create a choice point and save the execution environment; the second one, to backtrack to the last created choice point and restore the saved environment. Many languages that offer nondeterministic capabilities provide similar functions: for instance world+ and world- in Claire [CL96], try and retry in WAM [War83, AK90], onfail, fail, createlog and replaylog in the Alma-0 Abstract Machine [Par97, AS97]. Following [Vit96], two flow control functions, setChoicePoint and `fail`, have been implemented in assembly language. The setChoicePoint function sets a choice point, and the computation goes on. The `fail` function performs a jump into the last call of setChoicePoint. These functions can remind the pair of standard C functions `setjmp` and `longjmp`. However, the `longjmp` can be used only in a function called from the function setting `setjmp`. The two functions setChoicePoint and `fail` do not have such a limitation. Their implementation is described in [Mor98].

In order to take into account sets of results, we use the concept of strategy: a strategy is a function which, when applied to an initial term, returns a set of possible results (more precisely a multiset of results). The strategy fails if the set is empty. To precisely define how sets of results are handled, we introduce the following strategy constructors.

- A labelled rule is a primal strategy. The result of applying a rule labelled lab on a term t returns a multiset of terms. This primal strategy fails if the multiset of resulting terms is empty.
- Two strategies can be concatenated by the symbol ";", i.e. the second strategy is applied on all results of the first one. $S_1; S_2$ denotes the sequential composition of the two strategies. It fails if either S_1 fails or S_2 fails. Its results are all results of S_1 on which S_2 is applied and gives some results.
- dc$(S_1, \ldots, S_n)$ chooses one strategy S_i in the list that does not fail, and returns all its results. This strategy may return more than one result, or fails if all sub-strategies S_i fail.
- first$(S_1, \ldots, S_n)$ chooses the first strategy S_i in the list that does not fail, and returns all its results. Again, this strategy may return more than one result, or fails when all sub-strategies S_i fail.
- dc_one$(S_1, \ldots, S_n)$ chooses one strategy S_i in the list that does not fail, and returns its first result. This strategy returns at most one result or fails if all sub-strategies fail.
- first_one$(S_1, \ldots, S_n)$ chooses the first strategy S_i in the list that does not fail, and returns one of its first results. This strategy returns at most one result or fails if all sub-strategies fail.
- dk$(S_1, \ldots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This multiset of results may be empty, in which case the strategy fails.
- The strategy id is the identity that does nothing but never fails.
- fail is the strategy that always fails and never gives any result.
- *repeat**(S) applies repeatedly the strategy S until it fails and returns the results of the last unfailing application. This strategy may return more than one result but can never fail because zero applications of S is possible: in this case the initial term is returned.
- The strategy iterate(S) is similar to repeat*(S) but returns all intermediate results of repeated applications.

The strategy constructors introduced here are quite close to other tactics languages used on proof systems designed in the LCF style [Plo77, GMW79], such as for instance Isabelle [Pau94]. They have been chosen to express main control constructions: concatenation, iteration and search. All these constructors are part of the ELAN language, and have been useful to design in ELAN theorem proving and constraint solving tools.

From now on, let us consider that not only rules but also strategies can be applied on terms. $[S](t)$ denotes the application of the strategy S on the term t that produces a multiset of results. Indeed a rule itself may call a strategy in its matching conditions that are now of the form **where** $p := [S](t)$.

It may be interesting to remark that ELAN does not provide any strategy constructor for negation, simply because it may be expressed using others constructors. Let us consider the strategy $S' =$ first$(S;$ fail, id$)$. This strategy S' fails when S succeeds and S' succeeds when S fails. This example also illustrates the use of the id constructor.

Example 3.5.1 *We have shown in the introduction a simple rule that removes an element from a set, and returns the element and the new set. This rule is now labelled by* extract*:*

$$[extract] \quad (i) \cup S \quad \rightarrow \quad [i, S]$$

The strategies dk(extract)*,* dc(extract) *and* first(extract) *are all equivalent in this case and enumerate the elements of the set. On the other hand,* dc_one(extract) *and* first_one(extract) *produce only one result which corresponds to the first match which is found.*

To implement the N-queens problem, we need two more rules, where set, sol, s_1, p_1 are variables, and check a predicate that is satisfied when a queen can be set in position p_1 without being attacked by a previously partial solution sol:

$$\begin{array}{lllll} \textit{[queens]} & state(set, sol) & \rightarrow & state(s_1, p_1 \cdot sol) \\ & & & \textbf{\textit{where}}\ [p_1, s_1] := [\texttt{dk}(extract)](set) \\ & & & \textbf{\textit{if}}\ check(1, p_1, sol) \\ \textit{[final]} & state(\emptyset, sol) & \rightarrow & sol \end{array}$$

We also need to define a strategy that controls the application of these two rules:

$$qStrat \rightarrow \textit{repeat*}(\texttt{dk}(queens)); \textit{final}$$

The most important statement is: ***where*** $[p_1, s_1] := [\texttt{dk}(extract)](set)$.
When applied on a set of integers, for instance $set = (1) \cup (2) \cup \ldots \cup (8)$*, the strategy* dk(extract) *non-deterministically applies the rule* extract*, and non-deterministically chooses an assignment for i and S. In our case, there are 8 solutions. Each assignment of i and S is stored in a pair $[p_1, s_1]$, then the check predicate is evaluated. When it is satisfied, the position p_1 is added to the partial solution l, and the* queens *rule is applied again, thanks to the* repeat*(dk(queens)) *strategy. When the set of potential positions is empty (represented by the constructor $\emptyset$), each queen has a compatible assignment and the* final *rule is applied to return a solution of the N-queens problem. By using* dk(queens) *in the qStrat strategy, the search space is fully explored and we obtain the complete set of solutions to the N-queens problem. Using instead* dc_one(queens) *would result in searching for only one solution.*

In order to efficiently deal with strategies and this more general notion of rules, our compiler incorporates a static analysis phase that annotates every rule and strategy in the program with its determinism. This determinism information is used in later phases of the compiler: the matching phase, various optimisations on the generated code and detection of non termination. The determinism analysis runs after the type-checking analysis, the transformation of rules to fit into the restricted class of patterns described in Section 3.3.2, and the linearisation of patterns (left-hand sides of rewrite rules), but before the many-to-one AC matching compilation phase.

In order to facilitate the determinism analysis, we introduce four primitive operators that allow us to classify the cases according to two different levels of control.
Controlling the number of results: given a rewrite rule or a strategy,
- the one operator builds a strategy that returns at most one result;
- the all operator builds a strategy that returns all possible results of the strategy or the rule.

Controlling the choice mechanism: given a list of strategies (possibly reduced to a singleton),
- the select_one operator chooses and returns a non-failing strategy among the list of strategies;
- the select_first operator chooses and returns the first (from left to right) non-failing strategy among the list of strategies;
- the select_all operator returns all unfailing strategies.

In the current version of ELAN, these five primitives are hidden from the user and are internally used to perform the determinism analysis. However, all strategy constructors `dk`, `dc`, `first`, dc_one and first_one can be expressed using these primitives, using the following axioms, where S_i stands for a rule or a strategy:

$\texttt{dk}(S_1, \ldots, S_n)$	=	$\textsf{select\_all}(\textsf{all}(S_1), \ldots, \textsf{all}(S_n))$
$\texttt{dc}(S_1, \ldots, S_n)$	=	$\textsf{select\_one}(\textsf{all}(S_1), \ldots, \textsf{all}(S_n))$
$\texttt{first}(S_1, \ldots, S_n)$	=	$\textsf{select\_first}(\textsf{all}(S_1), \ldots, \textsf{all}(S_n))$
$\textsf{dc\_one}(S_1, \ldots, S_n)$	=	$\textsf{select\_one}(\textsf{one}(S_1), \ldots, \textsf{one}(S_n))$
$\textsf{first\_one}(S_1, \ldots, S_n)$	=	$\textsf{select\_first}(\textsf{one}(S_1), \ldots, \textsf{one}(S_n))$

Note that `dk`, `dc` and `first` operators are equivalent if they are applied on a unique argument: $\texttt{dk}(S) = \texttt{dc}(S) = \texttt{first}(S) = S$.

3.5.1 Determinism

For each strategy, a determinism information is inferred according to the maximum number of results it can produce (one or more than one) and whether or not it can fail before producing its first result. We adopt the same terminology for determinism as in Mercury [HCS96, HSC96]:

- if the strategy has exactly one result, its determinism is *deterministic* (det)
- if the strategy can fail and has at most one result, its determinism is *semi-deterministic* (semi)
- if the strategy cannot fail and has more than one result, its determinism is *multi-result* (multi)
- if the strategy can fail and may have more than one result, its determinism is *non-deterministic* (nondet)
- if the strategy always fail, i.e. has no result, its determinism is *failure* (`fail`)

A partial ordering on this determinism is defined as follows:

$$\mathsf{det} < \mathsf{semi}, \mathsf{multi} < \mathsf{nondet}$$

and intuitively corresponds to an inclusion ordering on the intervals which the number of results belongs to:

$$[1,1] < [0,1], [1,+\infty[< [0,+\infty[$$

The algorithm for inferring the determinism of strategies uses two operators *And* and *Or* that intuitively correspond to the composition and the union of two strategies (the union of two strategies is defined by the union of their results). Their values given in the following tables should be clear from the semantics given to the different determinisms. For instance, a conjunction of two strategies is semi-deterministic if any one can fail and none of them can return more than one result ($And(\mathsf{det}, \mathsf{semi}) = And(\mathsf{semi}, \mathsf{det}) = And(\mathsf{semi}, \mathsf{semi}) = \mathsf{semi}$). These values can be also computed with operations on boolean variables as for instance in [HSC96].

And	det	semi	multi	nondet	fail
det	det	semi	multi	nondet	fail
semi	semi	semi	nondet	nondet	fail
multi	multi	nondet	multi	nondet	fail
nondet	nondet	nondet	nondet	nondet	fail
fail	fail	fail	fail	fail	fail

Or	det	semi	multi	nondet	fail
det	multi	multi	multi	multi	det
semi	multi	nondet	multi	nondet	semi
multi	multi	multi	multi	multi	multi
nondet	multi	nondet	multi	nondet	nondet
fail	det	semi	multi	nondet	fail

3.5.2 Determinism inference

The algorithm for inferring the determinism is presented here in three steps: for a strategy, it uses the decomposed form of the strategy into the primitives introduced above. For a rule, it analyses the determinism of the matching conditions. Finally it deals with the recursion problem due to the fact that strategies are built from rules and that rules call strategies in their matching conditions.

Strategy detism inference

The detism of a strategy is inferred from its expression using one, all, select_one and select_all.

- detism(one(S)) = semi if S is a rewrite rule, since application of a rewrite rule may fail; otherwise,
  detism(one(S)) = $\begin{cases} \text{det} & \text{if detism(S) is det or multi} \\ \text{semi} & \text{if detism(S) is semi or nondet} \end{cases}$
- detism(all(S)) = And(semi, detism(S)) if S is a rewrite rule, since application of a rewrite rule may fail; otherwise, detism(all(S)) = detism(S)
- detism(repeat*(S)) = $\begin{cases} \text{det} & \text{if detism(S) is det or semi} \\ \text{multi} & \text{if detism(S) is multi or nondet} \end{cases}$

  The repeat* operator cannot fail because zero application of the strategy is allowed. Note that if S cannot fail, the repeat* construction cannot terminate.
- detism(iterate(S)) = multi. The iterate operator cannot fail either. In general, it returns more than one result because all intermediate steps are considered as results. If S cannot fail, the iterate construction cannot terminate, but this is quite useful to represent infinite data structures, like infinite lists.
- detism($S_1; S_2$) = And(detism(S_1), detism(S_2)).
- detism(select_one($S_1, \ldots, S_n$)) = And(detism(S_1), ..., detism(S_n))
- detism(select_all($S_1, \ldots, S_n$)) = Or(detism(S_1), ..., detism(S_n))

Rule detism inference

Inferring the determinism of a rewrite rule R consists of analysing the determinism of its matching conditions.

- Let us first consider a matching condition **where** $p := c$ where c does not involve any strategy. The normalisation of c (with unlabelled rules) cannot fail. If p does not match the normalised term, the current rule cannot be applied, but this does not modify the detism of the rule. Such a condition is usually said to be deterministic (det is a neutral element for the And operator).

  The only different situation is when a variable of c occurs in the left-hand side of the rule or in a pattern of a previous matching condition with AC function symbols: if this variable is involved in an AC matching problem, it may have several possible instances, thus, an application of the rule may return more than one result. The matching condition is said to be multi.
- Let us now consider a matching condition **where** $p := [S](t)$ involving a strategy call. Then the matching condition has in general the determinism of the strategy S, except as before when a variable of t occurs in the left-hand side of the rule or in a pattern of a previous matching condition with AC function symbols: the detism of the matching condition is multi or nondet, and is computed as And(multi, detism(S)).

The determinism of the rewrite rule R is the conjunction (And operation) of the inferred determinisms of all its matching conditions.

Recursion problem

In general, strategy definitions may be (mutually) recursive. So the detism of a strategy may depend on itself. A similar problem arises in logic programming for finding the determinism of a predicate [ST85]. To avoid non-termination of the determinism analysis algorithm, when the detism of a strategy depends on itself, a default determinism is given. On the strategy constructors, this default corresponds to the maximum of the determinism in the ordering < that the strategy can have and is given in the next table.

constructor	**one**	**all**	**repeat***	**iterate**	**;**
default detism	semi	nondet	multi	multi	nondet

In order to refine this brute force approximation we plan to explore a fixpoint technique similar to the one used in Sictus Prolog [Sah91].

3.5.3 Impact of determinism analysis

The determinism analysis enables us to design better compilation schemes for det or semi strategies. With this approach, the search space size, the memory usage, the number of necessary choice points, and the time spent in backtracking and memory management can be considerably reduced. We can also take benefit from the determinism analysis to improve the efficiency of AC matching and to detect some non-terminating strategies.

Several optimisations can be done to improve the backtracking management:

- When compiling a set of rules whose matching conditions are deterministic, no choice point is needed because no backtracking can occur between the matching conditions.
- When compiling a set of deterministic rules with some non-deterministic matching conditions, some choice points are needed to handle the backtracking. Note that all set choice points can be removed when the rule is applied, because at most one result is needed. For instance, when searching only one solution in a problem where several choice points are needed, one can delete them after finding the first solution.
- When dealing with non-deterministic strategies and the repeat* constructor, a lot of choice points have to be set, because the strategy is recursively called in all branches of the computation space. The situation can be depicted as follows, where the bullet represents a set choice point.

  $$t \bullet \overset{S}{\underset{S}{\rightleftarrows}} t_1 \bullet \overset{S}{\underset{S}{\rightleftarrows}} \cdots \bullet \overset{S}{\underset{S}{\rightleftarrows}} t_n \bullet \overset{S}{\underset{S}{\rightleftarrows}} \texttt{fail}$$

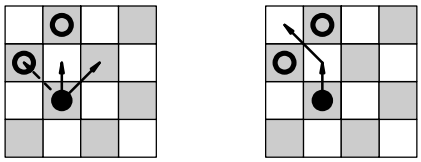
  One choice point per step is needed, and when a failure occurs, one choice point only is deleted and the process goes on.

  This is no more the case when compiling a strategy $repeat*(S)$ where S is det or semi. The compilation scheme then consists of setting a single choice point and trying to apply the strategy S as many times as possible. Each time the strategy S is applied, the resulting term is saved in a special variable $lastTerm$. When a failure occurs, the choice point is deleted and the saved term $lastTerm$ is returned. The situation is depicted as follows:

  $$\bullet\, t \longrightarrow^S t_1 \longrightarrow^S \cdots \longrightarrow^S t_n \longrightarrow \texttt{fail}$$

In order to illustrate this last point, let us consider the following example.

Example 3.5.2 *Let us consider a simple modeling of a game: a pawn on a chessboard can move in several directions (see Figure 3.5), each of them corresponding to one labelled rule d_i.*



(a) (b)

Figure 3.5: (a) – by applying `dk(move)` the pawn can move in three possible directions. (b) – an external square can be reached if the strategy `repeat*(dc_one(move))` is applied.

Exploring all possibilities of moves for this pawn in one step can be expressed by a strategy `move` $\rightarrow$ `dk`$(d_1, \ldots, d_n)$*, where $d_1, \ldots, d_n$ are basic moves. Once a move has been performed, in some situation, it*

may be considered as a definitive choice and the search space related to all other moves is forgotten. This is performed via a strategy `dc_one(move)`*. In order to iterate this process, the strategy* `repeat*(dc_one(move))` *repeatedly moves a given pawn up to a failure: in this example, a pawn cannot move when an external square is reached.*

This simple game is an example of situation where the last presented impact of determinism analysis is crucial: by reducing to one the number of set choice point, it considerably improves the efficiency and reduces the memory needed.

Other advantages of determinism analysis are related to the rewriting process. To improve efficiency of rewriting, a well-known idea is to reuse parts of left-hand sides of rules to construct the right-hand sides [DFG+94, Vit96]. This technique avoids memory cell copies and reduces the number of allocations. Unfortunately, the presence of non-deterministic strategies and rules limits its applicability, because backtracking requires access to structures that would otherwise be reused. The determinism information is then used to detect cases where reusing is possible.

The determinism analysis is also important to design more efficient AC matching algorithms: when a rule is deterministic, only the first match which is found is needed to apply a rewrite step. This remark has to be related to the design of the *eager matching* algorithm, described in Section 3.3.3, which avoids building the whole compact bipartite graph before solving it. Experiments show a reduction of the number of matching attempts up to 50%, which significantly improves the overall performance of the system.

Finally the determinism analysis is also useful to detect some non-terminating strategies, such as a strategy $repeat*(S)$, where S never fails. Detecting this non-termination problem at compile time allows the system to give a warning to the programmer and can help him to improve his strategy design.

3.6 A compiler for ELAN

The techniques described in the previous sections have been implemented in the ELAN system [KKV95]. ELAN provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It offers a natural and simple logical framework for the combination of the computation and deduction paradigms. It supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures and it offers a modular framework for studying their combination.

ELAN takes from functional programming the concept of abstract data types and the function evaluation principle based on rewriting. In ELAN a rewrite rule may be labelled, may have boolean conditions introduced by the keyword **if**, and matching conditions introduced by the keyword **where**. The evaluation mechanism also involves backtracking since in ELAN, a computation may have several results. One of the original aspects of the language is that it provides a strategy language allowing the programmer to specify the control used during rule applications. This is in contrast to many existing rewriting-based languages where the term reduction strategy is hard-wired and not accessible to the designer of an application. The strategy language offers primitives for sequential composition, iteration, deterministic and non-deterministic choices of elementary strategies that are labelled rules. From these primitives, which correspond to the strategy constructors defined in Section 3.5, more complex strategies can be expressed. In addition, the user can introduce new strategy operators and define them by rewrite rules. Evaluation of strategy application is itself based on rewriting. Moreover it should be emphasised that ELAN has logical foundations based on rewriting logic [Mes92] and detailed in [BKK96, BKK98a]. So the simple and well-known paradigm of rewriting provides both the logical framework in which deduction systems can be expressed and combined, and the evaluation mechanism of the language.

The current version of ELAN includes an interpreter and a compiler written respectively in C++ and Java, a library of standard ELAN modules, a user manual and examples of applications. Among those, let us mention for instance the design of rules and strategies for constraint satisfaction problems [Cas98], theorem proving tools in first-order logic with equality [KM95, CK97], the combination of unification algorithms and of decision procedures in various equational theories [Rin97, KR98a]. More information on the system can be found on the web site.[4]

[4] http://www.loria.fr/ELAN.

A first ELAN compiler was designed and presented in [Vit96]. Experimentations made clear that a higher-level of programming is achieved when some functions may be declared as associative and commutative. The new ELAN compiler has been implemented by the second author with approximatively 20,000 lines of Java. A runtime library has been implemented in C to handle basic terms and AC matching operations. This library contains more than 10,000 lines of code.

To give an intuition about the performance of the ELAN system compared to programming languages largely used in practice, we show in Table 3.6 the results of a brief comparison[5] with the Objective Caml (v 2.02) functional programming system and the GNU Prolog (v 1.1.2) logic programming system. We used the Fib benchmark which computes 100 times the N^{th} Fibonacci number, and the N-queens program illustrated in Example 3.5.1 on page 74.

(time in second)	OCaml	GNU Prolog	Elan
Fib(23)	0.67	12.08	1.12
Fib(25)	1.75	31.77	2.39
N-queens(10)	0.580	1.750	1.07
N-queens(12)	19	57	31.2

Table 3.1: Very small comparison of Objective Caml, GNU Prolog and ELAN

3.7 Conclusion

We have presented in this chapter the main techniques used in our ELAN compiler, that we hope to be useful too for other rewriting based languages. Actually, from the point of view of matching and rewriting, ELAN can be compared to other systems such as OBJ [GW88], ASF+SDF [Kli93], Maude [CELM96] or Cafe-OBJ [FN97]. Maude also provides efficient AC rewriting and ASF+SDF performs list matching, a specific instance of AC matching. However, these languages do not involve non-deterministic strategy constructors. With respect to the determinism analysis, ELAN is closer to logic programming languages such as Alma-0 [AS97] or Mercury [HCS96]. In our case, the determinism analysis simply makes possible to run programs that could not be executed before due to memory explosion. This analysis significantly decreased the number of set choice points and improved the performance.

It seems now that further improvements of the ELAN compiler rely on the backtracking management. The setChoicePoint and `fail` functions implemented in assembly language turned out to be very useful for designing complex compilation schemas. A deeper analysis reveals that useless information is also stored in local environments. So it should be possible to improve the low-level management of non-determinism and to combine this with an efficient garbage collector. Together with this re-design of the memory management, we think of using a shared terms library [vdBKO99, VdBdJKO00], as in ASF+SDF, or using a generational garbage collection approach as is Haskell [SPJ93].

Significant examples have been handled in ELAN that took benefit from the compilation methods developed in this chapter. Let us mention three of them.

- Techniques used in solving constraint satisfaction problems are based on exploration of the space of all solutions with backtracking, and problem reduction techniques that reduce the set of values that the variables can take. As explained in [Cas98], such techniques are expressible by rules and strategies and the system Colette implements them in ELAN.

- In the specification of authentication protocols [Cir99] the protocol, the intruder and the attack are modeled by rules and strategies. ELAN behaves like a model checker by generating all possible situations and looking for situations revealing an attack.

- Planning and scheduling problems have been explored in [DK98, DK99]. In this case, ELAN is used as a decision support tool, which can simulate plan executions and explore consequences of decision-making during a planification.

These three application areas have in common to involve set data structures, to develop huge search spaces, and to need non-deterministic rules and strategies. For running such examples, the use of the ELAN

[5] On a Sun Enterprise with Solaris 5.6.

compiler is essential. But the achievement of such significant programs comforts the affirmation that the rewriting paradigm can be promoted to the level of a realistic programming language.

We feel that the techniques presented in this chapter could benefit the functional programming community in at least two directions. The first is to add built-in equational theories in higher-order matching and rewriting. This was already explored by several authors [Wad87, JO91, NP98]. The second direction is to increase higher-order features of rewriting based languages. A promising approach to bring closer rewriting-based languages and functional languages is the rewriting calculus (ρ-calculus) proposed in [CK99]. This is a framework in which rule, rule application, and sets of results are explicit objects. The main intuition is that a rewrite rule is an abstractor generalising λ-abstraction: the left-hand side of a rule determines the bound variables and the contextual structure. The calculus handles non-determinism via sets of results. ELAN is actually an implementation of a large part of this calculus. To come closer to a functional language, the syntax should be extended by λ-expressions. The study of compilation techniques for such an extension, inspired from those used in the two classes of languages, is certainly a challenging research and development issue.

Bibliography

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[AG96] T. Arts and J. Giesl. Termination of constructor systems. In H. Ganzinger, editor, *Proc. 7th Intl. Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 1996.

[AG97] T. Arts and J. Giesl. Automatically proving termination where simplification orderings fail. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development (TAPSOFT'97), Proc. 7th Intl. Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 261–272. Springer, 1997.

[AK90] H. Aït-Kaci. The WAM: a (real) tutorial. Technical report 5, Digital Systems Research Center, Paris (France), January 1990.

[AK92] M. Adi and Claude Kirchner. Associative commutative matching based on the syntacticity of the AC theory. In Franz Baader, J. Siekmann, and W. Snyder, editors, *Proceedings 6th International Workshop on Unification, Dagstuhl (Germany)*. Dagstuhl seminar, 1992.

[AM90] J. Avenhaus and K. Madlener. Term rewriting and equational reasoning. In R.B. Banerji, editor, *Formal Techniques in Artifial Intelligence*, pages 1–43. Elsevier Science Publishers B.V. (North-Holland), 1990.

[Arm97] J. Armstrong. The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, June 1997.

[AS97] K. R. Apt and A. Schaerf. Search and Imperative Programming. In *24th POPL*, pages 67–79, 1997.

[AVWM96] J. Armstrong, R. Virding, C. Wikström, and Williams. M. *Concurrent Programming in Erlang*. Prentice-Hall International, 1996.

[Bac88] L. Bachmair. Proof by consistency in equational theories. In *Proc. 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*, pages 228–233, 1988.

[Bac91] L. Bachmair. *Canonical equational proofs*. Computer Science Logic, Progress in Theoretical Computer Science. Birkhäuser Verlag AG, 1991.

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[Bar84] H.P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B.V. (North-Holland), second edition, 1984.

[BCR93] Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, LNCS 668, pages 61–74, Orsay, France, April 13–17, 1993. Springer-Verlag.

[BD86] L. Bachmair and N. Dershowitz. Commutation, transformation and termination. In J. Siekmann, editor, *Proc. 8th Intl. Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 1986.

[BD87] L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. In *Proc. 2nd Conference on Rewriting Techniques and Applications*, volume 256 of *Lecture Notes in Computer Science*, pages 192–203. Springer, 1987.

[BD89] L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. *Theoretical Computer Science*, 67(2-3):173–202, 1989.

[BDH86] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proc. 1st IEEE Symposium on Logic in Computer Science, Cambridge, Mass. (USA)*, pages 346–357. IEEE, 1986.

[BDP89] L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Vol. 2: Rewriting Techniques*, pages 1–30. Academic Press Inc., 1989.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):1–31, 1994.

[BGLS92] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In *Proc. 11th Intl. Conference on Automated Deduction, Saratoga Springs, N.Y. (USA)*, pages 462–476, 1992.

[BGLS95] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[Bid81] M. Bidoit. *Une Méthode de Présentation des Types Abstraits, Applications.* Thèse de Doctorat de Troisième Cycle, Université de Paris-Sud, 1981.

[BJ97] A. Bouhoula and J.-P. Jouannaud. Automata-driven automated induction. In *Proc. 12th IEEE Symposium on Logic in Computer Science, Warsaw (Poland)*, pages 14–25. IEEE Comp. Soc. Press, 1997.

[BJM97] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proc. Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer, 1997.

[BK86] J. A. Bergstra and J. W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.

[BKK96a] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, 1996.

[BKK96] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996.

[BKK98a] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In Masahiko Sato and Yoshihito Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific. Also report LORIA 98-R-165.

[BKK+96b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, 1996.

[BKK+98b] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume15.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[BKKR01] Peter Borovanský, Claude Kirchner, Helene Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.

[BKN87] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1 & 2):203–216, April 1987.

[BKR95] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.

[BL87] F. Bellegarde and P. Lescanne. Transformation orderings. In *Proc. 12th Coll. on Trees in Algebra and Programming (TAPSOFT)*, volume 249 of *Lecture Notes in Computer Science*, pages 69–80. Springer, 1987.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[Bou94] A. Bouhoula. *Preuves automatiques par récurrence dans les théories conditionnelles*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, 1994.

[Bou96] A. Bouhoula. Using induction and rewriting to verify and complete parameterized specifications. *Theoretical Computer Science*, 170(1-2), 1996.

[Bou97] A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, 1997.

[BPW89] T. B. Baird, G. E. Peterson, and R. W. Wilkerson. Complete sets of reductions modulo associativity, commutativity and identity. In N. Dershowitz, editor, *Proc. 3rd Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 29–44. Springer, 1989.

[BR90] W. Bousdira and J.-L. Rémy. On sufficient completeness of conditional specifications. In S. Kaplan and M. Okada, editors, *Proc. 2nd Intl. Workshop on Conditional and Typed Rewriting Systems*, Lecture Notes in Computer Science. Springer, 1990.

[BR93] A. Bouhoula and M. Rusinowitch. Automatic case analysis in proof by induction. In R. Bajcsy, editor, *Proc. 13th Intl. Joint Conference on Artificial Intelligence, Chambéry (France)*, volume 1, pages 88–94. Morgan Kaufmann, 1993.

[BR95a] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

[BR95b] A. Bouhoula and M. Rusinowitch. Spike: a system for automatic inductive proofs. In V. Alagar and M. Nivat, editors, *Proc. 4th Intl. Conference on Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*. Springer, 1995.

[Bro75] T. Brown. *A structured design-method for specialized proof procedures.* PhD thesis, California Institute of Technology, Pasadena, California, 1975.

[BT88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*, pages 82–90, 1988.

[BvEvL+87] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, M. J. Plasmeijer, and H. P. Barendregt. CLEAN - A language for functional graph rewriting. In Kahn, editor, *In Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA'87)*, number 274 in Lecture Notes in Computer Science, pages 364–384, Portland, Oregon, USA, 1987. Springer-Verlag.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1988. Japanese translation, 1991. Dutch translation, 1991. German translation, 1992.

[Cas98] Carlos Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3):263–293, September 1998.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. North Holland, 1996.

[CELM96] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[CG91] P.-L. Curien and G. Ghelli. On confluence for weakly normalizing systems. In R. V. Book, editor, *Proc. 4th Conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 215–225. Springer, 1991.

[CHL96] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[CHL96] P.-L. Curien, Th. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[Chr92] J. Christian. Some termination criteria for narrowing and E-narrowing. In D. Kapur, editor, *Proc. 11th Intl. Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 582–588. Springer, 1992.

[Chr93] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, February 1993.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cir00] Horatiu Cirstea. *Calcul de réécriture : fondements et applications.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.

[Cir99] Horatiu Cirstea. Specifying Authentication Protocols Using ELAN. In *Workshop on Modelling and Verification*, Besancon, France, December 1999.

[CK01] Horatiu Cirstea and Claude Kirchner. The rewriting calculus. *Logic Journal of the IGPL*, 2001. To Appear.

[CK97] Horatiu Cirstea and Claude Kirchner. Theorem Proving Using Computational Systems: The Case of the B Predicate Prover. In *Workshop CCL'97*, Schloβ Dagstuhl, Germany, September 1997.

[CK99] Horatiu Cirstea and Claude Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In Dov Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[CL87] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–160, 1987.

[CL92] E. A. Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In D. Kapur, editor, *Proc. 11th Intl. Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.

[CL96] Yves Caseau and François Laburthe. Introduction to the CLAIRE programming language. Technical report 96-15, LIENS Technical, September 1996.

[CM98] G. Cousineau and M. Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.

[Col88] Loïc Colson. Une structure de données pour le λ-calcul typé. Private Communication, 1988.

[Com86] H. Comon. Sufficient completeness, term rewriting system and anti-unification. In J. Siekmann, editor, *Proc. 8th Intl. Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1986.

[Com92] H. Comon. Completion of rewrite systems with membership constraints. In W. Kuich, editor, *Proc. ICALP'92*, volume 623 of *Lecture Notes in Computer Science*. Springer, 1992.

[CPH+85] G. Cousineau, L. C. Paulson, G. Huet, R. Milner, M. Gordon, and C. Wadsworth. *The ML Handbook*. INRIA, Rocquencourt, May 1985.

[DDHY92] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE computer society, 1992.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Der83] N. Dershowitz. Applications of the Knuth-Bendix completion procedure. Technical Report ATR-83(8478)-2, The Aerospace Corporation, El Segundo, Calif. 90245, 1983.

[Der85] N. Dershowitz. Computing with rewrite systems. *Information and Control*, 65(2/3):122–157, 1985.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.

[Der89] N. Dershowitz. Completion and its applications. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Vol. 2: Rewriting Techniques*, pages 31–86. Academic Press Inc., 1989.

[DFG+94] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. OPAL: Design and implementation of an algebraic programming language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures PLSA'94*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer-Verlag, March 1994.

[DH95] N. Dershowitz and C. Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, 1995.

[DHK95] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions, extended abstract. In Dexter Kozen, editor, *Proceedings of LICS'95*, pages 366–374, San Diego, June 1995.

[DHK98] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz`.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In Michael Maher, editor, *Proceedings of JICSLP'96*, Bonn (Germany), September 1996. The MIT press.

[DHLT87] M. Dauchet, T. Heuillard, P. Lescanne, and S. Tison. Decidability of the confluence of ground term rewriting systems. In *Proc. 2nd IEEE Symposium on Logic in Computer Science, Ithaca (N.Y., USA)*, pages 353–359, 1987.

[DJ90a] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[DJ90b] Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

[DJ90] Nachem Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Chapter 6*, pages 244–320. Elsevier Science Publishers B.V. (North-Holland), 1990.

[DJ91] N. Dershowitz and J.-P. Jouannaud. Notations for rewriting. *Bulletin of European Association for Theoretical Computer Science*, 43:162–172, 1991.

[DK98] Hubert Dubois and Hélène Kirchner. Actions and plans in ELAN. In *Proceedings of the Workshop on Strategies in Automated Deduction - CADE-15, Lindau, Germany*, pages 35–45, 1998.

[DK99] Hubert Dubois and Hélène Kirchner. Rule based programming with constraints and strategies. Technical Report 99-R-084, LORIA, Nancy, France, November 1999. ERCIM workshop on Constraints, Paphos (Cyprus).

[DO90] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.

[Dow94] G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.

[DS88] Nachem Dershowitz and G. Sivakumar. Goal-directed equation solving. In *Proc. 7th National Conference on Artificial Intelligence, St. Paul, (MN, USA)*, pages 166–170, 1988.

[Eke95] Steven Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[Eke96] Steven Eker. Fast matching in combinations of regular equational theories. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[Eva51] T. Evans. On multiplicative systems defined by generators and relations. In *Proc. Cambridge Philosophical Society*, pages 637–649, 1951.

[Fay79] M. Fay. First order unification in equational theories. In *Proc. 4th Workshop on Automated Deduction, Austin (Tex., USA)*, pages 161–167, 1979.

[FH83] F. Fages and G. Huet. Unification and matching in equational theories. In *Proceedings Fifth Colloquium on Automata, Algebra and Programming, L'Aquila (Italy)*, volume 159 of *Lecture Notes in Computer Science*, pages 205–220. Springer-Verlag, 1983.

[FM89] K. Fukuda and T. Matsui. Finding all the perfect matchings in bipartite graphs. Technical Report B-225, Department of Information Sciences, Tokyo Institute of Technology, Oh-okayama, Meguro-ku, Tokyo 152, Japan, 1989.

[FN97] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.

[Fri85] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *IEEE Symposium on Logic Programming, Boston (MA)*, 1985.

[Fri86] L. Fribourg. A strong restriction of the inductive completion procedure. In *Proc. 13th Intl. Colloquium on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 105–115. Springer, 1986.

[Gan89] H. Ganzinger. Order-sorted completion: the many-sorted way. In *Proc. Intl. Joint Conference on Theory and Practice of Software Development: Colloquium on Software Engineering*, Lecture Notes in Computer Science. Springer, 1989.

[GBT89] J. Gallier and V. Breazu-Tannen. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *16th Colloquium Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 1989.

[GD92] J. A. Goguen and R. Diaconescu. A short survey of order-sorted algebra. *EATCS Bulletin*, 49:121–133, 1992.

[GH78] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[Gie95] J. Giesl. Generating polynomial orderings for termination proofs. In J. Hsiang, editor, *Proc. 6th Intl. Conference on Rewriting Techniques and Applications (RTA'95)*, volume 914 of *Lecture Notes in Computer Science*, pages 426–431. Springer, 1995.

[GJM85] J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics for order-sorted algebra. In W. Brauer, editor, *Proc. 12th Intl. Colloquium on Automata, Languages and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 221–231. Springer, 1985.

[GKK90] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.

[GKK+87] J. A. Goguen, Claude Kirchner, Hélène Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[GM86] Joseph A. Goguen and José Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 259–363. Prentice Hall, 1986. An earlier version appeared in *Journal of Logic Programming*, 1(2):179–210, 1984.

[GM92] J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[GMW79] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York (NY, USA), 1979.

[Gog80] J. A. Goguen. How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type implementation. In W. Bibel and R. Kowalski, editors, *Proc. 5th Intl. Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 1980.

[Grä91] A. Gräf. Left-to-right tree pattern matching. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 323–334. Springer-Verlag, April 1991.

[Gra96] Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.

[GSHH92] Joseph A. Goguen, Andrew Stevens, Keith Hobley, and Hendrik Hilberdink. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992. Also in *Mechanized Reasoning and Hardware Design*, edited by C.A.R. Hoare and M.J.C. Gordon, Prentice-Hall, 1992, pages 69–86.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 333, Ravenswood Ave., Menlo Park, CA 94025, August 1988.

[Han93] M. Hanus. The integration of functions into logic programming: from theory to practice. Technical report, Max-Planck-Institut für Informatik, September 1993.

[Har89] T. Hardin. Confluence results for the pure strong categorical combinatory logic CCL: λ-calculi as subsystems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[HCS96] Fergus Henderson, Thomas Conway, and Zoltan Somogyi. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–54, October-December 1996.

[HD83] J. Hsiang and N. Dershowitz. Rewrite methods for clausal and non-clausal theorem proving. In *Proc. 10th Intl. Colloquium on Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 1983.

[HH82] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, 1982. Preliminary version in Proc. 21st Symposium on Foundations of Computer Science, IEEE, 1980.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973.

[HK95] M. Hermann and P. G. Kolaitis. Computational complexity of simultaneous elementary AC-matching problems. In J. Wiedermann and P. Hájek, editors, *Proceedings 20th International Symposium on Mathematical Foundations of Computer Science, Prague (Czech Republic)*, volume 969 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, August 1995.

[HKK89] M. Hermann, C. Kirchner, and H. Kirchner. Implementations of term rewriting systems. Technical Report 89-R-218, Centre de Recherche en Informatique de Nancy, 1989. To appear in *Computer Journal*, British Computer Society.

[HKK94a] C. Hintermeier, C. Kirchner, and H. Kirchner. Dynamically-typed computations for order-sorted equational presentations –extended abstract–. In S. Abiteboul and E. Shamir, editors, *Proc. 21st Intl. Colloquium on Automata, Languages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 450–461. Springer, 1994.

[HKK94b] C. Hintermeier, C. Kirchner, and H. Kirchner. Dynamically-typed computations for order-sorted equational presentations. Research report 2208, INRIA, Inria Lorraine, 1994. 114 pp., also as CRIN report 93-R-309.

[HKK94c] C. Hintermeier, C. Kirchner, and H. Kirchner. Order-sorted completion with dynamic types. In *Proc. 10th WADT – 6th Compass Workshop*, May 1994.

[HKLR92] J. Hsiang, H. Kirchner, P. Lescanne, and M. Rusinowitch. The term rewriting approach to automated theorem proving. *Journal of Logic Programming*, 14(1&2):71–99, 1992.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HL78] G. Huet and D. S. Lankford. On the uniform halting problem for term rewriting systems. Technical Report 283, Laboria, France, 1978.

[HL91a] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, I. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Chapter 11*, pages 395–414. The MIT Press, 1991.

[HL91b] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, II. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Chapter 12*, pages 415–443. The MIT Press, 1991.

[HO80] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

[HR86] J. Hsiang and M. Rusinowitch. A new method for establishing refutational completeness in theorem proving. In J. Siekmann, editor, *Proc. 8th Intl. Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 141–152. Springer, 1986.

[HR87] J. Hsiang and M. Rusinowitch. On word problem in equational theories. In T. Ottmann, editor, *Proc. 14th Intl. Colloquium on Automata, Languages and Programming*, volume 267 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 1987.

[HS86] J. Roger Hindley and Johnathan P. Seldin. *Introduction to Combinators and Lambda-calculus.* Cambridge University, 1986.

[HSC96] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Nineteenth Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.

[Hue72] G. Huet. *Constrained Resolution: A Complete Method for Higher Order Logic.* PhD thesis, Case Western Reserve University, 1972.

[Hue73] G. Huet. A mechanization of type theory. In *Proceeding of the third international joint conference on artificial intelligence*, pages 139–146, 1973.

[Hue76] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω.* Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980. Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977.

[Hul80a] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proc. 5th Intl. Conference on Automated Deduction (CADE'80)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.

[Hul80b] J.-M. Hullot. *Compilation de Formes Canoniques dans les Théories équationelles.* Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France), 1980.

[Hul80] J.-M. Hullot. *Compilation de Formes Canoniques dans les Théories équationelles.* Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France), 1980.

[JK86a] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proc. 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK86b] J.-P. Jouannaud and E. Kounalis. Proof by induction in equational theories without constructors. In *Proc. 1st IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pages 358–366, 1986.

[JK86] J.-P. Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK89] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82:1–33, 1989.

[JK91] J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[JKK83] J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Proc. Intl. Colloquium on Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer, 1983.

[JKKM92] J.-P. Jouannaud, C. Kirchner, H. Kirchner, and A. Mégrelis. Programming with equalities, subsorts, overloading and parameterization in OBJ. *Journal of Logic Programming*, 12(3):257–280, 1992.

[JLR82] J.-P. Jouannaud, P. Lescanne, and F. Reinig. Recursive decomposition ordering. In D. Bjørner, editor, *Formal Description of Programming Concepts 2*, pages 331–348. Elsevier Science Publishers B.V. (North-Holland), 1982.

[JM90] J.-P. Jouannaud and C. Marché. Completion modulo associativity, commutativity and identity (AC1). In A. Miola, editor, *Proc. DISCO'90*, volume 429 of *Lecture Notes in Computer Science*, pages 111–120. Springer, 1990.

[JO91] Jean-Pierre Jouannaud and Mitsuhiro Okada. Executable higher-order algebraic specification languages. In *Proceedings 6th IEEE Symposium on Logic in Computer Science, Amsterdam (The Netherlands)*, pages 350–361. IEEE, 1991.

[JO97] Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 28 February 1997.

[Jon94] Mark P. Jones. The implementation of the Gofer functional programming system. Technical Report Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, 1994.

[Jon96] Mark P. Jones. Hugs 1.3, The Haskell User's Gofer System: User Manual. Technical Report Report NOTTCS-TR-96-2, Department of Computer Science, University of Nottingham, England, 1996.

[Jou83] J.-P. Jouannaud. Confluent and coherent equational term rewriting systems. Applications to proofs in abstract data types. In G. Ausiello and M. Protasi, editors, *Proc. 8th Colloquium on Trees in Algebra and Programming*, volume 159 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 1983.

[JW86] J.-P. Jouannaud and B. Waldmann. Reductive conditional term rewriting systems. In M. Wirsing, editor, *3rd IFIP Conference on Formal Description of Programming Concepts, Ebberup (Denmark)*, pages 223–244. Elsevier Science Publishers B.V. (North-Holland), 1986.

[Kah87] G. Kahn. Natural semantics. Technical Report 601, INRIA Sophia-Antipolis, 1987.

[Kah87] G. Kahn. Natural semantics. Technical Report 601, INRIA Sophia-Antipolis, February 1987.

[Kap83] S. Kaplan. *Un langage de spécification de types abstraits algébriques.* Thèse de Doctorat de Troisième Cycle, Université d'Orsay, France, 1983.

[Kap84] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33:175–193, 1984.

[Kap87] S. Kaplan. Simplifying conditional term rewriting systems: Unification, termination and confluence. *Journal of Symbolic Computation*, 4(3):295–334, 1987.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[KB91] S. Krischer and A. Bockmayr. Detecting redundant narrowing derivations by the lse-sl reducibility test. In R. V. Book, editor, *Proc. 4th Conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 74–85. Springer, 1991.

[Kes93] Delia Kesner. *La définition de fonctions par cas à l'aide de motifs dans des langages applicatifs.* PhD thesis, Université de Paris XI, December 1993.

[Kir95a] H. Kirchner. On the use of constraints in automated deduction. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 128–146. Springer, 1995.

[Kir95b] H. Kirchner. Some extensions of rewriting. In H. Comon and J.-P. Jouannaud, editors, *Term Rewriting*, volume 909 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 1995.

[Kir99] H. Kirchner. *Term Rewriting*, chapter 9, pages 273–320. IFIP State-of-the-Art Reports. Springer, 1999. Report LORIA 99-R-098.

[KK99] Claude Kirchner and Hélène Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKM88] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ-3. In *Proc. 15th Intl. Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 1988.

[KKR90] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[KKR90] Claude Kirchner, Hélène Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[KKV95] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P.Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers, Chapter 8*, Lecture Notes in Computer Science, pages 131–158. The MIT Press, 1995.

[KKV95] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[KL82] S. Kamin and J.-J. Lévy. Attempts for generalizing the recursive path ordering. Technical report, Inria, Rocquencourt, 1982.

[KL91] E. Kounalis and Denis Lugiez. Compilation of pattern matching with associative commutative functions. In *16th Colloquium on Trees in Algebra and Programming*, volume 493 of *Lecture Notes in Computer Science*, pages 57–73. Springer-Verlag, 1991.

[Kli93] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[Klo90] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[Klo92] Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2, Chapter 1*, pages 1–116. Oxford University Press, 1992.

[KM01] Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[KM87] D. Kapur and D. R. Musser. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, 1987.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[KM98] Hélène Kirchner and Pierre-Etienne Moreau. Non-deterministic computations in ELAN. In J.L. Fiadeiro, editor, *Recent Developements in Algebraic Specification Techniques, Proc. 13th WADT'98, Selected Papers*, number 1548 in Lecture Notes in Computer Science, pages 168–182. Springer-Verlag, 1998. Report LORIA 98-R-278.

[KN85] D. Kapur and P. Narendran. A finite Thue system with decidable word problem and without equivalent finite canonical system. *Theoretical Computer Science*, 35:337–344, 1985.

[KNO90] Deepak Kapur, Paliath Narendran, and Friedrich Otto. On ground-confluence of term rewriting systems. *Information and Computation*, 86(1):14–31, 1990.

[Kou85] E. Kounalis. Completeness in data type specifications. In B. Buchberger, editor, *Proc. EUROCAL Conference*, volume 204 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1985.

[Kou90] E. Kounalis. Testing for inductive (co)-reducibility. In A. Arnold, editor, *Proc. 15th CAAP*, volume 431 of *Lecture Notes in Computer Science*, pages 221–238. Springer, 1990.

[KR90] E. Kounalis and M. Rusinowitch. Mechanizing inductive reasoning. In *Proc. American Association for Artificial Intelligence Conference, Boston*, pages 240–245. AAAI Press and MIT Press, 1990.

[KR98a] C. Kirchner and Ch. Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998.

[KR98b] Claude Kirchner and Christophe Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998.

[Kri94] S. Krischer. *Méthodes de vérification de circuits digitaux.* PhD thesis, Institut National Polytechnique de Lorraine, 1994.

[KvOvR93] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[Lan75] D. S. Lankford. Canonical inference. Technical report, Louisiana Tech. University, 1975.

[Lan79] D. S. Lankford. On proving term rewriting systems are noetherian. Technical report, Louisiana Tech. University, Mathematics Dept., Ruston LA, 1979.

[Les90] P. Lescanne. On the recursive decomposition ordering with lexicographical status and other related orderings. *Journal of Automated Reasoning*, 6:39–49, 1990.

[LLT90] A. Lazrek, P. Lescanne, and J.-J. Thiel. Tools for proving inductive equalities, relative completeness and ω-completeness. *Information and Computation*, 84(1):47–70, 1990.

[LM93] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[LM94] Denis Lugiez and Jean-Luc Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4):297–318, 1994.

[LR96] S. Limet and P. Réty. Conditional directed narrowing. In *Proc. 5th Intl. Conference on Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 637–640. Springer, 1996.

[LS93] C. Lynch and W. Snyder. Redundancy criteria for constrained completion. In C. Kirchner, editor, *Proc. 5th Conference on Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 1993.

[Lyn94] C. Lynch. Local simplification. In J.-P. Jouannaud, editor, *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 1994.

[Lyn95] C. Lynch. Paramodulation without duplication. In D. Kozen, editor, *Proc. 10th IEEE Symposium on Logic in Computer Science, San Diego (Ca., USA)*, pages 167–177. IEEE, 1995.

[Mar94] C. Marché. Normalised rewriting and normalised completion. In S. Abramsky, editor, *Proc. 9th IEEE Symposium on Logic in Computer Science, Paris (France)*, pages 394–403. IEEE, 1994.

[Mar96] Claude Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253–288, 1996.

[McC92] W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mil84] R. Milner. A proposal for standard ML. In *Proceedings ACM Conference on LISP and Functional Programming*, 1984.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen, Germany, December 1989*, volume 475 of *Lecture Notes in Computer Science*, pages 253–281. Springer-Verlag, 1991.

[MK98] Pierre-Etienne Moreau and Hélène Kirchner. A compiler for rewrite programs in associative-commutative theories. In *"Principles of Declarative Programming"*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998. Report LORIA 98-R-226.

[MM93] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, Computer Science Laboratory, SRI International, 1993.

[MN70] Z. Manna and S. Ness. On the termination of Markov algorithms. In *Proc. 3rd Hawaii Intl. Conference on System Science, Honolulu, Hawaii*, pages 789–792, 1970.

[MN90] U. Martin and T. Nipkow. Ordered rewriting and confluence. In M. E. Stickel, editor, *Proc. 10th Intl. Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 1990.

[Mor98] Pierre-Etienne Moreau. A choice-point library for backtrack programming. JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic, 1998.

[Mor99] Pierre-Etienne Moreau. *Compilation de règles de réécriture et de stratégies non-deterministes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France, June 1999. also TR LORIA 98-T-326.

[MR92] J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: the language BABEL. *Journal of Logic Programming*, 12(3):191–223, 1992.

[MuP96] MuPAD Group, Benno Fuchssteiner et al. *MuPAD User's Manual - MuPAD Version 1.2.2.* John Wiley and sons, Chichester, New York, first edition, march 1996. includes a CD for Apple Macintosh and UNIX.

[Mus80] D. R. Musser. On proving inductive properties of abstract data types. In *Proc. 7th ACM Symp. on Principles of Programming Languages*, pages 154–162. ACM, 1980.

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

[Nip89] T. Nipkow. Combining matching algorithms: The regular case. In N. Dershowitz, editor, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, volume 355 of *Lecture Notes in Computer Science*, pages 343–358. Springer-Verlag, April 1989.

[NP98] Tobias Nipkow and Christian Prehofer. Higher-order rewriting and equational reasoning. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume I: Foundations.* Kluwer, 1998.

[NR92a] R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In B. Krieg-Brückner, editor, *Proc. ESOP'92*, volume 582 of *Lecture Notes in Computer Science*, pages 371–389. Springer, 1992.

[NR92b] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In D. Kapur, editor, *Proc. 11th Intl. Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 1992.

[NR94] R. Nieuwenhuis and A. Rubio. AC-superposition with constraints: no AC-unifiers needed. In A. Bundy, editor, *Proc. 12th Intl. Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 545–559. Springer, 1994.

[NRS89] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7(3&4):295–318, 1989. Special issue on unification. Part one.

[NWE97] N. Nedjah, C.D. Walter, and E. Eldrige. Optimal left-to-right pattern-matching automata. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Proceedings 6th International Conference on Algebraic and Logic Programming, Southampton (UK)*, volume 1298 of *Lecture Notes in Computer Science*, pages 273–286. Springer-Verlag, September 1997.

[Obe62] A. Oberschelp. Untersuchungen zur mehrsortigen Quantorenlogik. *Mathematische Annalen*, 145(1):297–333, 1962.

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

[Oka89] Mitsuhiro Okada. Strong normalizability for the combined system of the typed λ calculus and an arbitrary convergent term rewrite system. In Gaston H. Gonnet, editor, *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation: ISSAC '89 / July 17–19, 1989, Portland, Oregon*, pages 357–363, New York, NY 10036, USA, 1989. ACM Press.

[Pad00] Vincent Padovani. Decidability of fourth-order matching. *Mathematical Structures in Computer Science*, 3(10):361–372, June 2000.

[Pad96] V. Padovani. *Filtrage d'ordre supérieur.* Thèse de Doctorat d'Université, Université Paris VII, 1996.

[Pag98] Bruno Pagano. X.R.S : Explicit Reduction Systems - A First-Order Calculus for Higher-Order Calculi. In Claude Kirchner and Hélène Kirchner, editors, *15th International Conference on Automated Deduction*, LNAI 1421, pages 72–87, Lindau, Germany, July 5–July 10, 1998. Springer-Verlag.

[Par97] Vincent Partington. Implementation of an Imperative Programming Language with Backtracking. Technical Report P9714, University of Amsterdam, Programming Research Group, 1997. Available by anonymous ftp from ftp.wins.uva.nl, file pub/programming-research/reports/1997/P9712.ps.Z.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Pet83] G. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM Journal of Computing*, 12(1):82–100, 1983.

[Pet90] G. E. Peterson. Complete sets of reductions with constraints. In M. E. Stickel, editor, *Proc. 10th Intl. Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 381–395. Springer, 1990.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages.* Prentice Hall, Inc., 1987.

[PJ96] Simon Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In *Proceedings of the European Symposium on Programming (ESOP'96)*, volume 1058 of *Lecture Notes in Computer Science*, Linkping, Sweden, January 1996. Springer-Verlag.

[Pla93] D. Plaisted. Equational reasoning and term rewriting systems. In D. Gabbay, C. Hogger, J. A. Robinson, and J. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 1*, pages 273–364. Oxford University Press, 1993.

[Plo77] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Pro01] Protheo Team. The ELAN home page. WWW Page, 2001. `http://elan.loria.fr`.

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[PvE93] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Qui59] W. V. Quine. On cores and prime implicants of truth functions. *American Math. Monthly*, 66:755–760, 1959.

[Red90] U. S. Reddy. Term rewriting induction. In M. E. Stickel, editor, *Proc. 10th Intl. Conference on Automated Deduction (CADE 10)*, volume 449 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 1990.

[Rém82] J.-L. Rémy. *Etude des systèmes de Réécriture Conditionnels et Applications aux Types Abstraits Algébriques*. Thèse de Doctorat d'Etat, Institut National Polytechnique de Lorraine, Nancy (France), 1982.

[Rét87] P. Réty. Improving basic narrowing. In P. Lescanne, editor, *Proc. 2nd Conference on Rewriting Techniques and Applications*, volume 256 of *Lecture Notes in Computer Science*, pages 228–241. Springer, 1987.

[Rin96] Ch. Ringeissen. Combining Decision Algorithms for Matching in the Union of Disjoint Equational Theories. *Information and Computation*, 126(2):144–160, May 1996.

[Rin97] Christophe Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997.

[Río93] A. Ríos. *Contributions à l'étude des λ-calculs avec des substitutions explicites*. Thèse de Doctorat d'Université, Université Paris VII, 1993.

[RKKL85] P. Réty, C. Kirchner, H. Kirchner, and P. Lescanne. Narrower: A new algorithm for unification and its application to logic programming. In J.-P. Jouannaud, editor, *Proc. 1st Conference on Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 1985.

[Rus87] M. Rusinowitch. *Démonstration automatique par des techniques de réécriture*. Thèse de Doctorat d'Etat, Université Henri Poincaré – Nancy 1, 1987. Also published by InterEditions, Collection Science Informatique, directed by G. Huet, 1989.

[Rus89] M. Rusinowitch. *Démonstration automatique-Techniques de réécriture*. InterEditions, 1989.

[RW69] G. A. Robinson and L. T. Wos. Paramodulation and first-order theorem proving. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 135–150. Edinburgh University Press, 1969.

[Sah91] D. Sahlin. Determinacy analysis for full prolog. In *Proceeding of the ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation*. ACM Press, 1991.

[Sch87] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*. PhD thesis, Universität Kaiserslautern (Germany), 1987.

[SCL70] J. R. Slagle, C. L. Chang, and R. C. T. Lee. A new algorithm for generating prime implicants. *IEEE Transactions on Computing*, 19(4):304–310, 1970.

[Sla74] J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[SNGM89] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer. Order-sorted equational computation. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Vol. 2: Rewriting Techniques*, pages 297–367. Academic Press Inc., 1989.

[Soc91] R. Socher-Ambrosius. Boolean algebra admits no convergent term rewriting system. In R. V. Book, editor, *Proc. 4th Conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 264–274. Springer, 1991.

[SPJ93] Patrick Sansom and Simon Peyton Jones. Generational garbage collection for Haskell. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA'93)*, Copenhagen, June 1993.

[ST85] H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization. In *Proceedings of the Second International Logic Programming Conference*, pages 200–207, Boston, Massachusetts, 1985.

[Ste90] J. Steinbach. AC-termination of rewrite systems — A modified Knuth-Bendix ordering. In H. Kirchner and W. Wechler, editors, *Proc. 2nd Intl. Conference on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 1990.

[Sti85] M. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1(4):333–355, 1985.

[Thi84] J.-J. Thiel. Stop losing sleep over incomplete data type specifications. In *Proc. 11th ACM Symp. on Principles of Programming Languages*, pages 76–82. ACM, 1984.

[vD96] A. van Deursen. An Overview of ASF+SDF. In *Language Prototyping*, pages 1–31. World Scientific, 1996. ISBN 981-02-2732-9.

[VdBdJKO00] M. Van den Brand, H.A.. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.

[vdBKO99] Mark G. J. van den Brand, P. Klint, and P. Olivier. Compilation and Memory Management for ASF+SDF. In *Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.

[vdBvDK+96] M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *AMAST '96*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[Vig94] L. Vigneron. Associative-commutative deduction with constraints. In A. Bundy, editor, *Proc. 12th Intl. Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 530–544. Springer, 1994.

[Vir96] P. Viry. Input/Output for ELAN. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[Vit94] Marian Vittek. ELAN*: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[Vit96] M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proc. 7th Intl. Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 1996.

[Vit96] Marian Vittek. A compiler for nondeterministic term rewriting systems. In Harald Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.

[vO90] Vincent van Oostrom. Lambda calculus with patterns. Technical report, Vrije Universiteit, Amsterdam, November 1990.

[Vor95] A. Voronkov. The Anatomy of Vampire: Implementing Bottom-up Procedures with Code Trees. *Journal of Automated Reasoning*, 15:237–265, 1995.

[Wad87] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14'th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.

[Wal92] U. Waldmann. Semantics of order-sorted specifications. *Theoretical Computer Science*, 94(1):1–33, 1992.

[War83] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Artificial Intelligence Center, 1983.

[WBK94] A. Werner, A. Bockmayr, and S. Krischer. How to realize LSE narrowing. In *Proc. 4th Intl. Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science. Springer, 1994.

[Wer93] A. Werner. A semantic approach to order-sorted rewriting. In C. Kirchner, editor, *Proc. 5th Conference on Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 1993.

[WG94] C.-P. Wirth and B. Gramlich. On notions of inductive validity for first-order equational clauses. In A. Bundy, editor, *Proc. 12th Intl. Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1994.

[WL93] P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993.

[Wol93] D. A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[Wol99] Stephen Wolfram. *The Mathematica Book*, chapter Patterns, Transformation Rules and Definitions. Cambridge University Press, 1999. ISBN 0-521-64314-7.

[WRCS67] L. Wos, G. A. Robinson, D. F. Carso, and L. Shalla. The concept of demodulation in theorem proving. *Journal of the ACM*, 14(4):698–709, 1967.

[YH90] H. Yokouchi and T. Hikita. A rewriting system for categorical combinators with multiple arguments. *SIAM Journal of Computing*, 19(1), February 1990.

[You89] J.-H. You. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7(3&4):319–342, 1989. Special issue on unification. Part one.

[ZKK88] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. Lusk and R. Overbeek, editors, *Proc. 9th Intl. Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 1988.