Equational logic and rewriting — Source link

Claude Kirchner, Hélène Kirchner

Institutions: French Institute for Research in Computer Science and Automation

Published on: 01 Jan 2014

Topics: Equational logic, Substructural logic, Higher-order logic, Intermediate logic and Many-valued logic

Handbook of the History of Logic
Vol. 9: History of Logic and Computation in the 20th Century
Chap. 8: Equational logic and rewriting

Claude Kirchner and Hélène Kirchner
Inria

May 7, 2014

Abstract

Handbook-History-Logic-Computation-EquationalLogic
Handbook-History-Logic-Computation-Rewriting

Contents

1 Introduction

Equational logic is the logic of equality, a concept that looks familiar and is often encountered, but that has actually different facets. In high-school mathematics, equality is a binary relation between objects meaning that two objects are identical, in the sense that replacement of one by the other in an expression does not change its value. An equality is written with the predicate symbol "$=$" and two expressions of the same nature, in mathematics usually numbers, vectors, functions, sets, matrices, ... Equalities are used in many different situations, either to abbreviate expressions (e.g. $\Delta = b^2 - 4ac$), to express the result of a calculus (e.g. $2 + 3 = 5$), to write identities (e.g. $E = mc^2$), to model a problem under the form of an equation (e.g. $x^2 + y^2 = z^2$) and to describe the axiomatic properties of a given structure (e.g. $x + y = y + x$) . It may happen that this variety of usage introduces some confusion.

If equality looks so familiar, it is maybe also because it has a very long history. It is generally admitted that the first printed occurrence of the $=$ sign is found in the book entitled *The Whetstone of Witte* by the mathematician and physician Robert Recorde (1510-1558) in 1557. The sign was found also in italian manuscripts earlier in the 16th century and was already used by egyptians to denote friendship. Equality could also be associated to the result of a calculation, i.e. mathematical determination of the amount or number of something. Not surprisingly, it is also a primary concept in computation, i.e. the action of mathematical calculation, and associated to mechanical calculators, invented by Blaise Pascal since 1642, followed by Leibniz in 1685 who gave a definition of equality intuitively stating that two entities are identical if and only if they have the same properties so that they are undistinguishable.

Logicians have addressed early the need for clarification and rigourous definition. Axiomatization of equality on natural numbers provided by Peano in 1889 can be considered as the first step to define equational logic. Later on, in the 20th century, with Russel, Tarski and Church, set and type theories were developed and also addressed equality. In set theory for instance, a set is completely characterized by its elements, although it may be defined by different properties. Formally, $\forall x, y,\ (x = y)$ iff $(\forall P,\ P(x)$ iff $P(y))$ where P denotes a predicate. However this definition needs a quantification on predicate and so does not belong to first-order logic. In set theory, its instance is the property of extensionality: is A and B are two sets, $A = B$ iff $(\forall x,\ x \in A$ iff $x \in B)$. The idea that types comes equipped with an equality relation was made explicit by Bishop in 1967 and in Martin-Löf's type theory in 1972. In Church's type theory, only one notion of equality exists: it expresses that two expressions denote the same object, either as a consequence of a definition or as the result of a more complex reasoning such as replacement of equals by equals. In that case the two expressions contain variables that are universally quantified. For instance $\forall x, y \in \mathit{Reals},\ (x + y)^2 = x^2 + y^2 + 2xy$. Intuitionistic type theory introduces a second notion called equality by definition, where two propositions are equal if every proof of one of them is also a proof of the other one.

The transition between logic and computer science is largely due to Alan Turing (1912-1954), by giving a formalisation of the concepts of "algorithm" and "computation" with the Turing machine. After Turing's famous work on the halting problem, showing the existence of a problem unsolvable by mechanical means, Church and Turing then proved that the lambda calculus and the Turing machine used in the halting problem were equivalent in capabilities, and subsequently provided a mechanical process for computation. With this mechanization and the need to give an operational (implementable) version of equality, while avoiding the halting problem intrinsically related to the symetrical nature of equality, the notion of rewriting as an oriented equality came up. Then rewrite relation and equality, studied as two different concepts, nevertheless have had interleaved developments, at the intersection of logic and computation, i.e. of proof and algorithm.

In this survey, we do not address higher order logics nor type theory, but rather restrict to first-order concepts. We focus on equational logic and its relation to rewriting logic and we consider their impact on automated deduction and programming languages.

We organize our view of equational logic and rewriting history along three lines of approach, corresponding more or less to time periods where the research has been especially active on these respective approaches:

- From 1935 to 1970, the **theory and model settings main period**, studying equational logics and their models, with the development of data types and algebraic specification languages;

- From 1970 to 2000, the **operational semantics period**, based on the study of rewriting on words or terms, with its use in main algebraic language implementations as well as automated deduction for a large class of equational theories;

- From 2000, the **logical framework period** with the formalisation of rewriting calculi and logics, covering different types of structures (words, terms, graphs, formulas, proofs) with types, sorts, conditions, constraints,... and abstracted in the concept of abstract reduction systems.

In this survey, our intention is to describe the evolution of ideas rather than to cover all places where equality and equational logic are useful and have been embedded. Therefore, we do not attempt to be exhaustive in the description of all the results but rather give our personal perception and vision of this field main historical steps. However, references to technical surveys are expected to provide detailed information to interested readers.

2 Equational logics and their models

At the beginning of the 20th century, the notions of abstract algebra, algebraic logic and universal algebra emerge in particular in the works of Alfred Tarski and Garrett

Birkhoff. Function symbols and equalities at the logical level are interpreted by operations on algebras.

In equational logic, formulas are built from the equality predicate and first-order terms, i.e. well-formed expressions built on a set $\mathcal{F}$ of function symbols with arity (the number of arguments) and a set $\mathcal{X}$ of variables. The set of terms is denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Variables of a term may be instantiated by substitutions, defined as as mappings σ from a subset of variables to terms and extended on terms by preserving their structure.

An *equational axiom* or *equality* is denoted $(\forall X, l = r)$ where X is the set of variables occurring in the terms l and r. From a set of axioms, new equalities can be deduced via inference rules. A deduction system for equational deduction is given in Figure 1.

Reflexivity

$$t = t$$

Symmetry

$$\frac{t = t'}{t' = t}$$

Transitivity

$$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}$$

Congruence For any $f \in \mathcal{F}_n$

$$\frac{t_i = t_i',\ i = 1 \dots n}{f(t_1, \dots, t_n) = f(t_1', \dots, t_n')}$$

Substitutivity. For each substitution σ

$$\frac{t = t'}{\sigma(t) = \sigma(t')}$$

Figure 1: Deduction rules for Equational Logic

Given a set of equational axioms E and a set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$, the *equational theory of* E, denoted $\mathcal{T}h(E)$, is the set of equalities that can be obtained, starting from E, by applying the inference rules given in Figure 1. We write

$$E \vdash s = t \text{ if } (s = t) \in \mathcal{T}h(E).$$

A more compact inference system is the so-called *replacement of equals by equals*: given a set E of axioms, we write $s \longleftrightarrow_E t$ if there is an equality $l = r$ (or $r = l$) in E, such that the subterm of s at position ω, denoted $s|_\omega$, is an instance $\sigma(l)$ of l for some substitution σ and if t just differs from s by containing $\sigma(r)$ at position ω, which is denoted as $t = s[\sigma(r)]_\omega$. The reflexive transitive closure of the above symmetric relation is denoted $\overset{*}{\longleftrightarrow}_E$. It is a congruence relation, the quotient of the set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ by $\xleftrightarrow{*}_E$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{X})/E$ and the equivalence class of a term t is denoted $\langle t\rangle_E$.

The following theorem due to Birkhoff [23] states the equivalence between the two inference systems for equality deduction:

$$E \vdash s = t \text{ iff } s \xleftrightarrow{*}_E t.$$

Models of equational logic are $\mathcal{F}$-algebras, that are non-empty sets (called the carriers), with operations interpreting the symbols in $\mathcal{F}$. Mappings preserving the symbol interpretations are called homomorphisms.

In a non-empty class $\mathcal{C}$ of $\mathcal{F}$-algebras, there may exist two particularily interesting algebras: one is an *initial algebra* $\mathcal{I}$ such that for any algebra $\mathcal{A}$ in $\mathcal{C}$, there exists a unique homomorphism $\phi : \mathcal{I} \to \mathcal{A}$. The other one is the *free algebra* $\mathcal{T}$ over a set of variables $\mathcal{X}$ such that: for any algebra $\mathcal{A}$ in $\mathcal{C}$ and any assignment $\nu : \mathcal{X} \to \mathcal{A}$, ν can be extended by a unique homomorphism $\phi : \mathcal{T} \to \mathcal{A}$.

Note that by definition, the initial algebra can also be seen as the free algebra over the empty set of variables. When it exists, it is unique up to an isomorphism.

Given a set E of equality axioms, an algebra $\mathcal{A}$ is a *model* of E if for every axiom $s = t$ in E, for any assignment ν of the variables from s and t into the carrier of $\mathcal{A}$, $\nu(s) = \nu(t)$. We also say that the equality $s = t$ is *valid* in $\mathcal{A}$, and this is denoted by $\mathcal{A} \models s = t$. $\mathcal{M}od(E)$ denotes the set of models of E. The fondamental theorem due to Birkhoff [23] relates models and equational deduction: for any set of axioms E, for any terms $s, t \in \mathcal{T}(\mathcal{F},\mathcal{X})$,

$$\mathcal{M}od(E) \models s = t \text{ iff } \mathcal{T}(\mathcal{F},\mathcal{X})/E \models s = t \text{ iff } E \vdash s = t \text{ iff } s \xleftrightarrow{*}_E t$$

i.e. validity in the models is equivalent to replacement of equals by equals, therefore a semantic check can be achieved completely syntactically. We write $s =_E t$ iff $\mathcal{M}od(E) \models s = t$. Thanks to these results, the notations $s =_E t$ and $s \xleftrightarrow{*}_E t$ may be used interchangeably.

In the class of algebras that are models of E, the free algebra over $\mathcal{X}$ is (isomorphic to) the algebra $\mathcal{T}(\mathcal{F},\mathcal{X})/E$. In the class of algebras that are models of E, the initial algebra is (isomorphic to) the algebra $\mathcal{T}(\mathcal{F})/E$.

When only arity of functions is specified, it is easy to build terms such as $divide(2, 0)$ that do not have interpretations. By introducing a type Nat for natural numbers and Nat^+ for non-zero natural numbers, and by typing the function $divide$ with the rank $Nat, Nat^+ \mapsto Nat$, such written terms do not satisfy the typing constraint. Sorts have been introduced for classifying terms and typing arguments and results of functions. In a *many-sorted signature*, a function f has a rank (or type) $f : s_1 \dots s_n \mapsto s$, with s_i, s in a set of sorts S and variables are typed too, for instance $x : s$ means that variable x has sort s. Many-sorted terms are built on many-sorted signatures and classified according to their sorts. Many-sorted algebras have carriers corresponding to each sort and operations with sorted arguments.

Deduction rules for equational deduction given in Figure 1 generalize to the many-sorted framework provided there is no *empty* sort. Goguen and Meseguer made a precise analysis of the possible problems that may arise when this hypothesis is not satisfied [115]. For keeping valid the deduction rules of Figure 1 for many-sorted logic, only models with non-empty sorts should be considered.

Such type structure supports conceptual clarity and detection of many errors. However, implementing strong typing in many-sorted logic is quite rigid and lacks the expressive power needed for handling errors and partiality, see the survey by Mosses and all of different approaches to address these problems [117]. One of them, initiated in 1977 by Goguen leads to [62], where an order-sorted type structure is proposed, making many seemingly partial or problematic functions total and well-defined on the right subsort. This order-sorted type structure is provided by a partial ordering on the set of sorts that is interpreted as set inclusion in order-sorted algebras. For instance, a subsort relation `Nat < Int` is interpreted as the inclusion $N \subseteq Z$ of the naturals into the integers in the standard model. In addition, operator symbols such as _+_ may have overloaded declarations, for instance `_+_: Nat Nat -> Nat`, `_+_: Int Int -> Int`, that are required to yield the same result when restricted to arguments of the same subsorts. Overloading is a syntactical facility for handling operators defined on different subsets that may intersect, for achieving a kind of polymorphism. Moreover such operators are in general partial functions, which is expressed thanks to the subsort relationship.

Although all basic results of equational logic generalize to the many-sorted case, order-sorted deduction is more subtle [64]. For example, replacement of equals by equals and term rewriting require a careful analysis that was initiated in [60] and is further developed in [78, 94].

All these concepts have been used to define the semantics of several specification and programming languages. Let us mention two of them.

The OBJ programming language originates from Goguen's pioneering work at UCLA [63, 65] in the late seventies, later fully developed at SRI International by Goguen, Meseguer and several invited collaborators [56, 61, 80, 93]. OBJ is algebraic programming language whose latest versions (OBJ-2 [56] and OBJ-3 [61]) are based on order-sorted equational logic. Programs are order-sorted equational specifications and computation is order-sorted equational deduction [94].

In the late 1990's, the CoFI initiative has been created and sponsored by the IFIP working group WG1.3. It was an open collaborative effort to produce a Common Framework for Algebraic Specification and Development, together with the so-called Common Algebraic Specification Language Casl, a general-purpose specification language based on first-order logic [6] that also supports partial functions and subsorting. Casl has been designed with the aim to subsume many existing specification languages and to implement the most important features of algebraic specifications.

3 Rewriting techniques

From about 1970, the need for an operational version of equational deduction came up in two different communities, in formal specifications and functional programming on one hand, in automated reasoning on the other. With the emergence of equationally specified abstract data type, the question was to get executable specifications and to define operational semantics of function evaluation. In automated reasoning, the problem was to mechanize equational deduction and find decision procedures for equational theories. Giving a common solution to these two questions also provides a bridge between programming language theory and program verification.

In both situations, automating equational deduction needs to compute which is the right axiom to be applied at each step, in which direction, and possibly to backtrack. The idea of rewriting is to suppress the need for backtracking, first by using oriented axioms from left to right only, second by giving enough oriented axioms to have the same deduction power than originally. For instance, group theory is defined by three axioms (neutral element, inverse and associativity) but deciding equality in group theory needs ten oriented axioms (equivalent to the three previous ones), first discovered in the pioneer work of Knuth and Bendix [100]. We now recall the basic notions of rewrite systems, together with properties that rewriting must satisfy in order to provide a decision procedure for equational theories.

3.1 The initial concept

With the purpose to operationalize equation deduction, term rewriting was first proposed by Evans [51], then by Knuth and Bendix [100]. The original purpose was to generate *canonical term rewriting systems* which can be used as decision procedures for proving the validity of equalities in first-order equational theories.

The central idea of rewriting is to impose directionality in the use of equalities. So a rewrite rule is an ordered pair of terms $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, denoted $l \Rightarrow r$, where it is often required that l is not a variable and all variables of r are also variables of r. The terms l and r are respectively called the left-hand side and the right-hand side of the rule. A *rewrite system* is a (finite or infinite) set of rewrite rules.

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. Given a rewrite system R, a term t in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ rewrites to a term t' if there exists a rewrite rule $l \Rightarrow r$ of R, a position ω in t and a substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t[\sigma(r)]_{\omega}$. This is denoted $t \longrightarrow_{\omega, l \Rightarrow r, \sigma} t'$ or simply $t \rightarrow t'$ when there is no need to make precise which rewrite rule is used nor where. A subterm $t_{|\omega}$ where the rewriting step is applied is called *redex*. A term that has no redex is said to be irreducible for R or to be in *R-normal form*. The induced rewrite relation on terms is the reflexive transitive closure of $\longrightarrow$. From a logical point of view, the deduction rules are similar to equational logic (cf. deduction rules in Figure 1) without the **Symmetry** rule and with the rewrite relation instead of equality. Surveys on term rewriting include [76, 45, 8, 99, 127, 9, 132, 123, 22].

3.2 Termination

The normalizing property of a rewrite system (i.e. every term has a normal form) is not enough if infinite computations of normal forms must be avoided. Indeed, a term may have a normal form and nevertheless an infinite rewriting derivation. So a stronger property is often needed, namely the *termination property*. In general, it is undecidable whether a rewrite system is terminating. The idea of the undecidability proof is the following: given any Turing machine $\mathcal{M}$, there exists a rewrite system $R_{\mathcal{M}}$ such that $R_{\mathcal{M}}$ terminates for all terms iff $\mathcal{M}$ halts for all input tapes. Since it is undecidable if a Turing machine halts uniformly, it is also undecidable if rewrite systems terminate [43]. This shows that first-order term rewriting has the expressive power of any Turing machine that can even be encoded using a single rewrite rule left linear rewrite rule as shown by Dauchet [38].

A first approach, developped from the late 70th and landmarked by the works of Dershowitz, is to relate termination with orderings [41] and rewrite derivations to decreasing chains in a well-founded ordered structure. A *reduction ordering* is a well-founded ordering on terms closed under congruence and substitution. Then termination of rewriting can be proved by just comparing left and right-hand sides of rules: a rewrite system R over the set of terms is terminating iff there exists a reduction ordering $>$ such that each rule $l \Rightarrow r \in R$ satisfies $l > r$. To take advantage of known ordered structures, terms may be interpreted, for instance by polynomials [129].

Another approach, emerging in the early 80's, is the notion of *simplification ordering*. In addition to be reduction orderings, simplification orderings enjoy the so-called *subterm property*: any term is greater than any of its subterms. If the set $\mathcal{F}$ of operator symbols is finite, a rewrite system R is terminating if R is simply terminating [41]. Simplification orderings can be built from a well-founded ordering on the function symbols $\mathcal{F}$ called a precedence. Examples are the *multiset path ordering* [83], also called *recursive path ordering*, the lexicographic path ordering or the recursive decomposition ordering [106]. Simple termination is reviewed in [116].

During the 80's and 90's, many orderings and termination proof techniques have been developed: Knuth-Bendix orderings, forward closures, semantic interpretations, transformation ordering, dummy elimination, semantic labelling. Comprehensive surveys on termination are [44, 137, 138, 130].

Then, since 2000, the interest in termination was renewed with the dependency pairs approach introduced by Aart and Giesl in [5], an automated method where left-hand sides are compared with special subterms of the right-hand sides to capture forward closure. A lot of efforts have been devoted to make the method faster, more powerful and able to handle termination of functional programs. Several systems for automated termination proofs have been developed, such as CiME [37], TTT [68], AProVE [57], TORPA [139], and integrated in software development tools that analyze the termination of programs in programming languages like Java, C, Haskell, and Prolog.

Considering termination of programming languages also led to study termination of rewriting under evaluation strategies. The dependency pairs approach has been adapted to such cases and termination of rewriting under strategies is also addressed in [58].

To complete this brief historical panorama, modular termination techniques have to be mentioned. In complex proofs of termination, different orderings are combined. Usual combinations are multiset extension or lexicographic combination. Another simple idea would be to simply combine two terminating rewrite systems by taking their union. But in general, as shown by Toyama [134], the union of two terminating rewrite systems is not terminating, even if the function symbols are disjoint. Many results about the modularity of the termination property have been studied in particular by Gramlich and can be found for instance in [67].

3.3 Confluence and Completion

By definition a function is expected to always return the same value when the function is applied to the same arguments. When this function is computed with rewrite rules, the property required for the rewrite system is therefore the uniqueness of the normal form for any term. This is crucial when computing normal forms must be independent of the strategy of rule application. Uniqueness of the normal form is implied by adding to termination another property called *confluence*. A rewrite system is confluent when two rewrite sequences beginning from the same term can always be extended to end with the same term. Although undecidable in general, confluence is decidable for terminating finite rewrite systems.

Confluence (which may be defined as an abstract property of relations, not specifically of rewriting relations) is equivalent to the *Church-Rosser property* that gives the relation between equational proofs and rewrite proofs: given a Church-Rosser rewrite system R, every equational theorem $t =_R t'$ has a *rewrite proof* $t \downarrow_R t'$, meaning that t and t' both rewrite to a same term t'', thus have the same normal form if any. Assuming termination, confluence is equivalent to local confluence [119], itself equivalent to the convergence of *critical pairs* [100, 72]. Critical pairs characterize minimal conflicts that can happen when two rewrite rules apply to a same term and overlap each other. Then the term can be rewritten in two different ways that may or may not have a common normal form.

A *completion* procedure is aimed at building a Church-Rosser and terminating rewrite system from a set of equalities. Completion orients equalities into rewrite rules, computes critical pairs, and keeps terms in normal form for the current set of rewrite rules. In [13, 11], completion is described through inference rules with a fair strategy, which transform a set of equalities E and a set of rewrite rules R.

Completion is initialized with a given set of equalities in E and an empty set of rules R. It also requires a well-founded ordering on terms, used to determine in which direction an equality must be oriented. A completion process has three possible outcomes: either it terminates, or fails on an unorientable equality, or diverges, that is, generates infinitely many new rules.

When the completion ends up with an empty set of equalities in E and a rewrite system in R that by construction is then locally confluent and terminating, the validity of an equational theorem $(t = t')$ w.r.t. the initial set of equalities is decidable by reducing both terms to their normal forms and by checking the syntactic equality of the results, i.e. $t \downarrow_R t'$.

The completion procedure fails when a non-orientable critical pair is generated. In this case, nothing can be said on the set of rewrite rules generated so far, except that they are equational consequences of the initial set of equalities in E. Practical implementations of a completion procedure often postpone the non-orientable equality, with the hope that a further generated rule will simplify it.

If the completion does not terminate and generates an infinite set of rewrite rules, it can be used as a semi-decision procedure for proving the validity of a theorem: the theorem $(t = t')$ is valid iff there exists a step i and thus a set of rewrite rules R_i such that $t \downarrow_{R_i} t'$.

In 1986, following Bachmair, Dershowitz and Hsiang works [13], completion has been explained as a proof simplification process, where each inference rule decreases the complexity of some equational proofs. The minimal (i.e. less complex) proofs in this setting are the rewrite proofs, i.e. those of the form $t \downarrow_R t'$. Interestingly, the completion concept is indeed independent of rewriting as shown much later by Dershowitz and Kirchner [46].

3.4 Various extensions of Rewriting

Various extensions of the rewriting relation on terms exist. The two first notions, ordered rewriting and class rewriting, emerged from the problem of permutative axioms like commutativity that can be applied indefinitely at the same position.

The first proposed solutions initiated by Lankford and Ballantyne [103] followed by the seminal papers of Huet [72] and of Peterson and Stickel [125] amounted to define a rewrite relation on equivalence classes of terms and to simulate it by another rewrite relation on terms that transforms a representative element of the equivalence class. In some cases, the problematic non-orientable axioms can be built in the matching process which becomes equational matching. This requires the elaboration of a new abstract concept, namely the notion of *rewriting modulo a set of equalities* in which the matching takes into account non-oriented equalities. Adequate notions of *confluence and coherence modulo* a set of equalities [72, 79] have been defined for this kind of rewriting relation. A *class rewrite system* is defined by a set R of rewrite rules and a set A of equality axioms. It is *Church-Rosser modulo* A if any equational theorem deduced from $R \cup A$ has a rewrite proof using rewriting in equivalence classes modulo A, defined as the compound relation $=_A \circ \rightarrow_R \circ =_A$. The class rewrite system is *terminating modulo* A if $=_A \circ \rightarrow_R \circ =_A$ terminates. This general setting introduced in full generality by

Jouannaud and Kirchner [81] allows to define equational completion procedures generalizing the standard case [125, 81, 12], but they must be carefully set-up to minimize sources of inefficiency. Typical examples handled by such a method are theories with associativity and commutativity axioms [125, 91, 77]. More recently in deduction modulo has pioneered by Dowek, Hardin and Kirchner [50], not only terms but also propositions are identified modulo a congruence and this idea gave rise to a new generation of higher-order proof assistants.

Another approach of the same problem is taken in the notion of *ordered rewriting* which appeared in [14]. There, only orientable instances of axioms can be used in the rewrite relation.

Conditional rewriting started from a quite different motivation issued from abstract data types and algebras with partial functions and exceptions, and this algebraic point of view was the base of earlier approaches [85, 141, 19]. In conditional rewriting, the rule application requires the satisfiability of its condition. However to be able to associate with a conditional rewrite system a decidable and terminating reduction relation, it is necessary to provide a reduction ordering to compare terms involved in the left and right-hand sides and in the condition of a conditional rewrite rule [84, 86, 48]. Then conditional rewriting has been shown to provide a computational paradigm combining logic and functional programming [53, 49, 66]. Only later the connection between conditional equalities and equational Horn clauses has been exploited [140, 15, 16].

Rewriting with constraints emerged in the 90's [96, 92, 98] as a unified way to cover the previous concepts by looking at ordering and equations as symbolic constraints on terms. Even further, it provides a framework to incorporate disequations, built-in data types and sort constraints [33].

As will be shown later on, all these rewrite relations can be studied as abstract reduction systems. Note also that Church-Rosser properties have been adapted and generalized for these different rewrite relations.

4 Rewriting and/for equality in theorem proving

Reasoning about equality has been one of the most challenging problems in automated deduction. In the past fifty years, a number of methods have been proposed. In this section, we focus on the contribution of the term rewriting techniques to first-order theorem proving.

The term rewriting approach to theorem proving is unique in several aspects. First, for equational theories enjoying a Church-Rosser property, rewriting provides a (semi-)decision procedure. As such, a rewrite proof procedure can be embedded in more general provers and combined with other proof methods. But even without this property, a simplification process with valid rewrite rules can help to reduce the search space of the provers. Second, the analogy of the completion process with saturation methods in theorem proving and with Prolog theorem provers has given rise to efficient proof by consistency or by refutation procedures. Third, it is one of the few methods which can be applied to a variety of problem domains. In addition to the validity problem of equational logic, it has also been applied to inductive theorem proving, first-order theorem proving, unification theory, geometry theorem proving.

4.1 Rewriting in theorem proving

From the operational point of view, term rewriting provides a *forward chaining method* in automated deduction. In other words, its main inference mechanism deduces new lemmas from known theorems, rather than reduces the intended problem into sub-problems and tries to solve them separately. Forward chaining methods are usually not efficient due to the large number of theorems produced which makes the search space unmanageable. As a tradeoff, forward chaining methods usually do not require backtracking, which is necessary in most backchaining methods. In fact, term rewriting is one of the very few successful forward chaining methods. The problem of space explosion is handled in term rewriting through two techniques: a notion of *critical pairs*, which tries to find only "useful" lemmas, and more importantly, a notion of *simplification*. Basically, simplification replaces current terms or formulas by others which are logically equivalent but "smaller" according to some well-founded ordering. Since the ordering is well-founded, simplification cannot go on indefinitely.

It has been demonstrated through various implementations and experiments that simplification is indeed an effective way of controlling the search space. In addition to finding complete sets of rewrite rules, other notable problems have been solved using term rewriting including the one-axiom group theory [105], the commutativity problem of rings [131], Boolean algebra problems [69, 54], SAM's lemma in lattice theory [140], Moufang identities of alternative rings [2]. In [70], a survey of the use of term rewriting in automated theorem proving for various systems available in the 1990's is given. Rewriting techniques also contribute to one of the main successes of automated theorem proving, namely the fully automated solution, obtained by McCune and his system Otter in 1996, of the Robbins problem that had challenged mathematicians for over sixty years [113].

In a different community, interactive proof systems such as PVS [124], Isabelle [122], or Coq [39] have also extensively used rewriting techniques through tactics to handle the automated part of reductions.

4.2 Completion as a saturation process

The completion procedure must be supplied with a well-founded ordering used to determine in which direction an equality must be oriented. Even when such an ordering is provided, the procedure may fail to find a confluent and terminating set of rules though one exists [47]. An *ordered (or unfailing) completion* procedure does not stop with a non-orientable equality and may terminate with a non-empty set of equalities. This amounts to work with the notion of ordered rewriting previously defined and to adapt the Church-Rosser property [14]. Ordered completion provides also a semi-decision procedure for equational problems and so can be used as a refutationally complete equational theorem prover. As such, it is possible to prove existentially quantified theorems by negating them and deriving a contradiction by ordered completion [71]. This idea has been further explored later on, in the context of automated theorem provers for first-order clause logic with equality, where the paramodulation inference rule was used to deal with equality. Bachmair and Ganzinger present in [16] the superposition calculus that generalizes the previous idea to first-order logic and provides a refinement of paramodulation. In the same vein, Nieuwenhuis and Rubio address in [121] equational theorem proving with constrained formulas and explain the connection with paramodulation based theorem proving.

4.3 Rewriting and equality for inductive reasoning

Some proofs of properties on classical data structures, such as integers, require *induction*. An *inductive theorem* is an equality that is true in the initial model of the axioms. Three main approaches have been developed for mechanizing inductive proofs, two of them using rewriting techniques.

4.3.1 Explicit induction.

The first one, *explicit induction*, is used in proof assistants, for instance Nqthm-ACL2 [89], Coq [21], Isabelle [122] or Inka [7]. Explicit induction uses structural induction on data types and *induction rules* as for example Peano induction. These forms of induction are in fact subsumed by the single general schema of Noetherian induction, called *noetherian induction principle*. It is based on a well-founded relation $<$ on a set τ, so that there is no infinite decreasing sequence of elements in τ. The *Noetherian induction principle* states that if, for any element $x \in \tau$, a proposition P holds for x whenever it holds for all elements $\underline{x}$ such that $\underline{x} < x$, then P holds for all $x \in \tau$. Mechanizing proof by induction [29] is hard due to the intrinsic difficulty of finding the most convenient inductive rule to show a given conjecture. Indeed there may be a huge variety of possible noetherian relations and choosing an appropriate induction rule introduces a first branching point in the search space. Furthermore, such proofs involve in general two tasks: generalizing the induction formula and introducing an intermediate lemma. This is why user interaction and expertise is needed to develop an inductive reasoning in proof assistants.

4.3.2 Induction by consistency.

The second approach is based on the notion of consistency: a set of first-order axioms $\mathcal{A}$, assumed here to be a set of equational clauses, is consistent if it has a non-trivial model. *Induction by consistency* (or *inductionless induction*), roughly works as follows: given a consistent set of axioms $\mathcal{A}$ and a set of conjectures $\mathcal{E}$, we add $\mathcal{E}$ to $\mathcal{A}$ and run a deduction engine until one gets a saturated and consistent set of clauses (i.e. on which no more deduction rule applies). Historically, this deduction engine was given by the Knuth-Bendix completion procedure [100] in the pioneer works of Musser [118], Goguen [59], Lankford [102], Huet and Hullot [73]. So it worked when $\mathcal{A}$ and $\mathcal{E}$ were both sets of equalities. The principle of a *proof by consistency* is to assume the validity of the intended inductive theorem and show that there no contradiction or inconsistency is generated with the axioms. The definition of a contradiction has also evolved in various ways: in 1980, in work of Musser [118] and independently Goguen [59], an inconsistency is the equality between boolean terms $(true = false)$. This approach assumes an axiomatization of booleans and an equality predicate eq_s for each sort of data s. This also requires that any expression $eq_s(t, t')$ reduces either to $true$ or to $false$. In 1982, Huet and Hullot showed how the notion of constructors (function symbols with which data are built) allows dropping the requirements about the equality predicate [73], provided there is no relation between constructors. An inconsistency is then just an equality between two different terms built only with constructors. When there exists a confluent and terminating rewrite system for $\mathcal{A}$, the method can take advantage of the existence of free constructors, characterized as a subset $\mathcal{C}$ of $\mathcal{F}$ such that $\mathcal{T}(\mathcal{C})$ is exactly the set of normal forms of $\mathcal{T}(\mathcal{F})$. In this case, every non-constructor operator must be *completely defined* [101, 42, 133, 104], which means that any ground term containing this operator is reducible. The latter property is decidable for a confluent and terminating rewrite system. In 1986, Jouannaud and Kounalis introduced the concept of *ground reducibility* [82] of a term, meaning that any instantiation of its variables by ground terms is reducible. Their approach allows handling relations between constructors. Ground reducibility is decidable for finite rewrite systems [87, 126] and is EXPTIME-complete in the general case [35].

The completion procedure attempts to complete the initial system $\mathcal{A} \cup \mathcal{E}$ by iteratively adding new equalities computed with the critical pairs mechanism. One main problem is that such a completion often generates infinitely many critical pairs. Fribourg [55] first observed that only overlaps of axioms on conjectures are necessary: it is the so-called “linear strategy”. Moreover, whenever $\mathcal{A}$ can be turned into a ground convergent and sufficiently complete rewrite system, overlapping can be performed at specific positions. A further improvement is to use ordered completion to avoid failure due to a non-orientable inductive theorem like commutativity. The method, called in this case *unfailing proof by consistency* [10, 11] is *refutationally complete*: it refutes any equality which is not an inductive theorem. For that, it detects any *provably inconsistent* equality, i.e. any equality $(s = t)$ which satisfies either $s > t$ (in the reduction ordering $>$) and s is not inductively reducible, or $(s = t)$ is not inductively reducible (i.e. neither s nor t is inductively reducible). Inference rules for proof by consistency are given in [10] and work with a confluent and terminating rewrite system R that describes the equational theory, a set C of conjectures to be proved and a set L of inductive lemmas. (Inductive lemmas are equational theorems $(s = t)$ such that $\sigma(s) =_R \sigma(t)$ for any ground substitution σ). The procedure adds in C new conjectures to be proved and obtained by computation of critical pairs obtained by superposition of rules in R into conjectures in C. Once a conjecture has been proved valid, it can be deleted from C, then added to L. It is also possible to introduce in L inductive lemmas given by the user or produced by another system. Conjectures are simplified using either rules of R, lemmas of L or other smaller conjectures of C. Finally, a refutation is produced when a provably inconsistent conjecture is detected.

Comon and Nieuwenhuis [36] gave a more general view of the deduction system, without restricting the conjectures to be equalities. The interested reader may refer to [34, section 1.3] for all relevant references on this approach.

4.3.3 Implicit induction by rewriting.

The third approach, *implicit induction* (or *induction by rewriting*), is used in automated theorem provers like Spike [26] or RRL [88]. The main idea of implicit induction is as follows: given a terminating rewrite system, the corresponding rewrite relation is noetherian and can be used for induction. Reddy [128] provided a method to prove equalities in this way. Proving an inductive conjecture amounts to proving some specific instances. Each one is simplified by either a rule or a smaller instance of the conjecture until an identity is obtained. The pragmatic advantage of his method is that it does not need to explicitly check that the inductive hypothesis is applied to smaller terms. In [128], the relationship between this new method and the inductionless induction procedures is clarified. Bouhoula and Rusinowitch [27] designed a proof technique which works in conditional rewrite systems and applies to non-Horn clauses as well: it is the so-called *test set induction*. It works by computing an appropriate implicit induction scheme called a test set and then applies a refutation principle using proof by consistency techniques. Their method is refutationaly complete and can refute false conjectures even in the cases where the functions are not completely defined. It has been implemented in the prover Spike. The implicit induction techniques have been also used for induction modulo associativity and commutativity in [20] and [4].

In [40, 95], a bridge between explicit and implicit induction is provided by a proof search mechanism for inductive proofs, relying on the deduction modulo approach [50]. A semi-decision procedure, which is refutationaly correct and complete, comes along with a constructive proof of soundness which associates a proof in deduction modulo to every successful proof of the procedure. This feature opens the door to the procedure's integration as a tactic in proof assistants that require some proof witness like Coq.

5 Universal power of Rewriting

Since the 80s, many aspects of rewriting have been studied in automated deduction, programming languages, equational theory decidability, program or proof transformation, but also in various domains such as chemical or biological computing, plant growth modelling, security policies, etc. Faced to this variety of applications, the question arises to understand rewriting in a more abstract way, especially as a logical framework able to encode different logics and semantics. Discovering the universal power of rewriting, in particular through its matching and transformation power, led first to the emergence of Rewriting Logic and Rewriting Calculus.

On the other hand, with the development of rewrite frameworks and languages, more and more reasoning systems have been modeled, for proof search, program transformation, constraint solving, SAT solving. It then appeared that deterministic rule-based computations or deductions are often not sufficient to capture complex computations or proof developments. A formal mechanism is needed, for instance, to sequentialize the search for different solutions, to check context conditions, to request user input to instantiate variables, to process subgoals in a particular order, etc. This is the place where the notion of strategy comes in and this led to the design and study of strategy languages and semantics.

More recent approaches consider that rules describe local transformations and strategies describe the control of rule application. Most often, it is useful to distinguish between rules for computations, where a unique normal form is required and where the strategy is fixed, and rules for deductions, in which case no confluence nor termination is required but an application strategy is necessary. Due to the strong correlation of rules and strategy in many applications, claiming the universal character of rewriting also requires the formalisation of its control.

5.1 Matching and transformation power

A first step is to note that in all its applications, rewriting definitions have the same basic ingredients. Rewriting transforms syntactic structures that may be words, terms, propositions, dags, graphs, geometric objects like segments, and in general any kind of structured objects. Transformations are expressed with patterns or rules. Rules are buit on the same syntax but with an additional set of variables, say $\mathcal{X}$, and with a binder $\Rightarrow$, relating the left-hand side and the right-hand side of the rule, and optionally with a condition or constraint that restricts the set of values allowed for the variables. Performing the transformation of a syntactic structure t is applying the rule labelled ℓ on t, which is basically done in three steps: (1) match to select a redex of t at position ω denoted $t_{|\omega}$ (possibly modulo some axioms, constraints,...); (2) instantiate the rule variables by the result(s) of the matching substitution σ; (3) replace the redex by the instantiated right-hand side. Formally: t rewrites to t' using the rule $\ell : l \Rightarrow r$ if $t_{|\omega} = \sigma(l)$ and $t' = t[\sigma(r)]_{\omega}$. This is denoted, as in Section 3.1, $t \longrightarrow_{\omega,\ell,\sigma} t'$ or simply $t \to t'$.

In this process, there are many possible choices: the rule itself, the position(s) in the structure, the matching substitution(s). For instance, one may choose to apply a rule concurrently at all disjoint positions where it matches, or using matching modulo an equational theory like associativity-commutativity, or also according to some probability. These choices are most often implicit in systems or languages based on rewriting, such as OBJ, ML, Haskell, Maude,... where they are built in the interpreters or compilers. They can be instead considered as part of the control of the rewriting process and explicitely addressed in a strategy definition or language.

5.2 Rewriting logic

The Rewriting Logic is due to Meseguer [114, 110]: *Rewriting logic (RL) is a natural model of computation and an expressive semantic framework for concurrency, parallelism, communication, and interaction. It can be used for specifying a wide range of systems and languages in various application fields. It also has good properties as a metalogical framework for representing logics. In recent years, several languages based on RL (ASF+SDF, CafeOBJ, ELAN, Maude) have been designed and implemented.* [1]

In Rewriting Logic, the syntax is based on a set of terms $\mathcal{T}(\mathcal{F})$ built with an alphabet $\mathcal{F}$ of function symbols with arities, a theory is given by a set $\mathcal{R}$ of labeled rewrite rules denoted $\ell(x_1,\dots,x_n) : l \Rightarrow r$, where the label $\ell(x_1,\dots,x_n)$ records the set of variables occurring in the rewrite rule. Formulas are sequents of the form $\pi : \langle t\rangle_E \to \langle t'\rangle_E$, where π is a *proof term* recording the proof of the sequent: $\mathcal{R} \vdash \pi : \langle t\rangle_E \to \langle t'\rangle_E$ if $\pi : \langle t\rangle_E \to \langle t'\rangle_E$ can be obtained by finite application of equational deduction rules [114] given in Fig. 2. In this context, a proof term π encodes a sequence of rewriting steps called a derivation.

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

$$\mathbf{t} : \langle t\rangle_E \to \langle t\rangle_E$$

Transitivity

$$\frac{\pi_\mathbf{1} : \langle t_1\rangle_E \to \langle t_2\rangle_E \qquad \pi_\mathbf{2} : \langle t_2\rangle_E \to \langle t_3\rangle_E}{\pi_\mathbf{1};\pi_\mathbf{2} \ : \ \langle t_1\rangle_E \to \langle t_3\rangle_E}$$

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

$$\frac{\pi_\mathbf{1} : \langle t_1\rangle_E \to \langle t'_1\rangle_E \quad \dots \quad \pi_\mathbf{n} : \langle t_n\rangle_E \to \langle t'_n\rangle_E}{\mathbf{f}(\pi_\mathbf{1},\dots,\pi_\mathbf{n}) : \langle f(t_1,\dots,t_n)\rangle_E \to \langle f(t'_1,\dots,t'_n)\rangle_E}$$

Replacement For any $\ell(x_1,\dots,x_n) : l \Rightarrow r \in \mathcal{R}$,

$$\frac{\pi_\mathbf{1} : \langle t_1\rangle_E \to \langle t'_1\rangle_E \quad \dots \quad \pi_\mathbf{n} : \langle t_n\rangle_E \to \langle t'_n\rangle_E}{\ell(\pi_\mathbf{1},\dots,\pi_\mathbf{n}) : \langle l(t_1,\dots,t_n)\rangle_E \to \langle r(t'_1,\dots,t'_n)\rangle_E}$$

Figure 2: Deduction rules for Rewriting Logic

The ELAN language, designed in the early 90's, introduced the concept of strategy by giving explicit constructs for expressing control on the rule applica-

[1] http://wrla2012.lcc.uma.es/

tion [24]. Beyond labeled rules and concatenation, other constructs for deterministic or non-deterministic choice, failure, iteration, were also defined in ELAN. A strategy is there defined as a set of proof terms in rewriting logic and can be seen as a higher-order function : if the strategy ζ is a set of proof terms π, applying ζ to the term t means finding all terms t' such that $\pi : \langle t \rangle_E \to \langle t' \rangle_E$ with $\pi \in \zeta$. Since rewriting logic is reflective, strategy semantics can be defined inside the rewriting logic by rewrite rules at the meta-level. This is the approach followed by Maude in [32, 112].

5.3 Rewriting Calculus

The rewriting calculus, also called ρ-calculus, has been introduced in 1998 by Cirstea and Kirchner [30]: *The rho-calculus has been introduced as a general means to uniformly integrate rewriting and λ-calculus. This calculus makes explicit and first-class all of its components: matching (possibly modulo given theories), abstraction, application and substitutions.*

The rho-calculus is designed and used for logical and semantical purposes. It could be used with powerful type systems and for expressing the semantics of rule based as well as object oriented paradigms. It allows one to naturally express exceptions and imperative features as well as expressing elaborated rewriting strategies. [2]

Some features of the rewriting calculus are worth emphasizing here: first-order terms and λ-terms are ρ-terms ($\lambda x.t$ is $(x \Rightarrow t)$); a rule is a ρ-term as well as a strategy, so rules and strategies are abstractions of the same nature and "first-class concepts"; application generalizes $\beta-$reduction; composition of strategies is like function composition; recursion is expressed as in λ calculus with a recursion operator μ.

5.4 Abstract reduction systems

Another view of rewriting is to consider it as an abstract relation on structural objects. An *Abstract Reduction System (ARS)* [135, 90, 28] is a labelled oriented graph $(\mathcal{O}, \mathcal{S})$ with a set of labels $\mathcal{L}$. The nodes in $\mathcal{O}$ are called *objects*. The oriented labelled edges in $\mathcal{S}$ are called *steps*: $a \xrightarrow{\phi} b$ or (a, ϕ, b), with *source* a, *target* b and *label* ϕ. Derivations are composition of steps.

The concept of Abstract Reduction System (ARS for short) has been introduced to take into account abstract properties of relations induced by rewrite systems. But thanks to the flexibility due to the definition of objects and steps, ARS are quite convenient to give a rewrite semantics to state transformation systems.

Another interest is to give a semantics to strategies in this context. Reduction strategies in term rewriting study which expressions should be selected for evaluation and which rules should be applied. These choices usually increase efficiency

[2]http://rho.loria.fr/index.html

of evaluation but may affect fundamental properties of computations such as confluence or (non-)termination.

Regarding rewriting as a relation and considering abstract rewrite systems leads to consider derivation tree exploration: derivations are computations and strategies describe selected computations.

5.5 Strategies

In the classical setting of first-order term rewriting, strategies have been used to determine at each step of a derivation which is the next redex. Thus they have often been defined as functions on the set of terms like in [132]. Let us illustrate this point of view by a few examples of strategies that have been primarily provided to describe the operational semantics of functional programming languages and the related notions of call by value, call by name, call by need. Leftmost-innermost (resp. outermost) reduction chooses the rewriting position according to suffix (resp. prefix) ordering on the set of positions in the term. Lazy reduction, as performed in Haskell for instance, or described in [52], combines innermost and outermost strategies, in order to avoid redundant rewrite steps. For that, operators of the signature have labels specifying which arguments are lazy or eager. Positions in terms are then annotated as lazy or eager, and the strategy consists in reducing the eager subterms only when their reduction allows a reduction step higher in the term [120]. Lazy reduction can also be performed with other mechanisms, such as local strategies or context-sensitive rewriting. Local strategies on operators, used for instance in the OBJ-like languages [61, 1], introduce a function to specify which and in which order the arguments of each function symbol have to be reduced. In context-sensitive rewriting [108, 109], rewriting is allowed only at some specified positions in the terms. Already in 1979, later published in [74, 75], Huet and Lévy defined the notions of needed and strongly needed redexes for orthogonal rewrite systems. The main idea here is to find the optimal way, when it exists, to reach the normal form a term. A redex is needed when there is no way to avoid reducing it to reach the normal form. Reducing only needed redexes is clearly the optimal reduction strategy, as soon as needed redexes can be decided, which is not the case in general.

Coming back to abstract reduction systems, abstract strategies are defined in [90] and in [28] as follows: for a given ARS $\mathcal{A}$, an *abstract strategy* ζ is a subset of the set of all derivations (finite or not) of $\mathcal{A}$. Playing with these definitions, [28] explored adequate definitions of termination, normal form and confluence under strategy.

This very general definition of abstract strategies is called *extensional* in [28] in the sense that a strategy is defined explicitly as a set of derivations of an abstract reduction system. The concept is useful to understand and unify reduction systems and deduction systems as explored in [90].

But abstract strategies do not capture another point of view, also frequently adopted in rewriting: a strategy is a partial function that associates to a reduction-in-progress, the possible next steps in the reduction sequence. In this case, the strategy as a function depends only on the object and the derivation so far. This notion of strategy coincides with the definition of strategy in sequential path-building games, with applications to planning, verification and synthesis of concurrent systems. This remark leads to the following *intensional* definition given in [28]. The essence of the idea is that strategies are considered as a way of constraining and guiding the steps of a reduction. So at any step in a derivation, it should be possible to say whether a contemplated next step obeys the strategy ζ. In order to take into account the past derivation steps to decide the next possible ones, the history of a derivation can be memorized to be available at each step.

5.6 Strategy languages

In the 1990s, generalizing OBJ's concept of local strategies, the idea has emerged to better formalize the control on rewriting which was performed implicitly in interpreters or compilers of rule-based programming languages. Then instead of describing a strategy by a property of its derivations, the idea is to provide a *strategy language* to specify which derivations we are interested in. Various approaches have followed, yielding different strategy languages such as Elan [97, 25], APS [107], Stratego [136], Tom [18, 17] or more recently Maude [111] and Porgy [3]. All these languages share the concern to provide abstract ways to express control of rule applications. In these flexible and expressive strategy languages, high-level strategies are defined by combining low-level primitives. The semantics of the Tom strategy language as well as others are naturally described in the rewriting calculus [30, 31].

6 Conclusion

We have retraced in this overview the main evolutions of rewriting techniques in relation with equational deduction mainly during a 40 years period going roughly from 1970 to 2010. Useful complements can be found in RTA proceedings and on the rewriting web page (rewriting.loria.fr/systems.html). Strongly involved in automated deduction before 2000, rewriting is now understood as an abstract process of computation, well-adapted to model dynamic processes. Departing from equational logic, this evolution is opening new fields of application.

References

[1] M. Alpuente, S. Escobar, and S. Lucas. Correct and complete (positive) strategy annotations for OBJ. In *Proceedings of the 5th International Workshop on Rewriting Logic and its Applications (RTA)*, volume 71 of *Elecronic Notes In Theoretical Computer Science*, pages 70–89, 2004. 20

[2] S. Anantharaman and J. Hsiang. An automated proof of the Moufang identities in alternative rings. *Journal of Automated Reasoning*, 6:79–109, 1990. 13

[3] O. Andrei, M. Fernandez, H. Kirchner, G. Melançon, O. Namet, and B. Pinaud. PORGY: Strategy-Driven Interactive Transformation of Graphs. In Rachid Echahed, editor, *TERMGRAPH, 6^{th} Int. Workshop on Computing with Terms and Graphs*, volume 48, pages 54–68, 2011. 21

[4] T. Aoto. Dealing with Non-orientable Equations in Rewriting Induction. In F. Pfenning, editor, *Proceedings of the 17th International Conference on Rewriting Techniques and Applications*, volume 4098, pages 242–256, Nara (Japan), apr 2006. Lecture Notes in Computer Science. 16

[5] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000. 9

[6] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P.D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002. 7

[7] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. System description: Inka 5.0 – a logic voyager. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 207–211, Trento, Italy, july 1999. Springer. 14

[8] 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), Amsterdam, 1990. 9

[9] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998. 9

[10] L. Bachmair. Proof by consistency in equational theories. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*, pages 228–233, 1988. 15

[11] L. Bachmair. *Canonical equational proofs*. Computer Science Logic, Progress in Theoretical Computer Science. Birkhäuser Verlag AG, 1991. 10, 15

[12] L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. *Theoretical Computer Science*, 67(2-3):173–202, October 1989. 12

[13] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proceedings 1st IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pages 346–357. IEEE, 1986. 10, 11

[14] L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 1–30. Academic Press inc., 1989. 12, 14

[15] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994. 12

[16] L. Bachmair and H. Ganzinger. Resolution theorem proving. *Handbook of automated reasoning*, 1:19–99, 2001. 12, 14

[17] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. *Tom Manual*. LORIA, Nancy (France), version 2.4 edition, October 2006. 21

[18] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007. 21

[19] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: Confluency and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986. 12

[20] N. Berregeb, A. Bouhoula, and M. Rusinowitch. Automated verification by induction with associative-commutative operators. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 220–231. Springer, 1996. 16

[21] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004. 14

[22] M. Bezem, J.W. Klop, and R. de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003. 9

[23] G. Birkhoff. On the structure of abstract algebras. *Proceedings Cambridge Phil. Soc.*, 31:433–454, 1935. 6

[24] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 2(285):155–185, July 2002. 19

[25] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, February 2001. 21

[26] A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE: An automatic theorem prover. In *Proceedings of the 1st International Conference on Logic*

Programming and Automated Reasoning, St. Petersburg (Russia), volume 624 of *Lecture Notes in Artificial Intelligence*, pages 460–462. Springer-Verlag, July 1992. 16

[27] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995. 16

[28] T. Bourdier, H. Cirstea, D.J. Dougherty, and H. Kirchner. Extensional and intensional strategies. In *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming*, volume 15 of *Electronic Proceedings In Theoretical Computer Science*, pages 1–19, 2009. 19, 20, 21

[29] A. Bundy. The automation of proof by mathematical induction. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*. Elsevier Science Publishers B. V. (North-Holland), 1999. 14

[30] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:427–498, May 2001. 19, 21

[31] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003. 21

[32] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, Meseguer J., and C. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Reriting Logic*, volume 4350. Springer, 2007. 19

[33] H. Comon. Completion of rewrite systems with membership constraints. In W. Kuich, editor, *Proceedings of ICALP 92*, volume 623 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992. 12

[34] H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 14, pages 914–959. Elsevier Science, 2001. 16

[35] H. Comon and F. Jacquemard. Ground reducibility is exptime-complete. *Information and Computation*, 187(1):123 – 153, 2003. 15

[36] H. Comon and R. Nieuwenhuis. Induction = I-Axiomatization + First-Order Consistency. *Inf. Comput.*, 159(1-2):151–186, 2000. 16

[37] E. Contejean, C. Marché, A.P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005. 9

[38] M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In N. Dershowitz, editor, *Rewriting Techniques and Applications (RTA'03)*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120. Springer-Verlag, April 1989. 9

[39] D. Delahaye. A tactic language for the system Coq. In Michel Parigot and Andrei Voronkov, editors, *Proceedings of Logic Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. 13

[40] E. Deplagne, C. Kirchner, H. Kirchner, and Q.-H. Nguyen. Proof search and proof check for equational and inductive theorems. In Franz Baader, editor, *Proceedings of CADE-19*, Miami, Florida, July 2003. Springer-Verlag. 16

[41] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982. 9

[42] N. Dershowitz. Computing with rewrite systems. *Information and Control*, 65(2/3):122–157, 1985. 15

[43] N. Dershowitz. Termination. In *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 180–224, Dijon (France), May 1985. Springer. 9

[44] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1 & 2):69–116, 1987. 9

[45] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. 9

[46] N. Dershowitz and C. Kirchner. Abstract canonical presentations. *Theoretical Computer Science*, 357(1-3):53–69, July 2006. 11

[47] N. Dershowitz, L. Marcus, and A. Tarlecki. Existence, uniqueness and construction of rewrite systems. *SIAM Journal of Computing*, 17(4):629–639, August 1988. 13

[48] N. Dershowitz, M. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. 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 31–44. Springer-Verlag, July 1987. 12

[49] N. Dershowitz and D. A. Plaisted. Equational programming. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11: The logic and acquisition of knowledge*, chapter 2, pages 21–56. Oxford Press, Oxford, 1988. 12

[50] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003. 12, 16

[51] T. Evans. On multiplicative systems defined by generators and relations. In *Proceedings of the Cambridge Philosophical Society*, pages 637–649, 1951. 8

[52] W. Fokkink, J. Kamperman, and P. Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, 2000. 20

[53] L. Fribourg. SLOG: A logic programming language intepreter based on clausal superposition and rewriting. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 172–184, Boston, MA, July 1985. 12

[54] L. Fribourg. A superposition oriented theorem prover. *Theoretical Computer Science*, 35:129–164, 1985. 13

[55] L. Fribourg. A strong restriction of the inductive completion procedure. In *Proceedings 13th International Colloquium on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 105–115. Springer-Verlag, 1986. 15

[56] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ-2. In B. Reid, editor, *Proceedings 12th ACM Symp. on Principles of Programming Languages*, pages 52–66. ACM, 1985. 7

[57] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings IJCAR '06*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 281–286. Springer, 2006. 9

[58] I. Gnaedig and H. Kirchner. Termination of rewriting under strategies. *ACM Trans. of Comput. Logic*, 10(2):1–52, 2009. 10

[59] 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, *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, volume 87 of *Lecture Notes in Computer Science*, pages 356–373. Springer-Verlag, 1980. 15

[60] J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics for order-sorted algebra. In W. Brauer, editor, *Proceeding of the 12th International Colloquium on Automata, Languages and Programming, Nafplion (Greece)*, volume 194 of *Lecture Notes in Computer Science*, pages 221–231. Springer-Verlag, 1985. 7

[61] J. A. Goguen, C. Kirchner, H. 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. 7, 20

[62] J. A. Goguen and J. Meseguer. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problem. In *Proceedings 2nd IEEE Symposium on Logic in Computer Science, Ithaca (N.Y., USA)*, pages 18–29. IEEE Computer Society Press, 1987. 7

[63] J. A. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing software specifications. In M. K. Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE Press, Cambridge (MA, USA), 1979. Reprinted in *Software Specification Techniques*, N. Gehani and A. McGettrick, editors, Addison-Wesley, 1985, pages 391-420. 7

[64] J.A. Goguen. Order-Sorted Algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992. 7

[65] J.A. Goguen and G. Malcolm, editors. *Software Engineering with OBJ: Algebraic Specification in Action*, volume 2 of *Advances in Formal Methods*. Kluwer Academic Publishers, Boston, 2000. ISBN 0-7923-7757-5. 7

[66] J.A. Goguen and J. Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In Douglas De Groot and Gary Lindstrom, editors, *Functional and Logic Programming*, pages 295–363. Prentice Hall, Inc., 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984. 12

[67] B. Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 5(3-4):131–158, May 1994. 10

[68] N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In *Proc. 14th Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 311–320. Springer-Verlag, 2003. 9

[69] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25(3):255 – 300, 1985. 13

[70] 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, October 1992. 13

[71] J. Hsiang and M. Rusinowitch. On word problem in equational theories. In Th. Ottmann, editor, *Proceedings of 14th International Colloquium on Automata, Languages and Programming, Karlsruhe (Germany)*, volume 267 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 1987. 14

[72] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980. Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977. 10, 11

[73] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, October 1982. Preliminary version in Proceedings 21st Symposium on Foundations of Computer Science, IEEE, 1980. 15

[74] 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. MIT press, 1991. 20

[75] 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. MIT press, 1991. 20

[76] 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 inc., 1980. 9

[77] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, July 1980. 12

[78] T. Isakowitz and J. Gallier. Rewriting in order-sorted equational logic. In R. Kowalski and K. Bowen, editors, *Proceedings of Logic Programming Conference*, Seattle (USA), 1988. The MIT press. 7

[79] J.-P. Jouannaud. Confluent and coherent equational term rewriting systems. Applications to proofs in abstract data types. In G. Ausiello and M. Protasi, editors, *Proceedings of the 8th Colloquium on Trees in Algebra and Programming, L'Aquila (Italy)*, volume 159 of *Lecture Notes in Computer Science*, pages 269–283. Springer-Verlag, 1983. 11

[80] 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, February 1992. 7

[81] 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. 12

[82] J.-P. Jouannaud and E. Kounalis. Proof by induction in equational theories without constructors. In *Proceedings 1st IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pages 358–366, 1986. 15

[83] J.-P. Jouannaud and P. Lescanne. On multiset orderings. *Information Processing Letters*, 10:57–63, 1982. 9

[84] J.-P. Jouannaud and B. Waldmann. Reductive conditional term rewriting systems. In M. Wirsing, editor, *Proceedings of the Third IFIP Working Conference on Formal Description of Programming Concepts*, Ebberup, (Denmark), 1986. Elsevier Science Publishers B. V. (North-Holland). 12

[85] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33:175–193, 1984. 12

[86] S. Kaplan. Simplifying conditional term rewriting systems: Unification, termination and confluence. *Journal of Symbolic Computation*, 4(3):295–334, December 1987. 12

[87] D. Kapur, P. Narendran, and H. Zhang. On sufficient completeness and related properties of term rewriting systems. *Acta Informatica*, 24:395–415, 1987. 15

[88] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995. 16

[89] M. Kaufmann and J.S. Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology. 14

[90] C. Kirchner, F. Kirchner, and H. Kirchner. Strategic computations and deductions. In *Festchrift in honor of Peter Andrews*, Studies in Logic and the Foundations of Mathematics. Elsevier, 2008. 19, 20

[91] C. Kirchner and H. Kirchner. Reveur-3: Implementation of a general completion procedure parametrized by built-in theories and strategies. *Science of Computer Programming*, 20(8):69–86, 1986. 12

[92] C. Kirchner and H. Kirchner. Constrained equational reasoning. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, Portland (Oregon)*, pages 382–389. ACM Press, July 1989. 12

[93] C. Kirchner, H. Kirchner, and A. Mégrelis. OBJ for OBJ. In J.A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, chapter 6, pages 307–330. Kluwer, Boston, 2000. 7

[94] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ-3. In *Proceedings of 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 1988. 7

[95] C. Kirchner, H. Kirchner, and F. Nahon. Narrowing based inductive proof search. In A. Voronkov and C. Weidenbach, editors, *Programming Logics*, volume 7797 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2013. 16

[96] 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. 12

[97] 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, pages 131–158. The MIT press, 1995. 21

[98] 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-Verlag, 1995. 12

[99] 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. 9

[100] 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. 8, 10, 15

[101] E. Kounalis. Completeness in data type specifications. In B. Buchberger, editor, *Proceedings EUROCAL Conference, Linz (Austria)*, volume 204 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, 1985. 15

[102] D. S. Lankford. A simple explanation of inductionless induction. *Louisiana Tech. University, Dep. of Math.*, Memo MTP-14, Ruston, 1981. 15

[103] D. S. Lankford and A. Ballantyne. Decision procedures for simple equational theories with associative commutative axioms: complete sets of asso-

ciative commutative reductions. Technical report, Univ. of Texas at Austin, Dept. of Mathematics and Computer Science, 1977. 11

[104] 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, January 1990. 15

[105] P. Lescanne. Computer experiments with the REVE term rewriting systems generator. In *Proceedings of 10th ACM Symposium on Principles of Programming Languages*, pages 99–108. ACM, 1983. 13

[106] P. Lescanne. On the recursive decomposition ordering with lexicographical status and other related orderings. *Journal of Automated Reasoning*, 6:39–49, 1990. 9

[107] A.A. Letichevsky. Development of rewriting strategies. In Maurice Bruynooghe and Jaan Penjam, editors, *PLILP*, volume 714 of *Lecture Notes in Computer Science*, pages 378–390. Springer, 1993. 21

[108] S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In H. Sondergaard, editor, *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82–93, Firenze, Italy, September 2001. ACM Press, New York. 20

[109] S. Lucas. Termination of rewriting with strategy annotations. In A. Voronkov and R. Nieuwenhuis, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'01*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 669–684, La Habana, Cuba, December 2001. Springer-Verlag, Berlin. 20

[110] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000. 18

[111] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005. 21

[112] N. Martí-Oliet, J. Meseguer, and A. Verdejo. A rewriting semantics for Maude strategies. *EPTCS*, 238(3):227–247, 2008. 19

[113] W. McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19:263–276, 1997. 13

[114] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. 18

[115] J. Meseguer and J. A. Goguen. Initiality, induction and computability. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*. Cambridge University Press, 1985. 7

[116] A. Middeldorp and H. Zantema. Simple termination of rewrite systems. *Theoretical Computer Science*, 175(1):127–158, 1997. 9

[117] P.D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 115–137. Springer-Verlag, 1997. 7

[118] D. R. Musser. On proving inductive properties of abstract data types. In *Proceedings 7th ACM Symp. on Principles of Programming Languages*, pages 154–162. ACM, 1980. 15

[119] M. H. A. Newman. On theories with a combinatorial definition of equivalence. In *Annals of Math*, volume 43, pages 223–243, 1942. 10

[120] Q.-H. Nguyen. Compact normalisation trace via lazy rewriting. In S. Lucas and B. Gramlich, editors, *Proceedings of the 1st International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume 57 of *Elecronic Notes In Theoretical Computer Science*. Elsevier, 2001. 20

[121] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. *Handbook of automated reasoning*, 1:371–443, 2001. 14

[122] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. 13, 14

[123] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002. ISBN 978-1-4757-3661-8. 9

[124] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. 13

[125] G. Peterson and M.E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981. 11, 12

[126] D. Plaisted. Semantic confluence and completion method. *Information and Control*, 65:182–215, 1985. 15

[127] 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*, volume 1, pages 273–364. Oxford University Press, 1993. 9

[128] U. Reddy. Term rewriting induction. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449 of *Lecture Notes in Computer Science*, pages 162–177. Springer-Verlag, 1990. 16

[129] J. Steinbach. Generating polynomial orderings. *Information Processing Letters*, 49:85–93, 1994. 9

[130] J. Steinbach. Simplification orderings: History of results. *Fundam. Inform.*, 24(1/2):47–87, 1995. 9

[131] M. E. Stickel. A case study of theorem proving by the Knuth-Bendix method: Discovering that $x^3 = x$ implies ring commutativity. In R. Shostak, editor, *Proceedings 7th International Conference on Automated Deduction, Napa Valley (Calif., USA)*, volume 170 of *Lecture Notes in Computer Science*, pages 248–258. Springer-Verlag, 1984. 13

[132] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds. 9, 20

[133] J.-J. Thiel. Stop losing sleep over incomplete data type specifications. In *Proceeding 11th ACM Symp. on Principles of Programming Languages*, pages 76–82. ACM, 1984. 15

[134] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, May 1986. 10

[135] V. van Oostrom and R. de Vrijer. *Strategies*, volume 2, chapter 9. Cambridge University Press, 2003. 19

[136] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001. 21

[137] H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50, 1994. 9

[138] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24(1):89–105, 1995. 9

[139] H. Zantema. Termination of string rewriting proved automatically. *Journal of Automated Reasoning*, 34(2):105–139, 2005. 9

[140] H. Zhang and D. Kapur. First-order theorem proving using conditional rewrite rules. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 1988. 12, 13

[141] H. T. Zhang and J.-L. Rémy. Contextual rewriting. In J.-P. Jouannaud, editor, *Proceedings 1st Conference on Rewriting Techniques and Applications, Dijon (France)*, volume 202 of *Lecture Notes in Computer Science*, pages 46–62, Dijon (France), 1985. Springer-Verlag. 12