Electronic Notes in Theoretical Computer Science 4 (1996)

Rewriting Logic as a Logical and Semantic Framework ⋆

Narciso Martí-Oliet [a,b,1] and José Meseguer [b,2]

[a] *Departamento de Informática y Automática*
Escuela Superior de Informática
Universidad Complutense de Madrid, Spain
narciso@eucmos.sim.ucm.es

[b] *Computer Science Laboratory*
SRI International
Menlo Park CA 94025, USA
meseguer@csl.sri.com

Abstract

Rewriting logic [40] is proposed as a logical framework in which other logics can be represented, and as a semantic framework for the specification of languages and systems. Using concepts from the theory of general logics [39], representations of an object logic $\mathcal{L}$ in a framework logic $\mathcal{F}$ are understood as mappings $\mathcal{L} \to \mathcal{F}$ that translate one logic into the other in a conservative way. The ease with which such maps can be defined is discussed in detail for the cases of linear logic, logics with quantifiers, and any sequent calculus presentation of a logic for a very general notion of "sequent." Using the fact that rewriting logic is reflective, it is often possible to reify inside rewriting logic itself a representation map $\mathcal{L} \to \mathit{RWLogic}$ for the finitely presentable theories of $\mathcal{L}$. Such a reification takes the form of a map between the abstract data types representing the finitary theories of $\mathcal{L}$ and of *RWLogic*.

Regarding the different but related use of rewriting logic as a semantic framework, the straightforward way in which very diverse models of concurrency can be expressed and unified within rewriting logic is illustrated with CCS. In addition, the way in which constraint solving fits within the rewriting logic framework is briefly explained.

⋆ This paper is a short version of [36], where the reader can find more examples and details not discussed here.

[1] Supported by Office of Naval Research Contract N00014-95-C-0225 and by CICYT, TIC 95-0433-C03-01.

[2] Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9224005, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

1 Introduction

The relationships between logic and computation, and the mutual interactions between both fields, are becoming stronger and more pervasive than they have ever been. In fact, our way of thinking about both logic and computation is being altered quite strongly. For example, there is such an increasingly strong connection—in some cases to the point of complete identification—between computation and deduction, and such impressive progress in compilation techniques and computing power, that the frontiers between logical systems, theorem-provers, and declarative programming languages are shifting and becoming more and more tenuous, with each area influencing and being influenced by the others.

Similarly, in the specification of languages and systems there is an increasing shift from mathematically precise but somewhat restricted formalisms towards specifications that are not only mathematical, but actually logical in nature, as exemplified, for example, by specification formalisms such as algebraic specifications and structural operational semantics. In this way, languages and systems that in principle may not seem to bear any resemblance to logical systems and may be completely "conventional" in nature, end up being conceptualized primarily as *formal* systems.

However, any important development brings with it new challenges and questions. Two such questions, that we wish to address in this paper are:

- *How can the proliferation of logics be handled?*
- *Can flexible logics allowing the specification and prototyping of a wide variety of languages and systems with naturalness and ease be found?*

Much fruitful research has already been done with the aim of providing adequate answers to these questions. Our aim here is to contribute in some measure to their ongoing discussion by suggesting that rewriting logic [40] seems to have particularly good properties recommending its use as both a *logical framework* in which many other logics can be represented, and as a general *semantic framework* in which many languages and systems can be naturally specified and prototyped.

In our view, the main need in handling the proliferation of logics is primarily conceptual. What is most needed is a *metatheory* of logics helping us to better understand and explore the boundaries of the "space" of all logics, present and future, and to relate in precise and general ways many of the logics that we know or wish to develop.

Following ideas that go back to the original work of Goguen and Burstall on *institutions* [19], we find very useful understanding the space of all logics as a *category*, with appropriate translations between logics as the arrows or morphisms between them. The theory of general logics [39] that we present in summary form in Section 2 expands the primarily model-theoretic viewpoint provided by institutions to give an adequate treatment of proof-theoretic aspects such as entailment and proof structures, and suggests not just one space or category of logics, but several, depending on the proof-theoretic or model-

theoretic aspects that we wish to focus on.

In our view, the quest for a *logical framework*, understood as a logic in which many other logics can be represented, is important but is not the primary issue. Viewed from the perspective of a general space of logics, such a quest can in principle—although perhaps not in all approaches—be understood as the search within such a space for a logic $\mathcal{F}$ such that many other logics $\mathcal{L}$ can be represented in $\mathcal{F}$ by means of mappings $\mathcal{L} \longrightarrow \mathcal{F}$ that have particularly nice properties such as being conservative translations.

Considered in this way, and assuming a very general axiomatic notion of logic and ambitious enough requirements for a framework, there is in principle no guarantee that such an $\mathcal{F}$ will necessarily be found. However, somewhat more restricted successes such as finding an $\mathcal{F}$ in which all the logics of "practical interest," having finitary presentations of their syntax and their rules, can be represented can be very valuable and can provide a great economy of effort. This is because, if an implementation for such a framework logic exists, it becomes possible to implement through it all the other "object logics" that can be adequately represented in the framework logic.

Much work has already been done in this area, including the Edinburgh logical framework LF [25,26,17] and meta-theorem-provers such as Isabelle [51], λProlog [49,16], and Elf [52], all of which adopt as framework logics different variants of higher-order logics or type theories. There has also been important work on what Basin and Constable [3] call *metalogical* frameworks. These are frameworks supporting reasoning about the metalogical aspects of the logics being represented. Typically, this is accomplished by reifying as "data" the proof theory of the logic being represented in a process that is described in [3] as *externalizing* the logic in question. This is in contrast to the more *internalized* form in which logics are represented in LF and in meta-theorem-provers, so that deduction in the object logic is mirrored by deduction—for example, type inference—in the framework logic. Work on metalogical frameworks includes the already mentioned paper by Basin and Constable [3], who advocate constructive type theory as the framework logic, work of Matthews, Smaill and Basin [38], who use Feferman's FS_0 [15], a logic designed with the explicit purpose of being a metalogical framework, earlier work by Smullyan [56], and work by Goguen, Stevens, Hobley and Hilberdink [23] on the 2OBJ meta-theorem-prover, which uses order-sorted equational logic [22,24].

A difficulty with systems based on higher-order type theory such as LF is that it may be quite awkward and of little practical use to represent logics whose structural properties differ considerably from those of the type theory. For example, linear and relevance logics do not have adequate representations in LF, in a precise technical sense of "adequate" [17, Corollary 5.1.8]. Since in metalogical frameworks a direct connection between deduction in the object and framework logics does not have to be maintained, they seem in principle much more flexible in their representational capabilities. However, this comes at a price, since the possibility of directly using an implementation of the framework logic to implement an object logic is compromised.

In relation to this previous work, rewriting logic seems to have great flexibility to represent in a natural way many other logics, widely different in nature, including equational, Horn, and linear logics, and any sequent calculus presentation of a logic under extremely general assumptions about such a logic. Moreover, quantifiers can also be treated without problems. More experience in representing other logics is certainly needed, but we are encouraged by the naturalness and directness—often preserving the original syntax and rules—with which the logics that we have studied can be represented. This is due to the great simplicity and generality of rewriting logic, since in it all syntax and structural axioms are user-definable, so that the abstract syntax of an object logic can be represented as an algebraic data type, and is also due to the existence of only a few general "meta" rules of deduction relative to the rewrite rules given by a specification, where such a specification can be used to describe with rewrite rules the rules of deduction of the object logic in question. In addition, the direct correspondence between proofs in object logics and proofs in the framework logic can often be maintained in a *conservative* way by means of maps of logics, so that an implementation of rewriting logic can directly support an implementation of an object logic. Furthermore, given the directness with which logics can be represented, the task of proving conservativity is in many cases straightforward. Finally, although we do not discuss this aspect, externalization of logics to support metalogical reasoning is also possible in rewriting logic [37].

Another important difference is that most approaches to logical frameworks are proof-theoretic in nature, and thus they do not address the model theories of the logics being represented. By contrast, several of the representations into rewriting logic that we have studied—such as those for equational logic, Horn logic, and linear logic—involve both models and proofs and are therefore considerably more informative than purely proof-theoretic representations.

As we have already mentioned, the distinction between a logical system and a language or a model of computation is more and more in the eyes of the beholder, although of course efficiency considerations and the practical uses intended may indeed strongly influence the design choices. Therefore, even though at the most basic mathematical level there may be little distinction between the general way in which a logic, a programming language, a system, or a model of computation are represented in rewriting logic, the criteria and case studies to be used in order to judge the merits of rewriting logic as a semantic framework are different from those relevant for its use as a logical framework.

One important consideration is that, from a computational point of view, rewriting logic deduction is intrinsically *concurrent*. In fact, it was the search for a general concurrency model that would help unify the somewhat bewildering heterogeneity of existing models that provided the original impetus for the first investigations on rewriting logic [40]. The generality and naturalness with which many concurrency models can be expressed in rewriting logic has already been illustrated at length in several papers [40,42]. In this paper, we just discuss in some detail the case of Milner's CCS [48].

Deduction with constraints can greatly increase the efficiency of theorem provers and logic programming languages. The most classical constraint solving algorithm is syntactic unification, which corresponds to solving equations in a free algebra, the so-called Herbrand model, and is used in resolution. However, much more efficient deduction techniques than those afforded by resolution can be obtained by building in additional knowledge of special theories in the form of constraint solving algorithms such as, for example, semantic unification, or equalities and inequalities in a numerical domain. In the past few years many authors have become aware that many constraint solving algorithms can be specified declaratively using rewrite rules. However, since constraint solving is usually nondeterministic, the usual equational logic interpretation of rewrite rules is clearly inadequate as a mathematical semantics. By contrast, rewriting logic completely avoids such inadequacies and can serve as a semantic framework for logical systems and languages using constraints, including parallel ones.

The paper begins with a summary of the theory of general logics [39] that provides the conceptual basis for our discussion of logical frameworks. Then the rules of deduction of rewriting logic are introduced, and the Maude language based on rewriting logic is briefly discussed. This is followed by three sections illustrating the representation of logics in the rewriting logic framework: linear logic, quantifiers and sequent systems. Using the fact that rewriting logic is reflective [9,10], it is often possible to reify inside rewriting logic itself a representation map $\mathcal{L} \rightarrow \mathit{RWLogic}$ for the finitely presentable theories of $\mathcal{L}$. Such a reification takes the form of a map between the abstract data types representing the finitary theories of $\mathcal{L}$ and of *RWLogic*, as we illustrate with the linear logic example. The use of rewriting logic as a semantic framework is illustrated by means of the CCS and constraint solving examples. The paper ends with some concluding remarks.

2 General logics

A modular and general axiomatic theory of logics should adequately cover all the key ingredients of a logic. These include: a *syntax*, a notion of *entailment* of a sentence from a set of axioms, a notion of *model*, and a notion of *satisfaction* of a sentence by a model. The theory of *general logics* [39] provides axiomatic notions formalizing the different aspects of a logic and of their combinations into fuller notions of logic:

- An *entailment system* axiomatizes the consequence relation of a logic.
- The notion of *institution* [19,20] covers the model-theoretic aspects of a logic, focusing on the notion of satisfaction.
- A *logic* is obtained by combining an entailment system and an institution.
- A *proof calculus* enriches an entailment system with an actual proof theory.
- A *logical system* is a logic with a choice of a proof calculus for it.

Here we give a brief review of the required notions; a detailed account with many examples can be found in [39] (see also [36,37]).

Syntax can typically be given by a *signature* Σ providing a grammar on which to build *sentences*. We assume that for each logic there is a category **Sign** of possible signatures for it, and a functor *sen* assigning to each signature Σ the set $sen(\Sigma)$ of all its sentences. For a given signature Σ in **Sign**, *entailment* (also called *provability*) of a sentence $\varphi \in sen(\Sigma)$ from a set of axioms $\Gamma \subseteq sen(\Sigma)$ is a relation $\Gamma \vdash \varphi$ which holds if and only if we can prove φ from the axioms Γ using the rules of the logic. This relation must be reflexive, monotonic, transitive, and must preserve translations between signatures. These components constitute an *entailment system*.

A *theory* in a given entailment system is a pair (Σ, Γ) with Σ a signature and $\Gamma \subseteq sen(\Sigma)$.

An institution consists of a category **Sign** of signatures and a functor $sen : \mathbf{Sign} \to \mathbf{Set}$ associating to each signature a set of sentences, together with a functor **Mod** that associates to each signature Σ a category of Σ-*models*, and a binary relation $\models$ between models and sentences called *satisfaction* satisfying appropriate requirements.

Combining an entailment system and an institution we obtain a *logic*, defined as a 5-tuple $\mathcal{L} = (\mathbf{Sign}, sen, \mathbf{Mod}, \vdash, \models)$ such that:

- $(\mathbf{Sign}, sen, \vdash)$ is an entailment system,
- $(\mathbf{Sign}, sen, \mathbf{Mod}, \models)$ is an institution,

and the following *soundness condition* is satisfied: for any $\Sigma \in |\mathbf{Sign}|$, $\Gamma \subseteq sen(\Sigma)$, and $\varphi \in sen(\Sigma)$,

$$\Gamma \vdash \varphi \implies \Gamma \models \varphi,$$

where, by definition, the relation $\Gamma \models \varphi$ holds if and only if $M \models \varphi$ holds for any model M that satisfies all the sentences in Γ.

The detailed treatment in [39] includes also a flexible axiomatic notion of a *proof calculus*—in which proofs of entailments, not just the entailments themselves, are first class citizens—and the notion of a *logical system* that consists of a logic together with a choice of a proof calculus for it.

One of the most interesting fruits of the theory of general logics is that it gives us a method for *relating* logics in a general and systematic way, and to exploit such relations in many applications. The key notion is that of a *mapping* translating one logic into another

$$\mathcal{L} \longrightarrow \mathcal{L}'$$

that preserves whatever logical properties are relevant, such as provability of formulas, or satisfaction of a formula by a model. Therefore, we have maps of entailment systems, institutions, logics, proof calculi, and logical systems. Such mappings allow us to relate in a rigorous way different logics, to combine different formalisms together, and to explore new logics for computational purposes. A detailed treatment of such maps is given in [39]; here we just give a brief sketch.

Basically, a map of entailment systems $(\Phi, \alpha) : \mathcal{E} \longrightarrow \mathcal{E}'$ maps the language of $\mathcal{E}$ to that of $\mathcal{E}'$ in a way that respects the entailment relation. This means that signatures of $\mathcal{E}$ are functorially mapped by Φ to signatures of $\mathcal{E}'$, and that sentences of $\mathcal{E}$ are mapped by α to sentences of $\mathcal{E}'$ in a way that is coherent with the mapping of their corresponding signatures. In addition, α must respect the entailment relations $\vdash$ of $\mathcal{E}$ and $\vdash'$ of $\mathcal{E}'$, i.e., we must have

$$\Gamma \vdash \varphi \;\Rightarrow\; \alpha(\Gamma) \vdash' \alpha(\varphi).$$

The map is *conservative* when this implication is an equivalence. For many interesting applications one needs to map signatures of $\mathcal{E}$ to *theories* of $\mathcal{E}'$, that is, Σ is mapped by Φ to (Σ', Γ'), with $\Gamma' \subseteq sen'(\Sigma')$. It is this more general notion of *map between entailment systems* that is axiomatized by the definition in [39].

A *map of institutions* $(\Phi, \alpha, \beta) : \mathcal{I} \longrightarrow \mathcal{I}'$ is similar in its syntax part to a map of entailment systems. In addition, for models we have a natural functor $\beta : \mathbf{Mod}'(\Phi(\Sigma)) \longrightarrow \mathbf{Mod}(\Sigma)$ "backwards" from the models in $\mathcal{I}'$ of a translated signature $\Phi(\Sigma)$ to the models in $\mathcal{I}$ of the original signature Σ, and such a mapping respects the satisfaction relations $\models$ of $\mathcal{I}$ and $\models'$ of $\mathcal{I}'$, in the sense that $M' \models' \alpha(\varphi) \iff \beta(M') \models \varphi$. Maps of institutions are different from the *institution morphisms* in [20].

A *map of logics* has now a very simple definition. It consists of a pair of maps: one for the underlying entailment systems, and another for the underlying institutions, such that both maps agree on how they translate signatures and sentences. There are also notions of *map of proof calculi* and *map of logical systems*, for which we refer the reader to [39].

As we have already explained in the introduction, viewed from the perspective of a general space of logics that can be related to each other by means of mappings, the quest for a *logical framework* can be understood as the search within such a space for a logic $\mathcal{F}$ (the *framework* logic) such that many other logics (the *object* logics) such as, say, $\mathcal{L}$ can be represented in $\mathcal{F}$ by means of mappings $\mathcal{L} \longrightarrow \mathcal{F}$ that have good enough properties. The minimum requirement that seems reasonable to make on a representation map $\mathcal{L} \longrightarrow \mathcal{F}$ is that it should be a *conservative* map of entailment systems. Under such circumstances, we can reduce issues of provability in $\mathcal{L}$ to issues of provability in $\mathcal{F}$, by mapping the theories and sentences of $\mathcal{L}$ into $\mathcal{F}$ using the conservative representation map. Given a computer implementation of deduction in $\mathcal{F}$, we can use the conservative map to prove theorems in $\mathcal{L}$ by proving the corresponding translations in $\mathcal{F}$. In this way, the implementation for $\mathcal{F}$ can be used as a generic theorem-prover for many logics.

However, since maps between logics can, as we have seen, respect additional logical structure such as the model theory or the proofs, in some cases a representation map into a logical framework may be particularly informative because, in addition to being a conservative map of entailment systems, it is also a map of institutions, or a map of proof calculi. For example, when rewriting logic is chosen as a logical framework, appropriate representation maps for equational logic, Horn logic, and propositional linear logic can be

shown to be maps of institutions also [36]. In general, however, since the model theories of different logics can be very different from each other, it is not reasonable to expect or require that the representation maps into a logical framework will always be maps of institutions. Nevertheless, what it can always be done is to "borrow" the additional logical structure that $\mathcal{F}$ may have (institution, proof calculus) to endow $\mathcal{L}$ with such a structure, so that the representation map does indeed preserve the extra structure [7].

Having criteria for the adequacy of maps representating logics in a logical framework is not enough. An equally important issue is having criteria for the *generality* of a logical framework, so that it is in fact justified to call it by that name. That is, given a candidate logical framework $\mathcal{F}$, how many logics can be adequately represented in $\mathcal{F}$? We can make this question precise by defining the *scope* of a logical framework $\mathcal{F}$ as the class of entailment systems $\mathcal{E}$ having conservative maps of entailment systems $\mathcal{E} \longrightarrow \mathcal{F}$. In this regard, the axioms of the theory of general logics that we have presented are probably too general; without adding further assumptions it is not reasonable to expect that we can find a logical framework $\mathcal{F}$ whose scope is the class of *all* entailment systems. A much more reasonable goal is finding an $\mathcal{F}$ whose scope includes all entailment systems of "practical interest," having finitary presentations of their syntax and their rules of deduction. Axiomatizing such finitely presentable entailment systems and proof calculi so as to capture—in the spirit of the more general axioms that we have presented, but with stronger requirements—all logics of "practical interest" (at least for computational purposes) is a very important research task.

Another important property that can help measuring the suitability of a logic $\mathcal{F}$ as a logical framework is its *representational adequacy*, understood as the naturalness and ease with which entailment systems can be represented, so that the representation $\mathcal{E} \longrightarrow \mathcal{F}$ mirrors $\mathcal{E}$ as closely as possible. That is, a framework requiring very complicated encodings for many object logics of interest is less representationally adequate than one for which most logics can be represented in a straightforward way, so that there is in fact little or no "distance" between an object logic and its corresponding representation. Although at present we lack a precise definition of this property, it is quite easy to observe its absence in particular examples. We view representational adequacy as a very important practical criterion for judging the relative merits of different logical frameworks.

In this paper, we present rewriting logic as a logic that seems to have particularly good properties as a logical framework. The evidence we can give within the space constraints of this paper is necessarily partial; further evidence can be found in [36,37]. We conjecture that the scope of rewriting logic contains all entailment systems of "practical interest" for a reasonable axiomatization of such systems.

3 Rewriting logic

A *signature* in rewriting logic is an equational theory (Σ, E), where Σ is an equational signature and E is a set of Σ-equations [3]. Rewriting will operate on equivalence classes of terms modulo E. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set of equations E is empty. Techniques for rewriting modulo equations have been studied extensively [14] and can be used to implement rewriting modulo many equational theories of interest.

Given a signature (Σ, E), *sentences* of rewriting logic are of the form

$$[t]_E \longrightarrow [t']_E,$$

where t and t' are Σ-terms possibly involving some variables, and $[t]_E$ denotes the equivalence class of the term t modulo the equations E. A theory in this logic, called a *rewrite theory*, is a slight generalization of the usual notion of theory in that, in addition, we allow the axioms $[t]_E \longrightarrow [t']_E$ (called *rewrite rules*) to be labelled, because this is very natural for many applications [4].

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sentence $[t] \longrightarrow [t']$, or that $[t] \longrightarrow [t']$ is a *(concurrent) $\mathcal{R}$-rewrite*, and write $\mathcal{R} \vdash [t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ can be obtained by finite application of the following *rules of deduction* (where we assume that all the terms are well formed and $t(\overline{w}/\overline{x})$ denotes the simultaneous substitution of w_i for x_i in t):

(i) **Reflexivity**. For each $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] \longrightarrow [t]}$.

(ii) **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}.$$

(iii) **Replacement**. For each rule $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,

$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}.$$

(iv) **Transitivity**

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}.$$

[3] Rewriting logic is parameterized with respect to the version of the underlying equational logic, which can be unsorted, many-sorted, order-sorted, or the recently developed membership equational logic [6,43].

[4] Moreover, the main results of rewriting logic have been extended to conditional rules in [40] with very general rules of the form

$$r : [t] \longrightarrow [t'] \ \ \textit{if} \ \ [u_1] \longrightarrow [v_1] \ \wedge \ldots \wedge \ [u_k] \longrightarrow [v_k].$$

This increases considerably the expressive power of rewrite theories, of which later examples in this paper make use.

Rewriting logic is a logic for reasoning correctly about *concurrent systems* having *states*, and evolving by means of *transitions*. The signature of a rewrite theory describes a particular structure for the states of a system—e.g., multiset, binary tree, etc.—so that its states can be distributed according to such a structure. The rewrite rules in the theory describe which *elementary local transitions* are possible in the distributed state by concurrent local transformations. The rules of rewriting logic allow us to reason correctly about which *general* concurrent transitions are possible in a system satisfying such a description. Thus, computationally, each rewriting step is a parallel local transition in a concurrent system.

Alternatively, however, we can adopt a logical viewpoint instead, and regard the rules of rewriting logic as *metarules* for correct deduction in a *logical system*. Logically, each rewriting step is a logical *entailment* in a formal system. This second viewpoint is particularly fruitful when using rewriting logic as a logical framework [36,37].

Thus, in rewriting logic $[t]$ should not be understood as a *term* in the usual first-order logic sense, but as a *proposition* or *formula*—built up using the *connectives* in Σ—that asserts being in a certain *state* having a certain *structure*. However, unlike most other logics, the logical connectives Σ and their structural properties E are entirely *user-definable*. This provides great flexibility for considering many different state structures and makes rewriting logic very general in its capacity to deal with many different types of concurrent systems, and also in its capacity to represent many different logics.

The computational and the logical viewpoints under which rewriting logic can be interpreted can be summarized in the following diagram of correspondences, which will be further illustrated by means of the examples described in later sections.

$$\begin{array}{lll} \textit{State} & \longleftrightarrow \textit{Term} & \longleftrightarrow \textit{Proposition} \\ \textit{Transition} & \longleftrightarrow \textit{Rewriting} & \longleftrightarrow \textit{Deduction} \\ \textit{Distributed} & \longleftrightarrow \textit{Algebraic} & \longleftrightarrow \textit{Propositional} \\ \textit{Structure} & \textit{Structure} & \textit{Structure} \end{array}$$

The model theory of rewriting logic has been developed in detail in [40], where initiality, soundness and completeness theorems are proved. Rewriting logic models, called $\mathcal{R}$-*systems*, have a natural *category* structure, with states (or formulas) as objects, transitions (or proofs) as morphisms, and sequential composition as morphism composition, and in them dynamic behavior exactly corresponds to deduction.

In what follows, we will use the syntax of Maude [41,46,8], a wide spectrum programming language directly based on rewriting logic, to present rewrite theories. In Maude, there are essentially two kinds of *modules*:[5]

- *Functional modules*, which are of the form `fmod` $\mathcal{E}$ `endfm` for an equational

[5] There are also *object-oriented modules*, which can be reduced to a special case of system modules [41] and are not used here.

theory $\mathcal{E}$, and

- *System modules*, which are of the form `mod` $\mathcal{R}$ `endm` for a rewrite theory $\mathcal{R}$.

In functional modules, equations are declared with the keywords `eq` or `ceq` (for conditional equations), and in system modules with the keywords `ax` or `cax`. Certain equations, such as associativity, commutativity, or identity, for which rewriting modulo is provided, can be declared together with the corresponding function using the keywords `assoc`, `comm`, `id`. Rules can only appear in system modules, and are declared with the keywords `rl` or `crl`.

The version of rewriting logic used for Maude in this paper is *order-sorted*[6] . This means that rewrite theories are typed (types are called *sorts*) and can have subtypes (subsorts), and that function symbols can be overloaded. In particular, functional modules are order-sorted equational theories [22] and they form a sublanguage similar to OBJ [24]. Logically, this corresponds to a map of logics

$$\mathit{OSEqtl} \longrightarrow \mathit{OSRWLogic}$$

embedding (order-sorted) equational logic within (order-sorted) rewriting logic. The details of this map of logics are discussed in [36, Section 4.1].

As in OBJ, Maude modules can be imported by other modules, and can also be parameterized by means of *theories* that specify semantic requirements for interfaces.

Since the power and the range of applications of a multiparadigm logic programming language can be substantially increased if it is possible to solve queries involving *logical variables* in the sense of relational programming, as in the Prolog language, we are naturally led to seek a unification of the three paradigms of functional, relational and concurrent object-oriented programming into a single multiparadigm logic programming language. This unification can be attained in a language extension of Maude called MaudeLog. The integration of Horn logic is achieved by a map of logics

$$\mathit{OSHorn} \longrightarrow \mathit{OSRWLogic}$$

that systematically relates order-sorted Horn logic to order-sorted rewriting logic. The details of this map are discussed in [36, Section 4.2].

4 Linear logic

In this section, we describe a map of logics $\mathit{LinLogic} \longrightarrow \mathit{OSRWLogic}$ mapping theories in full quantifier-free first-order linear logic to rewrite theories. We do not provide much motivation for linear logic, referring the reader to [18,57,35] for example. We need to point out, nonetheless, the way linear logic is seen as an entailment system. If one thinks of formulas as sentences and of the turnstile symbol "$\vdash$" in a sequent as the entailment relation, then this relation is not monotonic, because in linear logic the structural rules of weakening

[6] The latest version of Maude [8] is based on the recently developed membership equational logic, which extends order-sorted equational logic and at the same time has a simpler and more general model theory [6,43].

and contraction are forbidden, so that, for example, we have the sequent $A \vdash A$ as an axiom, but we cannot derive either $A, B \vdash A$ or even $A, A \vdash A$. The point is that, for Σ a linear logic signature, the elements of $sen(\Sigma)$ should not be identified with *formulas* but with *sequents*. Viewed as a way of generating sequents, i.e., identifying our entailment relation $\vdash$ with the closure of the horizontal bar relation among linear logic sequents, the entailment of linear logic is indeed reflexive, monotonic and transitive. This idea is also supported by the categorical models for linear logic [55,35], in which sequents are interpreted as morphisms, and leads to a very natural correspondence between the models of rewriting and linear logic.

We use the syntax of the Maude language to write down the map of entailment systems from linear logic to rewriting logic. Note that any sequence of characters starting with either "`---`" or "`***`" and ending with "`end-of-line`" is a comment. Also, from now on, we usually drop the equivalence class square brackets, adopting the convention that a term t denotes the equivalence class $[t]_E$ for the appropriate set of structural axioms E.

We first define the *functional* theory `PROP0[X]` which introduces the syntax of propositions as a parameterized abstract data type. The parameterization permits having additional structure at the level of atoms if desired. In order to provide a proper treatment of negation, only equations are given, and no rewrite rules are introduced in this theory; they are introduced afterwards in the `LINLOG[X]` theory. The purpose of the equations in the `PROP0[X]` theory is to push negation to the atom level, by using the dualities of linear logic; this is a well-known process in classical and linear logic.

```
fth ATOM is
  sort Atom .
endft

fth PROP0[X :: ATOM] is
  sort Prop0 .
  subsort Atom < Prop0 .
  ops 1 0 ⊥ ⊤ : -> Prop0 .
  op _⊥ : Prop0 -> Prop0 .
  op _⊗_ : Prop0 Prop0 -> Prop0 [assoc comm id: 1] .
  op _⅋_ : Prop0 Prop0 -> Prop0 [assoc comm id: ⊥] .
  op _⊕_ : Prop0 Prop0 -> Prop0 [assoc comm id: 0] .
  op _&_ : Prop0 Prop0 -> Prop0 [assoc comm id: ⊤] .
  op !_ : Prop0 -> Prop0 .
  op ?_ : Prop0 -> Prop0 .

  vars A B : Prop0 .
  eq (A ⊗ B)⊥ = A⊥ ⅋ B⊥ .
  eq (A ⅋ B)⊥ = A⊥ ⊗ B⊥ .
  eq (A & B)⊥ = A⊥ ⊕ B⊥ .
  eq (A ⊕ B)⊥ = A⊥ & B⊥ .
  eq (!A)⊥ = ?(A⊥) .
```

```
  eq (?A)^⊥ = !(A^⊥) .
  eq A^⊥^⊥ = A .
  eq 1^⊥ = ⊥ .
  eq ⊥^⊥ = 1 .
  eq ⊤^⊥ = 0 .
  eq 0^⊥ = ⊤ .
endft
```

The `LINLOG[X]` theory introduces linear logic propositions and the rules of the logic. Propositions are of the form `[A]` for `A` an expression in `Prop0`. All logical connectives work similarly for `Prop0` expressions and for propositions, except negation, which is defined only for `Prop0` expressions.

Some presentations of linear logic are given in the form of one-sided sequents $\vdash \Gamma$ where negation has been pushed to the atom level, and there are no rules for negation in the sequent calculus [18]. In this section, in order to make the connections with category theory and with rewriting logic more direct, we prefer to use standard sequents of the more general form $\Gamma \vdash \Delta$. In Section 6, we will also use one-sided sequents just in order to reduce the number of rules.

The style of our formulation adopts a categorical viewpoint for the proof theory and semantics of linear logic [55,35]). This style exploits the close connection between the models of linear logic and those of rewriting logic which are also categories, as we have mentioned in Section 3. When seeking the minimal categorical structure required for interpreting linear logic, an important question is how to interpret the connective $\parr$ without using negation, and how to axiomatize its relationship with the tensor $\otimes$. Cockett and Seely have answered this question with the notion of a *weakly distributive category* [11]. A weakly distributive category consists of a category $\mathcal{C}$ with two symmetric tensor products $\otimes, \parr \colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, and a natural transformation $A \otimes (B \parr C) \longrightarrow (A \otimes B) \parr B$ (weak distributivity) satisfying some coherence equations [7]. Negation is added to a weakly distributive category by means of a function $(_)^{\perp} : |\mathcal{C}| \to |\mathcal{C}|$ on the objects of $\mathcal{C}$, and natural transformations $1 \longrightarrow A \parr A^{\perp}$ and $A \otimes A^{\perp} \longrightarrow \perp$ satisfying some coherence equations.

In the following theory, the rewrite rules for $\otimes$, $\parr$ and negation correspond to the natural transformations in the definition of a weakly distributive category, as explained above. The rules for & ($\oplus$, respectively) mirror the usual definition of final object and product (initial object and coproduct, respectively). Finally, the axioms and rules for the exponential ! (?, respectively) correspond to a comonad with a comonoid structure (monad with monoid structure, respectively). Note that some rules are redundant, but we have decided to include them in order to make the connectives less interdependent, so that, for example, if the connective & is omitted we do not need to add new rules for the modality !.

[7] Cockett and Seely develop in [11] the more general case in which the tensor products are not assumed to be symmetric.

```
th LINLOG[X :: ATOM] is
  protecting PROP0[X] .
  sort Prop .
  ops 1 0 ⊥ ⊤ : -> Prop .
  op _⊗_ : Prop Prop -> Prop [assoc comm id: 1] .
  op _⅋_ : Prop Prop -> Prop [assoc comm id: ⊥] .
  op _⊕_ : Prop Prop -> Prop [assoc comm id: 0] .
  op _&_ : Prop Prop -> Prop [assoc comm id: ⊤] .
  op !_ : Prop -> Prop .
  op ?_ : Prop -> Prop .

  op [_] : Prop0 -> Prop .

  vars A B : Prop0 .
  ax [A ⊗ B] = [A] ⊗ [B] .
  ax [A ⅋ B] = [A] ⅋ [B] .
  ax [A & B] = [A] & [B] .
  ax [A ⊕ B] = [A] ⊕ [B] .
  ax [!A] = ![A] .
  ax [?A] = ?[A] .
  ax [1] = 1 .
  ax [⊥] = ⊥ .
  ax [⊤] = ⊤ .
  ax [0] = 0 .

  *** [_] is injective
  cax A = B if [A] = [B] .

  *** Rules for negation
  rl 1 => [A] ⅋ [A^⊥] .
  rl [A] ⊗ [A^⊥] => ⊥ .

  vars P Q R : Prop .
  *** Rules for ⊗ and ⅋
  rl P ⊗ (Q ⅋ R) => (P ⊗ Q) ⅋ R .

  *** Rules for &
  rl P => ⊤ .                                  *** (1)
  rl P & Q => P .
  crl R => P & Q  if  R => P  and  R => Q .  *** (2)

  *** Rules for ⊕
  rl 0 => P .                                   *** (3)
  rl P => P ⊕ Q .
  crl P ⊕ Q => R  if  P => R  and  Q => R .  *** (4)
```

```
  *** Structural axioms and rules for !
  ax !(P & Q) = !P ⊗ !Q .                        *** (5)
  ax !⊤ = 1 .                                    *** (6)

  rl !P => P .
  rl !P => !!P .
  rl !P => 1 .          *** redundant from (1) and (6) above
  rl !P => !P ⊗ !P .    *** redundant from (2) and (5) above

  *** Structural axioms and rules for ?
  ax ?(P ⊕ Q) = ?P ⅋ ?Q .                        *** (7)
  ax ?0 = ⊥ .                                    *** (8)

  rl P => ?P .
  rl ??P => ?P .
  rl ⊥ => ?P .          *** redundant from (3) and (8) above
  rl ?P ⅋ ?P => ?P .    *** redundant from (4) and (7) above
endt
```

A linear logic formula is built from a set of propositional constants using the logical constants and connectives of linear logic. Notice that linear implication $A \multimap B$ is not necessary because it can be defined as $A^{\perp} \parr B$.

A *linear theory* T in propositional linear logic consists of a finite set C of propositional constants and a finite set of sequents of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, where each A_i and B_j is a linear logic formula built from the constants in C. Given such a theory T, it is interpreted in rewriting logic as follows.

First, we define a functional theory to interpret the propositional constants in C. For example, if $C = \{a, b, c\}$ we would define

```
fth C is
  sort Atom .
  ops a b c : -> Atom .
endft
```

Then, we can instantiate the parameterized theory `LINLOG[X]` using this functional theory, with the default view `ATOM` $\rightarrow$ `C`:

```
  make LINLOG0 is LINLOG[C] endmk
```

A linear logic formula A (with constants in C) is interpreted in `LINLOG0` as the term `[A]` of sort `Prop`. For example, the formula $(a \otimes b)^{\perp} \oplus (!(a \,\&\, c^{\perp}))^{\perp}$ is interpreted as the term

```
  [(a ⊗ b)^⊥ ⊕ (!(a & c^⊥))^⊥]
```

which is equal to the following term, using the equations in `PROP0[X]` and the structural axioms in `LINLOG[X]`,

```
  ([a^⊥] ⅋ [b^⊥]) ⊕ ?([a^⊥] ⊕ [c]).
```

Finally, we extend the theory `LINLOG0` by adding a rule

```
rl [A1] ⊗ ... ⊗ [An] => [B1] ⅋ ... ⅋ [Bm] .
```

for each sequent $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ in the linear theory T. For example, if T consists of the two sequents

$$a \otimes b, !c \oplus a \vdash a, (c \oplus b)^{\perp}$$

$$a \parr b, ?(c^{\perp}) \vdash (?b \parr !c)^{\perp}, a \oplus b,$$

the corresponding rewrite theory is

```
th LINLOG(T) is
  extending LINLOG0 .
  rl [a] ⊗ [b] ⊗ (![c] ⊕ [a]) => [a] ⅋ ([c⊥] & [b⊥]) .
  rl ([a] ⅋ [b]) ⊗ ?[c⊥] => (![b⊥] ⊗ ?[c⊥]) ⅋ ([a] ⊕ [b]) .
endt
```

The main result is the following conservativity theorem:

Theorem 4.1 *Given a linear theory T, a sequent $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ is provable in linear logic from the axioms in T if and only if the sequent*

```
[A1] ⊗ ... ⊗ [An] ⟶ [B1] ⅋ ... ⅋ [Bm]
```

is a `LINLOG(T)`*-rewrite, i.e., it is provable in rewriting logic from the rewrite theory* `LINLOG(T)`.

Thus, we have a map of entailment systems between linear logic and rewriting logic, which is *conservative*. Moreover, Section 4.3.3 of [36] explains in detail how to obtain from a model of the rewriting logic theory `LINLOG(T)` a model of the linear logic theory T, in such a way as to extend this map of entailment systems to a *conservative map of logics* $\Phi : \mathit{LinLogic} \longrightarrow \mathit{OSRWLogic}$.

5 Quantifiers

In Section 4 we have defined a map of logics between quantifier-free linear logic and rewriting logic. In this section, we show a technique that can be used to extend that map at the level of entailment systems to quantifiers. Our equational treatment of quantification, inspired by ideas of Laneve and Montanari on the definition of the lambda calculus as a theory in rewriting logic [33], is very general and encompasses not only existential and universal quantification, but also lambda abstraction and other such binding mechanisms.

The main idea is to internalize as operations in the theory the notions of free variables and substitution that are usually defined at the metalevel. Then, the typical definitions of such notions by structural induction on terms can be easily written down as equations in the theory, but, more importantly, we can consider terms modulo these axioms and we can also use the operation of substitution explicitly in the rules introducing or eliminating quantifiers. This is similar to the lambda calculus with explicit substitutions defined by Abadi, Cardelli, Curien, and Lévy in [1].

We only present here the example of the lambda abstraction binding mechanism in the lambda calculus, as defined by Laneve and Montanari in [33]. We assume a parameterized functional module `SET[X]` that provides finite sets over a parameter set `X` with operations `_U_` for union, `_-_` for set difference, {_} for singleton, `emptyset` for the empty set, and a predicate `_is-in_` for membership.

```
fth VAR is
  sort Var .
  protecting SET[Var] .
  op new: Set -> Var .
  var S : Set .
  eq new(S) is-in S = false .  *** new variable
endft

fmod LAMBDA[X :: VAR] is
  extending SET[X] .
  sort Lambda .
  subsort Var < Lambda .                        *** variables
  op λ_._ : Var Lambda -> Lambda .      *** lambda abstraction
  op __ : Lambda Lambda -> Lambda .             *** application
  op _[_/_] : Lambda Lambda Var -> Lambda .  *** substitution
  op fv : Lambda -> Set .                       *** free variables

  vars X Y : Var .
  vars M N P : Lambda .

  *** Free variables
  eq fv(X) = {X} .
  eq fv(λX.M) = fv(M) - {X} .
  eq fv(MN) = fv(M) U fv(N) .
  eq fv(M[N/X]) = (fv(M) - {X}) U fv(N) .

  *** Substitution equations
  eq X[N/X] = N .
  ceq Y[N/X] = Y  if  not(X == Y) .
  eq (MN)[P/X] = (M[P/X])(N[P/X]) .
  eq (λX.M)[N/X] = λX.M .
  ceq (λY.M)[N/X] = λY.(M[N/X])   if not(X == Y) and
                (not(Y is-in fv(N)) or not(X is-in fv(M))) .
  ceq (λY.M)[N/X] = λ(new(fv(MN))).((M[new(fv(MN))/Y])[N/X])
       if not(X == Y) and Y is-in fv(N) and X is-in fv(M) .
endfm
```

Note that substitution is here another term constructor instead of a meta-syntactic operation. Of course, using the above equations, all occurrences of the substitution constructor can be eliminated. After having defined in the previous functional module the class of lambda terms with substitution, we

just need to add the equational axiom of alpha conversion and the beta rule in the following module:

```
mod ALPHA-BETA[X :: VAR] is
  extending LAMBDA[X] .
  vars X Y : Var .
  vars M N : Lambda .

  *** Alpha conversion
  cax λX.M = λY.(M[Y/X])  if  not(Y is-in fv(M)) .

  *** Beta reduction
  rl (λX.M)N => M[N/X] .
endm
```

In order to introduce quantifiers, we can develop a completely similar approach, by first introducing substitution in the syntax together with the quantifiers, and then adding rewrite rules for the new connectives. A detailed application of this technique to first-order linear logic is developed in Section 4.4 of [36].

6 Sequent systems

In Section 4, we have mapped linear logic formulas to terms, and linear logic sequents to rewrite rules in rewriting logic. There is another map of entailment systems between linear logic and rewriting logic in which linear sequents become also terms, and rewrite rules correspond to rules in a Gentzen sequent calculus for linear logic. In order to reduce the number of rules of this calculus, we consider one-sided linear sequents in this section, but a completely similar treatment can be given for two-sided sequents. Thus, a linear logic sequent will be a turnstile symbol "$\vdash$" followed by a multiset M of linear logic formulas, that in our translation to rewriting logic will be represented by the term $\vdash$ `M`. Using the duality of linear logic negation, a two-sided sequent $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ can in this notation be expressed as the one-sided sequent $\vdash A_1^{\perp}, \ldots, A_n^{\perp}, B_1, \ldots, B_m$.

First, we define a parameterized module for multisets.

```
fth ELEM is
  sort Elem .
endft

fmod MSET[X :: ELEM] is
  sort Mset .
  subsort Elem < Mset .
  op null : -> Mset .
  op _,_ : Mset Mset -> Mset [assoc comm id: null] .
endfm
```

Now we can use this parameterized module to define the main module for

sequents [8] and give the corresponding rules. A sequent calculus rule of the form

$$\frac{\vdash M_1, \ldots, \vdash M_n}{\vdash M}$$

becomes the rewrite rule

```
rl ⊢ M1 ... ⊢ Mn => ⊢ M .
```

on the sort `Configuration`. Recalling that "`---`" introduces a comment, this rule can be written as

```
rl    ⊢ M1 ... ⊢ Mn
   => --- ----------
          ⊢ M .
```

In the module below, we assume a module `FO-PROP0[X]` extending the syntax of propositional linear logic in `PROP0[X]` to the syntax of first-order linear logic, by means of the technique described in Section 5. See [36, Section 4.4] for the complete details.

```
mod LL-SEQUENT[X :: VAR] is
  protecting FO-PROP0[X] .
  extending MSET[FO-PROP0[X]] .
  --- a configuration is a multiset of sequents
  sort Configuration .
  op ⊢_ : Mset -> Configuration .
  op empty : -> Configuration .
  op __ : Configuration Configuration -> Configuration
                                       [assoc comm id: empty] .
  op ?_ : Mset -> Mset .
  vars M N : Mset .
  ax ?null = null .
  ax ?(M,N) = (?M,?N) .
  op fv : Mset -> Set .
  ax fv(null) = emptyset .
  ax fv(M,N) = fv(M) U fv(N) .

  var P : Atom .
  vars A B : Prop0 .
  var T : Term .
  var X : Var .

  *** Identity                        *** Cut
  rl      empty                       rl     (⊢ M,A) (⊢ N,A^⊥)
      => --- ----                         => --- -------------
         ⊢ P,P^⊥ .                              ⊢ M,N .
```

[8] The multiset structure is one particular way of building in certain *structural rules*, in this case *exchange*. Many other such data structuring mechanisms are as well possible to build in, or to drop, desired structural properties.

```
*** Tensor                          *** Par
rl     (⊢ M,A) (⊢ B,N)             rl      ⊢ M,A,B
    => --- -----------                 => --- -------
        ⊢ M,A ⊗ B,N .                     ⊢ M,A ⅋ B .

*** Plus                            *** With
rl        ⊢ M,A                    rl     (⊢ M,A) (⊢ M,B)
    => --- -------                     => --- ------------
       ⊢ M,A ⊕ B .                          ⊢ M,A & B .

*** Weakening                       *** Contraction
rl       ⊢ M                       rl     ⊢ M,?A,?A
    => --- ----                        => --- ------
       ⊢ M,?A .                            ⊢ M,?A .

*** Dereliction                     *** Storage
rl      ⊢ M,A                      rl      ⊢ ?M,A
    => --- ----                        => --- -----
       ⊢ M,?A .                            ⊢ ?M,!A .

*** Bottom            *** One              *** Top
rl       ⊢ M          rl     empty         rl      empty
    => --- ---           => --- --            => --- ----
       ⊢ M,⊥ .              ⊢ 1 .                ⊢ M,⊤ .

*** Universal                       *** Existential
crl      ⊢ M,A                     rl     ⊢ M,A[T/X]
    => --- -----                       => --- ------
       ⊢ M,∀X.A                            ⊢ M.∃X.A .
    if  not(X is-in fv(M)) .
```

```
endm
```

Given a linear theory $T = (C, S)$ (where we can assume that all the sequents in S are of the form $\vdash A_1, \ldots, A_n$), we instantiate the parameterized module `LL-SEQUENT[X]` using a functional module `C` that interprets the propositional constants in C, as in Section 4, and then extend it by adding a rule

```
rl empty => ⊢ A1,...,An .
```

for each sequent $\vdash A_1, \ldots, A_n$ in S, obtaining in this way a rewrite theory `LL-SEQUENT(T)`.

With this map we have also an immediate conservativity result:

Theorem 6.1 *Given a linear theory T, a linear logic sequent $\vdash A_1, \ldots, A_n$ is provable in linear logic from the axioms in T if and only if the sequent*

```
empty ⟶ ⊢ A1,...,An
```

is provable in rewriting logic from the rewrite theory `LL-SEQUENT(T)`.

It is very important to realize that the technique used in this conservative map of entailment systems is very general and it is in no way restricted to linear logic. Indeed, it can be applied to any sequent calculus, be it for intuitionistic, classical or any other logic. Moreover, it can even be applied to systems more general than traditional sequent calculi. Thus, a "sequent" can for example be a sequent presentation of natural deduction, a term assignment system, or even any predicate defined by structural induction in some way such that the proof is a kind of tree, as for example the operational semantics of CCS given later in Section 8 and any other use of the so-called structural operational semantics (see [53,27]). The general idea is to map a rule in the "sequent" system to a rewrite rule over a "configuration" of sequents or predicates, in such a way that the rewriting relation corresponds to provability of such a predicate.

7 Reflection

We give here a brief summary of the notion of a universal theory in a logic and of a reflective entailment system introduced in [9]. These notions axiomatize reflective logics within the theory of general logics [39]. We focus here on the simplest case, namely entailment systems. However, reflection at the proof calculus level—where not only sentences, but also proofs are reflected—is also very useful; the adequate definitions for that case are also in [9].

A reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. Two obvious metatheoretic notions that can be so reflected are theories and the entailment relation $\vdash$. This leads us to the notion of a universal theory. However, universality may not be absolute, but only relative to a class $\mathcal{C}$ of *representable* theories. Typically, for a theory to be representable at the object level, it must have a finitary description in some way—say, being recursively enumerable—so that it can be represented as a piece of language.

Given an entailment system $\mathcal{E}$ and a set of theories $\mathcal{C}$, a theory U is $\mathcal{C}$-*universal* if there is a function, called a *representation function*,

$$\overline{(_ \vdash _)} : \bigcup_{T \in \mathcal{C}} \{T\} \times sen(T) \longrightarrow sen(U)$$

such that for each $T \in \mathcal{C}$, $\varphi \in sen(T)$,

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi}.$$

If, in addition, $U \in \mathcal{C}$, then the entailment system $\mathcal{E}$ is called $\mathcal{C}$-*reflective*.

Note that in a reflective entailment system, since U itself is representable, representation can be iterated, so that we immediately have a "reflective tower"

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi} \iff U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \ \ldots$$

Clavel and Meseguer have shown in [9,10] that indeed rewriting logic is reflective. That is, there is a rewrite theory $\mathcal{U}$ with a finite number of operations and rules that can simulate any other finitely presentable rewrite theory $\mathcal{R}$ in

the following sense: given any two terms t, t' in $\mathcal{R}$, there are corresponding terms $\langle \overline{\mathcal{R}}, \overline{t} \rangle$ and $\langle \overline{\mathcal{R}}, \overline{t'} \rangle$ in $\mathcal{U}$ such that we have

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Moreover, it is often possible to reify inside rewriting logic itself a representation map $\mathcal{L} \to \mathit{OSRWLogic}$ for the finitely presentable theories of $\mathcal{L}$. Such a reification takes the form of a map between the abstract data types representing the finitary theories of $\mathcal{L}$ and of *OSRWLogic*. In this section we illustrate this powerful idea with the linear logic mapping defined in Section 4.

We have defined a linear theory T as a finite set C of propositional constants together with a finite set S of sequents of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, where each A_i and B_j is a linear logic formula built from the constants in C. Note that with this definition, all linear theories are finitely presentable. First, we define an abstract data type `LL-ADT` to represent linear theories. A linear theory is represented as a term `<C | G>`, where `C` is a list of propositional constants (that is, identifiers), and `G` is a list of sequents written in the usual way. Moreover, all the propositional constants in `G` must be included in `C`. To enforce this condition, we use a sort constraint [44], which is introduced with the keyword `sct` and defines a subsort `LLTheory` of a sort `LLTheory?` by means of the given condition. In the functional module below, we do not give the equations defining the auxiliary functions `const` that extracts the constants of a list of sequents, and the list containment predicate `_=<_`. These functions are needed to write down the sort constraint for theories.

```
fmod LL-ADT is
  protecting QID .
  sorts Ids Formula Formulas Sequent .
  sorts Sequents LLTheory? LLTheory .

  subsort Id < Formula .
  ops 1 0 ⊥ ⊤ : -> Formula .
  op _⊗_ : Formula Formula -> Formula .
  op _⅋_ : Formula Formula -> Formula .
  op _⊕_ : Formula Formula -> Formula .
  op _&_ : Formula Formula -> Formula .
  op !_ : Formula -> Formula .
  op ?_ : Formula -> Formula .
  op _⊥ : Formula -> Formula .

  subsort Formula < Formulas .
  op null : -> Formulas .
  op _,_ : Formulas Formulas -> Formulas [assoc comm id: null] .

  op _⊢_ : Formulas Formulas -> Sequent .

  subsort Id < Ids .
  op nil : -> Ids .
```

```
  op _,_ : Ids Ids -> Ids [assoc id: nil] .

  subsort Sequent < Sequents .
  op nil : -> Sequents .
  op _,_ : Sequents Sequents -> Sequents [assoc id: nil] .

  op <_|_> : Ids Sequents -> LLTheory? .

  var C : Ids .
  var G : Sequents .
  sct <C | G> : LLTheory if const(G) =< C .

  eq ...
  *** several equations defining the auxiliary operations
  *** "const" and "_=<_" used in the sort constraint condition
  eq ...
endfm
```

An order-sorted rewrite theory has much more structure, and therefore the corresponding `RWL-ADT` is more complex, but the basic ideas are completely similar as we sketch here. First we have an order-sorted signature, declaring sorts, subsorts, constants, operations and variables. Then, in addition, we have equations and rules. Thus, a finitely presentable rewrite theory is represented as a term `<S | E | R>`, where `S` is a term representing a signature, `E` is a list of equations, and `R` is a list of rules. In turn, the term `S` has the form `<T ; B ; C ; O ; V>` where each subterm corresponds to a component of a signature as mentioned before. In addition, several sort constraints are necessary to ensure for example that the variables used in equations and rules are included in the list of variables. Just to give the flavor of the construction, here is a small fragment of the module `RWL-ADT`, where we have omitted most of the list constructors, operations to handle conditional equations and rules, and sort constraints.

```
sorts Sort Subsort Constant Op Var .
sorts Term Equation Rule Signature RWLTheory .

op sort{_} : Id -> Sort .
subsort Sort < Sorts .
op nil : -> Sorts .
op __ : Sorts Sorts -> Sorts [assoc id: nil] .

op (_<_) : Id Id -> Subsort .
subsort Subsort < Subsorts .

op (cons{_}:sort{_}) : Id Id -> Constant .
subsort Constant < Constants .
op nil : -> Constants .
op _,_ : Constants Constants -> Sorts [assoc id: nil] .
```

```
op (op{_}:_->sort{_}) : Id Sorts Id -> Op .
subsort Op < Ops .

op (var{_}:sort{_}) : Id Id -> Var .
subsort Var < Vars .

op <_;_;_;_;_> : Sorts Subsorts Constants Ops Vars
                                              -> Signature .
subsort Var < Term .
subsort Constant < Term .
subsort Term < Terms .
op nil : -> Terms .
op op{_}[_] : Id Terms -> Term .
op _,_ : Terms Terms -> Terms [assoc id: nil] .

op (_=_) : Term Term -> Equation .
subsort Equation < Equations .

op (_=>_) : Term Term ->  Rule .
subsort Rule < Rules .

op <_|_|_> : Signature Equations Rules -> RWLTheory .
```

In this way, the rewrite theory `LINLOG` presented in Section 4 gives rise to a term in `RWLTheory` that we denote

$$\texttt{<<}\mathtt{T}_{LL}\ \texttt{;}\ \mathtt{B}_{LL}\ \texttt{;}\ \mathtt{C}_{LL}\ \texttt{;}\ \mathtt{O}_{LL}\ \texttt{;}\ \mathtt{V}_{LL}\texttt{>}\ \texttt{|}\ \mathtt{E}_{LL}\ \texttt{|}\ \mathtt{R}_{LL}\texttt{>}.$$

Having defined the abstract data types to represent both linear and rewrite theories, we define a function $\overline{\Phi}$ mapping a term in `LLTheory` representing a linear theory T to a term in `RWLTheory` representing the corresponding rewrite theory `LINLOG(T)` as defined in Section 4.

The representation `<C | F1` $\vdash$ `G1,...,Fn` $\vdash$ `Gn>` is mapped by $\overline{\Phi}$ to the following term

$$\texttt{<<}\mathtt{T}_{LL}\ \texttt{;}\ \mathtt{B}_{LL}\ \texttt{;}\ \texttt{cons(C),}\mathtt{C}_{LL}\ \texttt{;}\ \mathtt{O}_{LL}\ \texttt{;}\ \mathtt{V}_{LL}\texttt{>}\ \texttt{|}\ \mathtt{E}_{LL}\ \texttt{|}$$
$$\mathtt{R}_{LL}\texttt{,([tensor(F1)] => [par(G1)]),...,([tensor(Fn)] => [par(Gn)])>}$$

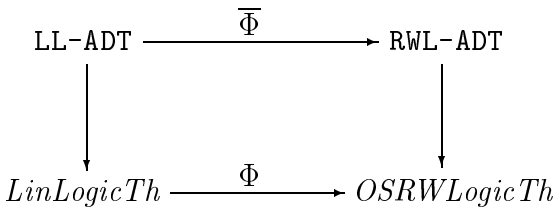
where the auxiliary operations `cons`, `tensor` and `par` are defined as follows, and correspond exactly to the description in Section 4.

```
op tensor : Formulas -> Formula .
op par : Formulas -> Formula .
op cons : Ids -> Constants .

vars F1 F2 : Formulas .
var I : Id .
var L : Ids .
```

```
eq tensor(null) = 1 .
eq tensor(F1,F2) = tensor(F1) ⊗ tensor(F2) .
eq par(null) = ⊥ .
eq par(F1,F2) = par(F1) ⅋ par(F2) .
eq cons(nil) = nil .
eq cons(I,L) = (cons{I}:sort{Atom}),cons(L) .
```

We can summarize the reification $\overline{\Phi}$: `LL-ADT` $\longrightarrow$ `RWL-ADT` of the map of logics Φ : *LinLogic* $\longrightarrow$ *OSRWLogic* we have just defined by means of the following commutative diagram:



This method is completely general, in that it should apply to any effectively presented map of logics $\Psi : \mathcal{L} \longrightarrow$ *RWLogic* that maps finitely presentable theories in $\mathcal{L}$ to finitely presentable theories in rewriting logic. Indeed, the effectiveness of Ψ should exactly mean that the corresponding $\overline{\Psi}$: $\mathcal{L}$`-ADT` $\longrightarrow$ `RWL-ADT` is a computable function and therefore, by the metatheorem of Bergstra and Tucker [4], that it is specifiable by a finite set of Church-Rosser and terminating equations inside rewriting logic.

8 CCS

Milner's *Calculus of Communicating Systems* (CCS) [47,48] is among the best well-known and studied concurrency models, and has become the paradigmatic example of an entire approach to "process algebras." We give a very brief introduction to CCS, referring the reader to Milner's book [48] for motivation and a comprehensive treatment; then, we give a formulation of CCS in rewriting logic and show its conservativity.

We assume a set A of *names*; the elements of the set $\overline{A} = \{\overline{a} \mid a \in A\}$ are called *co-names*, and the members of the (disjoint) union $\mathcal{L} = A \cup \overline{A}$ are *labels* naming ordinary actions. The function $a \mapsto \overline{a}$ is extended to $\mathcal{L}$ by defining $\overline{\overline{a}} = a$. There is a special action called *silent action* and denoted τ, intended to represent internal behaviour of a system, and in particular the synchronization of two processes by means of actions a and $\overline{a}$. Then the set of *actions* is $\mathcal{L} \cup \{\tau\}$. The set of processes is intuitively defined as follows:

- 0 is an inactive process that does nothing.
- If α is an action and P is a process, $\alpha.P$ is the process that performs α and subsequently behaves as P.
- If P and Q are processes, $P + Q$ is the process that may behave as either P or Q.
- If P and Q are processes, $P|Q$ represents P and Q running concurrently

with possible communication via synchronization of the pair of ordinary actions a and $\overline{a}$.

- If P is a process and $f : \mathcal{L} \to \mathcal{L}$ is a relabelling function such that $f(\overline{a}) = \overline{f(a)}$, $P[f]$ is the process that behaves as P but with the actions relabelled according to f, assuming $f(\tau) = \tau$.
- If P is a process and $L \subseteq \mathcal{L}$ is a set of ordinary actions, $P\backslash L$ is the process that behaves as P but with the actions in $L \cup \overline{L}$ prohibited.
- If P is a process, I is a process identifier, and $I =_{def} P$ is a defining equation where P may recursively involve I, then I is a process that behaves as P.

This intuitive explanation can be made precise in terms of the following structural operational semantics that defines a labelled transition system for CCS processes.

Action:

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

Summation:

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

Composition:

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

Relabelling:

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

Restriction:

$$\frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L} \quad \alpha \notin L \cup \overline{L}$$

Definition:

$$\frac{P \xrightarrow{\alpha} P'}{I \xrightarrow{\alpha} P'} \quad I =_{def} P$$

We now show how CCS can be described and given semantics in rewriting logic.

```
fth LABEL is
  sort Label . *** ordinary actions
  op ~_ : Label -> Label .
  var N : Label .
  eq ~~N = N .
endft
```

```
fmod ACTION[X :: LABEL] is
  sort Act .
  subsort Label < Act .
  op tau : -> Act .  *** silent action
endfm

fth PROCESSID is
  sort ProcessId . *** process identifiers
endft

fmod PROCESS[X :: LABEL, Y :: PROCESSID] is
  protecting ACTION[X] .
  sort Process .
  subsort ProcessId < Process .
  op 0 : -> Process .                          *** inaction
  op _._ : Act Process -> Process .            *** prefix
  op _+_ : Process Process -> Process [assoc comm idem id: 0] .
                                               *** summation
  op _|_ : Process Process -> Process [assoc comm id: 0] .
                                               *** composition
  op _[_/_] : Process Label Label -> Process .
                      *** relabelling: [b/a] relabels "a" to "b"
  op _\_ : Process Label -> Process .  *** restriction
endfm
```

Before defining the operational semantics of CCS processes, we need an auxiliary module in order to build contexts in which process identifiers can be associated with processes, providing in this way recursive definitions of processes. A sort constraint [44], which is introduced with the keyword `sct` and defines a subsort `Context` by means of a condition, is used to enforce the requirement that the same process identifier cannot be associated with two different processes in a context.

```
fmod CCS-CONTEXT[X :: LABEL, Y :: PROCESSID] is
  protecting PROCESS[X,Y] .
  sorts Def Context .
  op (_ =def _) : ProcessId Process -> Def .
  protecting LIST[ProcessId]*(op _;_ to __) .
  protecting LIST[Def]*(sort List to Context?) .
  subsorts Def < Context < Context? .
  op nil : -> Context .
  op pid : Context? -> List .
  var X : ProcessId .
  var P : Process .
  var C : Context .
  vars D D' : Context? .
  eq pid(nil) = nil .
```

```
  eq pid((X =def P)) = X .
  eq pid(D;D') = pid(D) pid(D') .
  sct (X =def P);C : Context if not(X in pid(C)) .
endfm
```

The semantics of CCS processes is usually defined relative to a given context that provides defining equations for all the necessary process identifiers [48, Section 2.4]. The previous module defines the data type of all contexts. We now need to parameterize the module defining the CCS semantics by the choice of a context. This is accomplished by means of the following theory that picks up a context in the sort `Context`.

```
fth CCS-CONTEXT*[X :: LABEL, Y :: PROCESSID] is
  protecting CCS-CONTEXT[X,Y] .
  op context : -> Context .
endft
```

As in the case of linear logic, we have two possibilities in order to write the operational semantics for CCS by means of rewrite rules. On the one hand, we can interpret a transition $P \xrightarrow{\alpha} P'$ as a rewrite, so that the above operational semantics rules become conditional rewrite rules. On the other hand, the transition $P \xrightarrow{\alpha} P'$ can be seen as a term, forming part of a configuration, in such a way that the semantics rules correspond to rewrite rules, as a particular case of the general mapping of sequent systems into rewriting logic that we have presented in Section 6. Here we describe the first possibility and refer the reader to [36, Section 5.3] for the second one.

```
mod CCS1[X :: LABEL, Y :: PROCESSID, C :: CCS-CONTEXT*[X,Y]] is
  sort ActProcess .
  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess .
  *** {A}P means that the process P has performed the action A
  vars L M : Label .
  var A : Act .
  vars P P' Q Q' : Process .
  var X : ProcessId .
  *** Prefix
  rl A . P => {A}P .

  *** Summation
  crl P + Q => {A}P'  if  P => {A}P' .

  *** Composition
  crl P | Q => {A}(P' | Q)  if  P => {A}P' .
  crl P | Q => {tau}(P' | Q')  if  P => {L}P' and Q => {~L}Q' .

  *** Restriction
  crl P \ L => {A}(P' \ L)  if  P => {A}P' and
                                not(A == L) and not(A == ~L) .
```

```
  *** Relabelling
  crl P[M / L] => {M}(P'[M / L])  if  P => {L}P' .
  crl P[M / L] => {~M}(P'[M / L])  if  P => {~L}P' .
  crl P[M / L] => {A}(P'[M / L])  if  P => {A}P' and
                                  not(A == L) and not(A == ~L) .
  *** Definition
  crl X => {A}P'  if  (X =def P) in context and P => {A}P' .
endm
```

In the above module, the rewrite rules have the property of being sort-increasing, i.e., in a rule $[t] \longrightarrow [t']$ the least sort of $[t']$ is bigger than the least sort of $[t]$. Thus, one rule cannot be applied unless the resulting term is well-formed. This prevents, for example, rewrites of the following form:

$$\texttt{\{A\}(P | Q)} \longrightarrow \texttt{\{A\}(\{B\}P' | \{C\}Q')}$$

because the term on the righthand side is not well formed according to the order-sorted signature of the module `CCS1[X,Y,C[X,Y]]`. More precisely, the *Congruence* rule of order-sorted rewriting logic, like the corresponding rule of order-sorted algebra [22], cannot be applied unless the resulting terms $f(t_1, \ldots, t_n)$ are well formed according to the given order-sorted signature. To illustrate this point further, although $\texttt{A.P} \longrightarrow \texttt{\{A\}P}$ is a correct instance of the `Prefix` rewrite rule, we cannot use the *Congruence* rule to derive

$$\texttt{(A.P) | Q} \longrightarrow \texttt{(\{A\}P) | Q}$$

because the second term `(\{A\}P) | Q` is not well formed.

The net effect of this restriction is that an `ActProcess` term of the form $\texttt{\{A1\}}\ldots\texttt{\{Ak\}P}$ can only be rewritten into another term of the same form $\texttt{\{A1\}}\ldots\texttt{\{Ak\}\{B\}P'}$, being $\texttt{P} \longrightarrow \texttt{\{B\}P'}$ a `CCS1[X,Y,C[X,Y]]`-rewrite. As another example, a process of the form `A.B.P` can be rewritten first into `\{A\}B.P` and then into `\{A\}\{B\}P`, but cannot be rewritten into $\texttt{A.\{B\}P}$, because this last term is not well formed. After this discussion, it is easy to see that we have the following conservativity result.

Theorem 8.1 *Given a CCS process P, there are processes $P_1, \ldots, P_{k-1}$ such that*

$$P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{k-1}} P_{k-1} \xrightarrow{a_k} P'$$

if and only if `P` *can be rewritten into* $\texttt{\{a1\}}\ldots\texttt{\{ak\}P'}$ *using the rules in the module* `CCS1[X,Y,C[X,Y]]`.

Note also that, since the operators `_+_` and `_|_` are declared commutative, one rule is enough for each one, instead of the two rules in the original presentation. On the other hand, we need three rules for relabelling, due to the representation of the relabelling function.

9 Constraint solving

Deduction can in many cases be made much more efficient by making use of *constraints* that can drastically reduce the search space, and for which

special purpose constraint solving algorithms can be much faster than the alternative of expressing everything in a unique deduction mechanism such as some form of resolution. Typically, constraints are symbolic expressions associated with a particular *theory*, and a constraint solving algorithm uses intimate knowledge about the truths of the theory in question to find solutions for those expressions by transforming them into expressions in *solved form*.

One of the simplest examples is provided by standard syntactic unification, the constraint solver for resolution in first-order logic without equality and in particular for Prolog, where the constraints in question are equalities between terms in a free algebra, i.e., in the so-called Herbrand universe. There are however many other constraints and constraint solving algorithms that can be used to advantage in order to make the representation of problems more expressive and logical deduction more efficient. For example,

- *Semantic unification* (see for example [30]), which corresponds to solving equations in a given equational theory.
- *Sorted unification*, either many-sorted or order-sorted [58,54,45,30], where type constraints are added to variables in equations.
- *Higher-order unification* [28], which corresponds to solving equations between λ-expressions.
- *Disunification* [13], which corresponds to solving not only equalities but also negated equalities.
- *Solution of equalities and inequalities in a numerical theory*, as for example the solution of numerical constraints built into the constraint logic programming language $CLP(\mathcal{R})$ [29] and in other languages.

A remarkable property shared by most constraint-solving processes, and already implicit in the approach to syntactic unification problems proposed by Martelli and Montanari [34], is that the process of solving constraints can be naturally understood as one of applying transformations to a set or multiset of constraints. Furthermore, many authors have realized that the most elegant and simple way to specify, prove correct, or even implement many constraint solving problems is by expressing those transformations as rewrite rules (see for example [21,30,12,13]). In particular, the survey by Jouannaud and Kirchner [30] makes this viewpoint the cornerstone of a unified conceptual approach to unification.

For example, the so-called *decomposition* transformation present in syntactic unification and in a number of other unification algorithms can be expressed by a rewrite rule of the form

```
f(t1,...,tn) =?= f(t'1,...,t'n)
          => (t1 =?= t'1) ...  (tn =?= t'n)
```

where in the righthand side multiset union has been expressed by juxtaposition.

Although the operational semantics of such rewrite rules is very obvious and intuitive, their logical or mathematical semantics has remained ambiguous. Although appeal is sometimes made to equational logic as the framework

in which such rules exist, the fact that many of these rules are nondeterministic, so that, except for a few exceptions such as syntactic unification, there is in general not a unique solution but rather a, sometimes infinite, set of solutions, makes an interpretation of the rewrite rules as equations highly implausible and potentially contradictory.

We would like to suggest that rewriting logic provides a very natural framework in which to interpret rewrite rules of this nature and, more generally, deduction processes that are nondeterministic in nature and involve the exploration of an entire space of solutions. Since in rewriting logic rewrite rules go only in one direction and its models do not assume either the identification of the two sides of a rewrite step, or even the possible reversal of such a step, all the difficulties involved in an equational interpretation disappear.

Such a proposed use of rewriting logic for constraint solving and constraint programming seems very much in the spirit of recent rewrite rule approaches to constrained deduction such as those of C. Kirchner, H. Kirchner, and M. Rusinovitch [31], Bachmair, Ganzinger, Lynch, and Snyder [2], and Nieuwenhuis and Rubio [50]. In particular, the ELAN language of C. Kirchner, H. Kirchner, and M. Vittek [32,5] proposes an approach to the prototyping of constraint solving languages similar in some ways to the one that would be natural using a Maude interpreter.

10 Concluding remarks

Rewriting logic has been proposed as a logical framework that seems particularly promising for representing logics, and its use for this purpose has been illustrated in detail by a number of examples. The general way in which such representations are achieved is by:

- Representing formulas or, more generally, proof-theoretic structures such as sequents, as *terms* in an order-sorted equational data type whose equations express structural axioms natural to the logic in question.
- Representing the rules of deduction of a logic as rewrite rules that transform certain patterns of formulas into other patterns modulo the given structural axioms.

Besides, the theory of general logics [39] has been used as both a method and a criterion of adequacy for defining these representations as conservative maps of logics or of entailment systems. From this point of view, our tentative conclusion is that, at the level of entailment systems, rewriting logic should in fact be able to represent any finitely presented logic via a conservative map, for any reasonable notion of "finitely presented logic." Making this tentative conclusion definite will require proposing an intuitively reasonable formal version of such a notion in a way similar to previous proposals of this kind by Smullyan [56] and Feferman [15].

In some cases, such as for equational logic, Horn logic with equality, and linear logic, we have in fact been able to represent logics in a much stronger sense, namely by conservative maps of logics that also map the models [36]. Of

course, such maps are much more informative, and may afford easier proofs, for example for conservativity. However, one should not expect to find representations of this kind for logics whose model theory is very different from that of rewriting logic.

We have also shown how the fact that rewriting logic is reflective greatly enhances its capabilities as a logical framework, by allowing the metalevel representation maps $\mathcal{L} \longrightarrow RWLogic$ to be reified inside rewriting logic itself.

The uses of rewriting logic as a semantic framework for the specification of languages, systems, and models of computation have also been discussed and illustrated with examples. Such uses include the specification and prototyping of concurrent models of computation and concurrent object-oriented systems, of general programming languages, of automated deduction systems and logic programming languages that use constraints, and of logical representation of action and change in AI [36,42].

From a pragmatic point of view, the main goal of this study is to serve as a guide for the design and implementation of a theoretically-based high-level system in which it can be easy to define logics and to perform deductions in them, and in which a very wide variety of systems, languages, and models of computation can similarly be specified and prototyped. Having this goal in mind, the following features seem particularly useful:

- *Executability*, which is not only very useful for prototyping purposes, but is in practice a must for debugging specifications of any realistic size.
- *User-definable abstract syntax*, which can be specified as an order-sorted equational data type with the desired structural axioms.
- *Modularity and parameterization* [9], which can make specifications very readable and reusable by decomposing them in small understandable pieces that are as general as possible.
- *Simple and general logical semantics*, which can naturally express both logical deductions and concurrent computations.

These features are supported by the Maude interpreter [8]. A very important additional feature that the Maude interpreter has is good support for flexible and expressive strategies of evaluation [8,10], so that the user can explore the space of rewritings in intelligent ways.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, Explicit substitutions, *Journal of Functional Programming* **1**(4) (1991) 375–416.

[2] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder, Basic paramodulation and superposition, in: D. Kapur, ed., *Proc. 11th. Int. Conf. on Automated Deduction, Saratoga Springs, NY* (LNAI 607, Springer-Verlag, 1992) 462–476.

[9] Parameterization is based on the existence of relatively free algebras in rewriting logic, which generalizes the existence of initial algebras.

[3] D. A. Basin and R. L. Constable, Metalogical frameworks, in: G. Huet and G. Plotkin, eds., *Logical Environments* (Cambridge University Press, 1993) 1–29.

[4] J. Bergstra and J. Tucker, Characterization of computable data types by means of a finite equational specification method, in: J. W. de Bakker and J. van Leeuwen, eds., *Proc. 7th. Int. Colloquium on Automata, Languages and Programming* (LNCS 81, Springer-Verlag, 1980) 76–90.

[5] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek, ELAN: A logical framework based on computational systems, in: J. Meseguer, ed., *Proc. First Int. Workshop on Rewriting Logic and its Applications* (ENTCS 4, Elsevier, 1996) [This volume].

[6] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer, Specification and proof in membership equational logic, paper in preparation (1996).

[7] M. Cerioli and J. Meseguer, May I borrow your logic? (Transporting logical structure along maps), to appear in Theoretical Computer Science (1996).

[8] M. G. Clavel, S. Eker, P. Lincoln, and J. Meseguer, Principles of Maude, in: J. Meseguer, ed., *Proc. First Int. Workshop on Rewriting Logic and its Applications* (ENTCS 4, Elsevier, 1996) [This volume].

[9] M. G. Clavel and J. Meseguer, Axiomatizing reflective logics and languages, in: G. Kiczales, ed., *Proc. Reflection'96* (San Francisco, USA, April 1996) 263–288.

[10] M. G. Clavel and J. Meseguer, Reflection in rewriting logic, in: J. Meseguer, ed., *Proc. First Int. Workshop on Rewriting Logic and its Applications* (ENTCS 4, Elsevier, 1996) [This volume].

[11] J. R. B. Cockett and R. A. G. Seely, Weakly distributive categories, in: M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds., *Applications of Categories in Computer Science* (Cambridge University Press, 1992) 45–65.

[12] H. Comon, Equational formulas in order-sorted algebras, in: M. S. Paterson, ed., *Proc. 17th. Int. Colloquium on Automata, Languages and Programming, Warwick, England* (LNCS 443, Springer-Verlag, 1990) 674–688.

[13] H. Comon, Disunification: A survey, in: J.-L. Lassez and G. Plotkin (eds.), *Computational Logic: Essays in Honor of Alan Robinson* (The MIT Press, 1991) 322–359.

[14] N. Dershowitz and J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen *et al.*, eds., *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics* (The MIT Press/Elsevier, 1990) 243–320.

[15] S. Feferman, Finitary inductively presented logics, in: R. Ferro *et al.*, eds., *Logic Colloquium'88* (North-Holland, 1989) 191–220.

[16] A. Felty and D. Miller, Encoding a dependent-type λ-calculus in a logic programming language, in: M. E. Stickel, ed., *Proc. 10th. Int. Conf. on Automated Deduction, Kaiserslautern, Germany* (LNAI 449, Springer-Verlag, 1990) 221–235.

[17] P. Gardner, *Representing Logics in Type Theory*, Ph.D. Thesis, Department of Computer Science, University of Edinburgh (1992).

[18] J.-Y. Girard, Linear logic, *Theoretical Computer Science* **50** (1987) 1–102.

[19] J. A. Goguen and R. M. Burstall, Introducing institutions, in: E. Clarke and D. Kozen, eds., *Proc. Logics of Programming Workshop* (LNCS 164, Springer-Verlag, 1984) 221–256.

[20] J. A. Goguen and R. M. Burstall, Institutions: Abstract model theory for specification and programming, *Journal of the Association for Computing Machinery* **39**(1) (1992) 95–146.

[21] J. A. Goguen and J. Meseguer, Software for the rewrite rule machine, in: *Proc. of the Int. Conf. on Fifth Generation Computer Systems* (ICOT, Tokyo, Japan, 1988) 628–637.

[22] J. A. Goguen and J. Meseguer, Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions, and partial operations, *Theoretical Computer Science* **105** (1992) 217–273.

[23] J. A. Goguen, A. Stevens, K. Hobley, and H. Hilberdink, 2OBJ: A meta-logical framework based on equational logic, *Philosophical Transactions of the Royal Society, Series A* **339** (1992) 69–86.

[24] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, Introducing OBJ, Technical report SRI-CSL-92-03, Computer Science Laboratory, SRI International (March 1992). To appear in J. A. Goguen and G. Malcolm, eds., *Software Engineering with OBJ: Algebraic Specification in Practice* (Cambridge University Press).

[25] R. Harper, F. Honsell, and G. Plotkin, A framework for defining logics, *Journal of the Association for Computing Machinery* **40**(1) (1993) 143–184.

[26] R. Harper, D. Sannella, and A. Tarlecki, Structure and representation in LF, in: *Proc. Fourth Annual IEEE Symp. on Logic in Computer Science* (Asilomar, California, June 1989) 226–237.

[27] M. Hennessy, *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics* (John Wiley and Sons, 1990).

[28] G. Huet, A unification algorithm for typed lambda calculus, *Theoretical Computer Science* **1**(1) (1973) 27–57.

[29] J. Jaffar and J. Lassez, Constraint logic programming, in: *Proc. 14th. ACM Symp. on Principles of Programming Languages* (Munich, Germany, 1987) 111-119.

[30] J.-P. Jouannaud and C. Kirchner, Solving equations in abstract algebras: A rule-based survey of unification, in: J.-L. Lassez and G. Plotkin, eds., *Computational Logic: Essays in Honor of Alan Robinson* (The MIT Press, 1991) 257–321.

[31] C. Kirchner, H. Kirchner, and M. Rusinowitch, Deduction with symbolic constraints, *Revue Francaise d'Intelligence Artificielle* **4**(3) (1990) 9–52.

[32] C. Kirchner, H. Kirchner, and M. Vittek, Designing constraint logic programming languages using computational systems, in: V. Saraswat and P. van Hentenryck, eds., *Principles and Practice of Constraint Systems: The Newport Papers* (The MIT Press, 1995) 133–160.

[33] C. Laneve and U. Montanari, Axiomatizing permutation equivalence in the λ-calculus, in: H. Kirchner and G. Levi, eds., *Proc. Third Int. Conf. on Algebraic and Logic Programming, Volterra, Italy* (LNCS 632, Springer-Verlag, 1992) 350–363.

[34] A. Martelli and U. Montanari, An efficient unification algorithm, *ACM Transactions on Programming Languages and Systems* **4**(2) (1982) 258-282.

[35] N. Martí-Oliet and J. Meseguer, From Petri nets to linear logic through categories: A survey, *International Journal of Foundations of Computer Science* **2**(4) (1991) 297–399.

[36] N. Martí-Oliet and J. Meseguer, Rewriting logic as a logical and semantic framework, Technical report SRI-CSL-93-05, Computer Science Laboratory, SRI International (August 1993). To appear in D. M. Gabbay, ed., *Handbook of Philosophical Logic* (Kluwer Academic Publishers).

[37] N. Martí-Oliet and J. Meseguer, General logics and logical frameworks, in: D. Gabbay, ed., *What Is a Logical System?* (Oxford University Press, 1994) 355–392.

[38] S. Matthews, A. Smaill, and D. Basin, Experience with FS_0 as a framework theory, in: G. Huet and G. Plotkin, eds., *Logical Environments* (Cambridge University Press, 1993) 61–82.

[39] J. Meseguer, General logics, in: H.-D. Ebbinghaus *et al.*, eds., *Logic Colloquium'87* (North-Holland, 1989) 275–329.

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

[41] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Object-Based Concurrency* (The MIT Press, 1993) 314–390.

[42] J. Meseguer, Rewriting logic as a semantic framework for concurrency: A progress report, in: *Proc. CONCUR'96* (LNCS, Springer-Verlag, 1996).

[43] J. Meseguer, Membership algebra, Lecture at the *Dagstuhl Seminar on Specification and Semantics* (July 1996). Extended version in preparation.

[44] J. Meseguer and J. A. Goguen, Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems, *Information and Computation* **104** (1993) 114–158.

[45] J. Meseguer, J. A. Goguen, and G. Smolka, Order-sorted unification, *Journal of Symbolic Computation* **8** (1989) 383–413.

[46] J. Meseguer and T. Winkler, Parallel programming in Maude, in: J. P. Banâtre and D. Le Métayer, eds., *Research Directions in High-Level Parallel Programming Languages* (LNCS 574, Springer-Verlag, 1992) 253–293.

[47] R. Milner, *A Calculus of Communicating Systems* (LNCS 92, Springer-Verlag, 1980).

[48] R. Milner, *Communication and Concurrency* (Prentice Hall, 1989).

[49] G. Nadathur and D. Miller, An overview of λProlog, in: K. Bowen and R. Kowalski, eds., *Fifth Int. Joint Conf. and Symp. on Logic Programming* (The MIT Press, 1988) 810–827.

[50] R. Nieuwenhuis and A. Rubio, Theorem proving with ordering constrained clauses, in: D. Kapur, ed., *Proc. 11th. Int. Conf. on Automated Deduction, Saratoga Springs, NY* (LNAI 607, Springer-Verlag, 1992) 477–491.

[51] L. Paulson, The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5** (1989) 363–397.

[52] F. Pfenning, Elf: A language for logic definition and verified metaprogramming, in: *Proc. Fourth Annual IEEE Symp. on Logic in Computer Science* (Asilomar, California, June 1989) 313–322.

[53] G. D. Plotkin, A structural approach to operational semantics, Technical report DAIMI FN-19, Computer Science Department, Aarhus University (September 1981).

[54] M. Schmidt-Schauss, *Computational Aspects of Order-Sorted Logic with Term Declarations* (LNCS 395, Springer-Verlag, 1989).

[55] R. A. G. Seely, Linear logic, $*$-autonomous categories and cofree coalgebras, in: J. W. Gray and A. Scedrov, eds., *Categories in Computer Science and Logic, Boulder, June 1987* (Contemporary Mathematics 92, AMS, 1989) 371–382.

[56] R. M. Smullyan, *Theory of Formal Systems* (Princeton University Press, 1961).

[57] A. S. Troelstra, *Lectures on Linear Logic* (CSLI, Stanford University, 1992).

[58] C. Walther, A classification of many-sorted unification theories, in: J. H. Siekmann, ed., *Proc. 8th. Int. Conf. on Automated Deduction, Oxford, England* (LNCS 230, Springer-Verlag, 1986) 525–537.