Rewriting Logic as a Logical and Semantic Framework*

Narciso Martí-Oliet and José Meseguer
SRI International, Menlo Park, CA 94025, and
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305

Abstract

Rewriting logic [72] 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 [70], 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 for a number of quite different logics of interest, including equational logic, Horn logic with equality, linear logic, logics with quantifiers, and any sequent calculus presentation of a logic for a very general notion of "sequent," is discussed in detail. 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*. Representation maps of this kind provide executable specifications of the corresponding object logics within rewriting logic, which can be very useful for prototyping purposes.

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 emphasized and illustrated with examples such as concurrent object-oriented programming and CCS. The relationship with structural operational semantics is discussed by means of examples. In addition, the way in which constraint solving fits within the rewriting logic framework is briefly explained. Finally, the use of rewriting logic as a logic of change that overcomes the frame problem in AI is also discussed.

*Supported by Office of Naval Research Contracts N00014-90-C-0086, N00014-92-C-0518, 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). The first author was also supported by a Postdoctoral Research Fellowship of the Spanish Ministry for Education and Science and by CICYT, TIC 95-0433-C03-01.

Contents

1 Introduction **3**
- 1.1 Rewriting logic as a logical framework . . . 3
- 1.2 Rewriting logic as a semantic framework . . . 5

2 General logics **7**
- 2.1 Syntax . . . 7
- 2.2 Entailment systems . . . 7
- 2.3 Institutions . . . 8
- 2.4 Logics . . . 9
- 2.5 Proof calculi . . . 9
- 2.6 Mapping logics . . . 10
- 2.7 The idea of a logical framework . . . 12
- 2.8 Reflection . . . 13

3 Rewriting logic **14**
- 3.1 Basic universal algebra . . . 14
- 3.2 The rules of rewriting logic . . . 14
- 3.3 The meaning of rewriting logic . . . 16
- 3.4 The Maude and MaudeLog languages . . . 17
- 3.5 The models of rewriting logic . . . 19
  - 3.5.1 Preorder, poset, and algebra models . . . 23

4 Rewriting logic as a logical framework **24**
- 4.1 Mapping equational logic . . . 24
- 4.2 Mapping Horn logic . . . 26
- 4.3 Mapping linear logic . . . 28
  - 4.3.1 Expressing linear logic in rewriting logic . . . 29
  - 4.3.2 Representing a linear logic theory in rewriting logic . . . 32
  - 4.3.3 The map of logics . . . 34
- 4.4 Quantifiers . . . 35
- 4.5 Mapping sequent systems . . . 40
- 4.6 Reflection in rewriting logic . . . 47

5 Rewriting logic as a semantic framework **50**
- 5.1 Generality of rewriting logic as a model of computation . . . 51
- 5.2 Concurrent object-oriented programming . . . 52
- 5.3 CCS . . . 54
- 5.4 Structural operational semantics . . . 60
- 5.5 Constraint solving . . . 66
- 5.6 Action and change in rewriting logic . . . 68

6 Concluding remarks **71**

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 [72] 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.

1.1 Rewriting logic as a logical framework

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 [31] on *institutions*, 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 work on institutions has been further developed by their original proponents and by others [32, 33, 108, 109], and has influenced other notions proposed by different authors [68, 94, 27, 70, 43, 97, 24, 4]. Some of the notions proposed are closely related to institutions; however, in other cases the main intent is to substantially expand the primarily model-theoretic

viewpoint provided by institutions to give an adequate treatment of proof-theoretic aspects such as entailment and proof structures. The theory of general logics [70] that we present in summary form in Section 2 is one such attempt to encompass also proof-theoretic aspects, 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 [41, 42, 28] and meta-theorem-provers such as Isabelle [91], λProlog [87, 26], and Elf [92], 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 [9] 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 [9] 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 [9], who advocate constructive type theory as the framework logic, work of Matthews, Smaill and Basin [67], who use Feferman's FS_0 [25], a logic designed with the explicit purpose of being a metalogical framework, earlier work by Smullyan [105], and work by Goguen, Stevens, Hobley and Hilberdink [36] on the 2OBJ meta-theorem-prover, which uses order-sorted equational logic [35, 37].

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" [28, 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 which is left for a subsequent paper, externalization of logics to support metalogical reasoning is also possible in rewriting logic.

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 consider—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.

The fact that rewriting logic is *reflective* [17, 18] has very important practical consequences for its use as a logical framework. Note that a representation map $\Psi : \mathcal{L} \rightarrow \mathit{RWLogic}$ for a logic $\mathcal{L}$ is by its very nature a *metatheoretic* construction above the object levels of both $\mathcal{L}$ and *RWLogic*. In particular, Ψ includes as one of its key components a function $\Psi_{\mathbf{Th}} : \mathbf{Th}_{\mathcal{L}} \rightarrow \mathbf{Th}_{\mathit{RWLogic}}$ translating theories in $\mathcal{L}$ into rewrite theories. However, thanks to the fact that the finitely presentable rewrite theories can be reified as an abstract data type `RWL-ADT`, for $\mathcal{L}$ a logic having a finitary presentation of its syntax and its deduction rules, and such that Ψ maps finitely presented theories in $\mathcal{L}$ to finitely presented rewrite theories, we can often *reify* a metatheoretic construction such as Ψ inside rewriting logic by first defining an abstract data type $\mathcal{L}$`-ADT` representing the finitely presentable theories of $\mathcal{L}$, and then reifying Ψ itself as an equationally defined function $\overline{\Psi} : \mathcal{L} - \mathtt{ADT} \longrightarrow \mathtt{RWL} - \mathtt{ADT}$. In this way, the translation Ψ becomes itself expressible and executable inside rewriting logic.

1.2 Rewriting logic as a semantic framework

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. A good case in point is the isomorphism between the Petri net model of concurrent computation [96] and the tensor fragment of linear logic [29] (see [62] and references therein). 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 [72]. Since the generality and naturalness with which many concurrency models can be expressed in rewriting logic has already been illustrated at length in [72], only a brief summary is given in this paper. However, the CCS [84] and the concurrent object-oriented programming models are discussed in some detail to provide relevant examples.

Concurrent object-oriented programming is of particular interest. Given that the semantics of object-oriented programs is still poorly understood, and that the semantics of concurrent object-oriented systems is even less well understood, the ease with which rewriting logic can be used to give a precise semantics to concurrent object-oriented programs and to make such programs declarative is quite encouraging. In this paper, only the basic ideas of such a semantics are sketched; a much more detailed account can be found in [74].

The similarities between rewriting logic and structural operational semantics [93, 54] already noted in [72] are further explored in this paper. We give examples showing that different styles of structural operational semantics can be regarded as special cases of rewriting logic. The two main differences are the greater expressive power of rewriting logic due to the ability for rewriting modulo user-definable axioms, and the fact that rewriting logic is a full-fledged logic with both a proof and a model theory, whereas structural operational semantics accounts are only proof-theoretic.

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 frame problem in artificial intelligence is caused by the need, typical of classical logic representations, to specify changes of state by stating not only what changes, but

also what does not change. This is basically due to the essentially Platonic character of classical logic. Since rewriting logic is by design a logic of change that allows sound and complete deductions about the transitions of a system whose basic changes are axiomatized by rewrite rules, the difficulties associated with the frame problem disappear [63]. In addition, the conservative mappings of Horn logic with equality and of linear logic studied in Sections 4.2 and 4.3, respectively, directly show how other logics of change recently proposed [48, 38, 39, 65, 66] can be subsumed as special cases. Added benefits include the straightforward support for concurrent change and the logical support for object-oriented representation.

The paper begins with a summary of the theory of general logics [70] that provides the conceptual basis for our discussion of logical frameworks. Then the rules of deduction and the model theory of rewriting logic are introduced, and the Maude and MaudeLog languages based on rewriting logic are briefly discussed. This is followed by a section presenting examples of logics representable in the rewriting logic framework. The role of rewriting logic as a semantic framework is then discussed and illustrated with examples. The paper ends with some concluding remarks.

2 General logics

A 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. A flexible axiomatic notion of a *proof calculus*, in which proofs of entailments, not just the entailments themselves, are first class citizens should also be included. This section gives a brief review of the required notions and axioms that will be later used in our treatment of rewriting logic as a logical framework; a detailed account with many examples can be found in [70].

2.1 Syntax

Syntax can typically be given by a *signature* Σ providing a grammar on which to build *sentences*. For first order logic, a typical signature consists of a list of function symbols and a list of predicate symbols, each with a prescribed number of arguments, which are used to build up sentences by means of the usual logical connectives. For our purposes, it is enough to 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.

2.2 Entailment systems

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. We make this relation relative to a signature.

In what follows, $|\mathcal{C}|$ denotes the collection of objects of a category $\mathcal{C}$.

Definition 1 [70] An *entailment system* is a triple $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ such that

- **Sign** is a category whose objects are called *signatures*,
- $sen : \mathbf{Sign} \longrightarrow \mathbf{Set}$ is a functor associating to each signature Σ a corresponding set of Σ-*sentences*, and
- $\vdash$ is a function associating to each $\Sigma \in |\mathbf{Sign}|$ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$ called Σ-*entailment* such that the following properties are satisfied:
  1. *reflexivity*: for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
  2. *monotonicity*: if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash_\Sigma \varphi$,
  3. *transitivity*: if $\Gamma \vdash_\Sigma \varphi_i$, for all $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$,
  4. $\vdash$-*translation*: if $\Gamma \vdash_\Sigma \varphi$, then for any $H : \Sigma \to \Sigma'$ in **Sign**, $sen(H)(\Gamma) \vdash_{\Sigma'} sen(H)(\varphi)$.

Except for the explicit treatment of syntax translations, the axioms are very similar to Scott's axioms for a consequence relation [100].

Definition 2 [70] Given an entailment system $\mathcal{E}$, its category **Th** of *theories* has as objects pairs $T = (\Sigma, \Gamma)$ with Σ a signature and $\Gamma \subseteq sen(\Sigma)$. A *theory morphism* $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is a signature morphism $H : \Sigma \to \Sigma'$ such that if $\varphi \in \Gamma$, then $\Gamma' \vdash_{\Sigma'} sen(H)(\varphi)$.

A theory morphism $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ is called *axiom-preserving* if it satisfies the condition that $sen(H)(\Gamma) \subseteq \Gamma'$. This defines a subcategory $\mathbf{Th}_0$ with the same objects as **Th** but with morphisms restricted to be axiom-preserving theory morphisms. Notice that the category $\mathbf{Th}_0$ does not depend at all on the entailment relation $\vdash$.

2.3 Institutions

The axiomatization of a model theory is due to the seminal work of Goguen and Burstall on *institutions* [31, 33].

Definition 3 [31] An *institution* is a 4-tuple $\mathcal{I} = (\mathbf{Sign}, sen, \mathbf{Mod}, \models)$ such that

- **Sign** is a category whose objects are called *signatures*,
- $sen : \mathbf{Sign} \longrightarrow \mathbf{Set}$ is a functor associating to each signature Σ a set of Σ-*sentences*,
- $\mathbf{Mod} : \mathbf{Sign} \longrightarrow \mathbf{Cat}^{op}$ is a functor that gives for each signature Σ a category whose objects are called Σ-*models*, and
- $\models$ is a function associating to each $\Sigma \in |\mathbf{Sign}|$ a binary relation $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times sen(\Sigma)$ called Σ-*satisfaction* satisfying the following *satisfaction condition* for each $H : \Sigma \to \Sigma'$ in **Sign**: for all $M' \in |\mathbf{Mod}(\Sigma')|$ and all $\varphi \in sen(\Sigma)$,

$$M' \models_{\Sigma'} sen(H)(\varphi) \quad \Longleftrightarrow \quad \mathbf{Mod}(H)(M') \models_\Sigma \varphi.$$

The satisfaction condition just requires that, for any syntax translation between two signatures, a model of the second signature satisfies a translated sentence if and only if the translation of this model satisfies the original sentence. Note that **Mod** is a contravariant functor, that is, translations of models go backwards.

Given a set of Σ-sentences Γ, we define the category $\mathbf{Mod}(\Sigma, \Gamma)$ as the full subcategory of $\mathbf{Mod}(\Sigma)$ determined by those models $M \in |\mathbf{Mod}(\Sigma)|$ that satisfy all the sentences in Γ, i.e., $M \models_\Sigma \varphi$ for each $\varphi \in \Gamma$.

Since the definition above of the category of theories $\mathbf{Th}_0$ only depends on signatures and sentences, it also makes sense for an institution.

2.4 Logics

Defining a *logic* is now almost trivial.

Definition 4 [70] A *logic* is 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 $\varphi \in sen(\Sigma)$,

$$\Gamma \vdash_\Sigma \varphi \implies \Gamma \models_\Sigma \varphi,$$

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

The logic is called *complete* if the above implication is in fact an equivalence.

2.5 Proof calculi

A given logic may admit many different proof calculi. For example, in first order logic we have Hilbert style, natural deduction, and sequent calculi among others, and the way in which proofs are represented and generated by rules of deduction is different for each of these calculi. It is useful to make proofs relative to a given theory T whose axioms we are allowed to use in order to prove theorems.

A proof calculus associates to each theory T a *structure* $P(T)$ of proofs that use axioms of T as hypotheses. The structure $P(T)$ typically has an *algebraic structure* of some kind so that we can obtain new proofs out of previously given proofs by operations that mirror the rules of deduction of the calculus in question. We need not make a choice about the particular types of algebraic structures that should be allowed for different proof calculi; we can abstract from such choices by simply saying that for a given proof calculus there is a category **Str** of such structures and a functor $P : \mathbf{Th}_0 \longrightarrow \mathbf{Str}$ assigning to each theory T its structure of proofs $P(T)$. Of course, it should be possible to extract from $P(T)$ the underlying set $proofs(T)$ of all the proofs of theorems of the theory T, and this extraction should be functorial. Also, each proof, whatever it is, should contain information about what theorem it is a proof of; this can be formalized by postulating a "projection function" π_T (parameterized by T in a natural way) that maps each proof

$p \in \mathit{proofs}(T)$ to the sentence φ that it proves. Of course, each theorem of T must have at least one proof, and sentences that are not theorems should have no proof. To summarize, a *proof calculus* [70] consists of an entailment system together with:

- A functorial assignment P of a structure $P(T)$ to each theory T.
- An additional functorial assignment of a set $\mathit{proofs}(T)$ to each structure $P(T)$.
- A natural function π_T assigning a sentence to each proof $p \in \mathit{proofs}(T)$ and such that, for Γ the axioms of T, a sentence φ is in the image of π_T if and only if $\Gamma \vdash \varphi$.

It is quite common to encounter proof systems of a specialized nature. In these calculi, only certain signatures are admissible as syntax—e.g., finite signatures—, only certain sentences are allowed as axioms, and only certain sentences—possibly different from the axioms—are allowed as conclusions. The obvious reason for introducing such specialized calculi is that proofs are simpler under the given restrictions. In computer science the choice between an efficient and an inefficient calculus may have dramatic practical consequences. For logic programming languages, such calculi do (or should) coincide with what is called their *operational semantics*, and mark the difference between a hopelessly inefficient theorem-prover and an efficient programming language. In practice, of course, we are primarily interested in proof calculi and proof subcalculi that are computationally effective. This is axiomatized by the notion of an *(effective) proof subcalculus* which can be found in [70].

2.6 Mapping logics

The advantage of having an axiomatic theory of logics is that the "space" of all logics (or that of all entailment systems, institutions, proof calculi, etc.) becomes well understood. This space is not just a collection of objects bearing no relationship to each other. In fact, the most interesting fruit of the theory of general logics outlined in this section 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 simplest kind of relation is a *sublogic* (subentailment system, etc.) relation. Thus, first order equational logic and Horn logic are both sublogics of first order logic with equality. However, more subtle and general ways of relating logics are possible. For example, we may want to represent the universal fragment of first order logic in a purely functional way by taking all the predicates and formulas to be *functions* whose value is either *true* or *false* so that a universal formula then becomes an equation equating a given term to *true*. The general way of relating logics (entailment systems, etc.) is to consider *maps* that interpret one logic into another. A detailed treatment of such maps is given in [70]; here we summarize some of the key ideas.

Let us first discuss in some detail *maps of entailment systems*. Basically, a map of entailment systems $\mathcal{E} \longrightarrow \mathcal{E}'$ 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 to signatures of $\mathcal{E}'$, and that sentences of $\mathcal{E}$ are mapped to sentences of $\mathcal{E}'$ in a way that is coherent with the mapping of their corresponding signatures. In addition, such a mapping α must respect the entailment relations $\vdash$ of $\mathcal{E}$ and $\vdash'$ of $\mathcal{E}'$, i.e., we must have $\Gamma \vdash \varphi \;\Rightarrow\; \alpha(\Gamma) \vdash' \alpha(\varphi)$. It

turns out that for many interesting applications, including the functional representation of first order logic sketched above, one wants to be more general and allow maps that send a signature of $\mathcal{E}$ to a *theory* of $\mathcal{E}'$. These maps extend to maps between theories, and in this context the coherence with the mapping at the level of signatures is expressed by the notion of *sensible functor* defined in [70].

Definition 5 [70] Given entailment systems $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ and $\mathcal{E}' = (\mathbf{Sign}', sen', \vdash')$, a *map of entailment systems* $(\Phi, \alpha) : \mathcal{E} \longrightarrow \mathcal{E}'$ consists of a natural transformation $\alpha : sen \Rightarrow \Phi; sen'$ and an α-sensible functor[1] $\Phi : \mathbf{Th}_0 \longrightarrow \mathbf{Th}'_0$ satisfying the following property:

$$\Gamma \vdash_\Sigma \varphi \implies \Gamma' \cup \alpha_\Sigma(\Gamma) \vdash'_{\Sigma'} \alpha_\Sigma(\varphi),$$

where, by convention, $(\Sigma', \Gamma') = \Phi(\Sigma, \Gamma)$.

We call (Φ, α) *conservative* when the above implication is an equivalence.

The property of being conservative may be essential for many applications. For example, since proof calculi are in a sense computational engines on which the design and implementation of theorem-provers and logic programming languages can be based, we can view the establishment of a map of proof calculi having nice properties, such as conservativity, as a proof of correctness for a *compiler* that permits implementing a system based on the first calculus in terms of another system based on the second. Besides establishing correctness, the map itself specifies the compilation function.

A *map of institutions*[2] $\mathcal{I} \longrightarrow \mathcal{I}'$ 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 $M' \models' \alpha(\varphi) \iff \beta(M') \models \varphi$.

Definition 6 [70] Given institutions $\mathcal{I} = (\mathbf{Sign}, sen, \mathbf{Mod}, \models)$ and $\mathcal{I}' = (\mathbf{Sign}', sen', \mathbf{Mod}', \models')$, a *map of institutions* $(\Phi, \alpha, \beta) : \mathcal{I} \longrightarrow \mathcal{I}'$ consists of a natural transformation $\alpha : sen \Rightarrow \Phi; sen'$, an α-sensible functor $\Phi : \mathbf{Th}_0 \longrightarrow \mathbf{Th}'_0$, and a natural transformation $\beta : \Phi^{op}; \mathbf{Mod}' \Rightarrow \mathbf{Mod}$ such that for each $\Sigma \in |\mathbf{Sign}|$, $\varphi \in sen(\Sigma)$, and $M' \in |\mathbf{Mod}'(\Phi(\Sigma, \emptyset))|$ the following property is satisfied:

$$M' \models'_{\Sigma'} \alpha_\Sigma(\varphi) \iff \beta_{(\Sigma,\emptyset)}(M') \models_\Sigma \varphi,$$

where Σ' is the signature of the theory $\Phi(\Sigma, \emptyset)$.

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.

[1] We refer to [70] for the detailed definition of α-sensible functor. Basically, what is required is that the provable consequences of the theory $\Phi(\Sigma, \Gamma)$ are entirely determined by $\Phi(\Sigma, \emptyset)$ and by $\alpha(\Gamma)$.

[2] Such maps are different from the "institution morphisms" considered by Goguen and Burstall in [31, 33].

Definition 7 [70] Given logics $\mathcal{L} = (\mathbf{Sign}, sen, \mathbf{Mod}, \vdash, \models)$ and $\mathcal{L}' = (\mathbf{Sign}', sen', \mathbf{Mod}', \vdash', \models')$, a *map of logics* $(\Phi, \alpha, \beta) : \mathcal{L} \longrightarrow \mathcal{L}'$ consists of a functor $\Phi : \mathbf{Th}_0 \longrightarrow \mathbf{Th}'_0$, and natural transformations $\alpha : sen \Rightarrow \Phi; sen'$ and $\beta : \Phi^{op}; \mathbf{Mod}' \Rightarrow \mathbf{Mod}$ such that:

- $(\Phi, \alpha) : (\mathbf{Sign}, sen, \vdash) \longrightarrow (\mathbf{Sign}', sen', \vdash')$ is a map of entailment systems, and
- $(\Phi, \alpha, \beta) : (\mathbf{Sign}, sen, \mathbf{Mod}, \models) \longrightarrow (\mathbf{Sign}', sen', \mathbf{Mod}', \models')$ is a map of institutions.

We call (Φ, α, β) *conservative* if and only if (Φ, α) is so as a map of entailment systems.

There is also a notion of map of proof calculi, for which we refer the reader to [70].

2.7 The idea of a logical framework

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 (see Section 4). 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 [15].

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. We conjecture that the scope of rewriting logic contains all entailment systems of "practical interest" for a reasonable axiomatization of such systems.

2.8 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 [17]. These notions axiomatize reflective logics within the theory of general logics [70]. 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 [17].

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"

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

3 Rewriting logic

This section gives the rules of deduction and semantics of rewriting logic and explains its computational meaning. The Maude and MaudeLog languages, based on rewriting logic, are also briefly discussed.

3.1 Basic universal algebra

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 [13, 77]. For the sake of simplifying the exposition, we treat here the *unsorted* case.

A set Σ of function symbols is a ranked alphabet $\Sigma = \{\Sigma_n \mid n \in \mathbb{N}\}$. A Σ-algebra is then a set A together with an assignment of a function $f_A : A^n \longrightarrow A$ for each $f \in \Sigma_n$ with $n \in \mathbb{N}$. We denote by T_Σ the Σ-algebra of ground Σ-terms, and by $T_\Sigma(X)$ the Σ-algebra of Σ-terms with variables in a set X. Similarly, given a set E of Σ-equations, $T_{\Sigma,E}$ denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo the equations E; in the same way, $T_{\Sigma,E}(X)$ denotes the Σ-algebra of equivalence classes of Σ-terms with variables in X modulo the equations E. Let $[t]_E$ or just $[t]$ denote the E-equivalence class of t.

Given a term $t \in T_\Sigma(\{x_1, \ldots, x_n\})$, and terms $u_1, \ldots, u_n$, $t(u_1/x_1, \ldots, u_n/x_n)$ denotes the term obtained from t by *simultaneously substituting* u_i for x_i, $i = 1, \ldots, n$. To simplify notation, we denote a sequence of objects $a_1, \ldots, a_n$ by $\overline{a}$; with this notation, $t(u_1/x_1, \ldots, u_n/x_n)$ can be abbreviated to $t(\overline{u}/\overline{x})$.

3.2 The rules of rewriting logic

A *signature* in rewriting logic is a pair (Σ, E) with Σ a ranked alphabet of function symbols and E a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo the set of equations 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 E of equations is empty. Techniques for rewriting modulo equations have been studied extensively [23] and can be used to implement rewriting modulo many equational theories of interest.

Given a signature (Σ, E), *sentences* of rewriting logic are "sequents" of the form $[t]_E \longrightarrow [t']_E$, where t and t' are Σ-terms possibly involving some variables from the countably infinite set $X = \{x_1, \ldots, x_n, \ldots\}$. A *theory* in this logic, called a rewrite theory, is a slight generalization of the usual notion of theory as in Definition 2 in that, in addition, we allow the axioms—in this case the sequents $[t]_E \longrightarrow [t']_E$—to be labelled. This is very natural for many applications, and customary for automata—viewed as labelled transition systems—and for Petri nets, which are both particular instances of our definition.

Definition 8 A *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and R is a set of pairs $R \subseteq L \times T_{\Sigma,E}(X)^2$ whose first component is a label and whose second component is a pair of E-equivalence classes of terms, with $X = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables. Elements of R are called *rewrite rules*[3]. We understand a rule $(r, ([t], [t']))$ as a labelled sequent and use for it the notation $r : [t] \longrightarrow [t']$. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t or t', we write $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, or in abbreviated notation $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$.

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sequent $[t] \longrightarrow [t']$ 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*:

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

$$\frac{}{[t] \longrightarrow [t]}.$$

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

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

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

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

4. **Transitivity**.

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

Equational logic (modulo a set of axioms E) is obtained from rewriting logic by adding the following rule:

[3]To simplify the exposition the rules of the logic are given for the case of *unconditional* rewrite rules. However, all the ideas presented here have been extended to conditional rules in [72] 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 of course increases considerably the expressive power of rewrite theories, as illustrated by several of the examples presented in this paper.

5. **Symmetry**.

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

With this new rule, sequents derivable in equational logic are *bidirectional*; therefore, in this case we can adopt the notation $[t] \leftrightarrow [t']$ throughout and call such bidirectional sequents *equations*.

A nice consequence of having defined rewriting logic is that concurrent rewriting, rather than emerging as an operational notion, actually coincides with deduction in such a logic.

Definition 9 Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, a (Σ, E)-sequent $[t] \longrightarrow [t']$ is called a *concurrent $\mathcal{R}$-rewrite* (or just a *rewrite*) if and only if it can be derived from $\mathcal{R}$ by finite application of the rules 1-4, i.e., $\mathcal{R} \vdash [t] \longrightarrow [t']$.

3.3 The meaning of rewriting logic

A logic worth its salt should be understood as a method of correct reasoning about some class of entities, not as an empty formal game. For equational logic, the entities in question are sets, functions between them, and the relation of identity between elements. For rewriting logic, the entities in question are *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. Clearly, concurrent systems should be the *models* giving a semantic interpretation to rewriting logic, in the same way that algebras are the models giving a semantic interpretation to equational logic. A precise account of the model theory of rewriting logic, giving rise to an initial model semantics for Maude modules and fully consistent with the above system-oriented interpretation, is sketched in Section 3.5 and developed in full detail for the more general conditional case in [72].

Therefore, in rewriting logic a sequent $[t] \longrightarrow [t']$ should not be read as "$[t]$ *equals* $[t']$," but as "$[t]$ *becomes* $[t']$." Clearly, rewriting logic is a logic of *becoming* or *change*, not a logic of equality in a static sense. The apparently innocent step of adding the symmetry rule is in fact a *very strong* restriction, namely assuming that *all change is reversible*, thus bringing us into a timeless Platonic realm in which "before" and "after" have been identified.

A related observation, which is particularly important for the use of rewriting logic as a logical framework, is that $[t]$ 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. For the case of concurrent systems, this generality is discussed at length in [72] (see also [63] for the advantages of this generality in the context of unifying AI logics of action). In a similar vein, but with a broader focus, Section 5 discusses the advantages of rewriting logic as a general semantic framework in which to specify and prototype languages and systems. Finally, Section 4 explores the generality of rewriting logic as a logical framework in which logics can be represented and prototyped.

In summary, the rules of rewriting logic are rules to reason about *change in a concurrent system*, or, alternatively, metarules for reasoning about *deduction in a logical system*. They allow us to draw valid conclusions about the evolution of the system from certain basic types of change known to be possible, or, in the alternative viewpoint, about the correct deductions possible in a logical system. Our present discussion is summarized as follows:

State	$\leftrightarrow$	*Term*	$\leftrightarrow$	*Proposition*
Transition	$\leftrightarrow$	*Rewriting*	$\leftrightarrow$	*Deduction*
Distributed Structure	$\leftrightarrow$	*Algebraic Structure*	$\leftrightarrow$	*Propositional Structure*

Section 4 will further clarify and illustrate each of the correspondences in the last two columns of the diagram, and Section 5 will do the same for the first two columns.

3.4 The Maude and MaudeLog languages

Rewriting logic can be used directly as a wide spectrum language supporting specification, rapid prototyping, and programming of concurrent systems. As explained later in this paper, rewriting logic can also be used as a logical framework in which other logics can be naturally represented, and as a semantic framework for specifying languages and systems. The Maude language [74, 16] supports all these uses of rewriting logic in a particularly modular way in which modules are rewrite theories and in which functional modules with equationally defined data types can also be declared in a functional sublanguage. The examples given later in this paper illustrate the syntax of Maude. Details about the language design, its semantics, its parallel programming and wide spectrum capabilities, and its support of object-oriented programming can be found in [72, 81, 74, 75]. Here we provide a very brief sketch that should be sufficient for understanding the examples presented later.

In Maude there are three kinds of *modules*:

1. *Functional modules*, introduced by the keyword `fmod`,
2. *System modules*, introduced by the keyword `mod`, and
3. *Object-oriented modules*, introduced by the keyword `omod`.

Object-oriented modules can be reduced to a special case of system modules for which a special syntax is used; therefore, in essence we only have functional and system modules. Maude's functional and system modules are respectively of the form

- `fmod` $\mathcal{E}$ `endfm`, and

- `mod` $\mathcal{R}$ `endm`,

for $\mathcal{E}$ an equational theory and $\mathcal{R}$ a rewrite theory[4]. In functional modules, equations are declared with the keywords `eq` or `ceq` (for conditional equations), and in system or object-oriented modules with the keywords `ax` or `cax`. In addition, certain equations, such as any combination of associativity, commutativity, or identity, for which rewriting modulo is provided, can be declared together with the corresponding operator using the keywords `assoc`, `comm`, `id`. Rules can only appear in system or object-oriented modules, and are declared with the keywords `rl` or `crl`.

In Maude a module can have *submodules*, which can be imported with `protecting`, `extending`, and `using` qualifications stating (in decreasing order) the degree of integrity enjoyed by the submodule when imported by the supermodule.

The version of rewriting logic used for Maude in this paper is *order-sorted*[5]. 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 [35] and they form a sublanguage similar to OBJ [37].

Like OBJ, Maude has also *theories* to specify semantic requirements for interfaces and to make high level assertions about modules. They are of the three kinds:

1. *Functional theories*, introduced by the keyword `fth`,

2. *System theories*, introduced by the keyword `th`, and

3. *Object-oriented theories*, introduced by the keyword `oth`.

Also as OBJ, Maude has *parameterized modules* and *theories*, again of the three kinds, and *views* that are theory interpretations relating theories to modules or to other theories.

Maude can be further extended to a language called MaudeLog that unifies the paradigms of functional programming, Horn logic programming, and concurrent object-oriented programming. In fact, Maude's design is based on a general axiomatic notion of "logic programming language" based on the general axiomatic theory of logic sketched in Section 2 [70, 73]. Technically, a unification of paradigms is achieved by mapping the logics of each paradigm into a richer logic in which the paradigms are unified. In the case of Maude and MaudeLog, what is done is to define a new logic (rewriting logic) in which concurrent computations, and in particular concurrent object-oriented computations, can be expressed in a natural way, and then to formally relate this logic to the logics of the functional and relational paradigms, i.e., to equational logic and to Horn logic, by means of maps of logics that provide a simple and rigorous unification of paradigms. As it has already been mentioned, we actually assume an order-sorted structure throughout, and therefore the logics in question are: order-sorted rewriting logic, denoted *OSRWLogic*, order-sorted equational logic, denoted *OSEqtl*, and order-sorted Horn logic, denoted *OSHorn*.

[4] This is somewhat inaccurate in the case of system modules having functional submodules because we have to "remember" that the submodule is functional.

[5] The latest version of Maude [16] 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 [13, 77].

The logic of equational programming can be embedded within (order-sorted) rewriting logic by means of a map of logics

$$OSEqtl \longrightarrow OSRWLogic.$$

The details of this map of logics are discussed in Section 4.1. At the programming language level, such a map corresponds to the inclusion of Maude's functional modules (essentially identical to OBJ modules) within the language.

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

$$OSHorn \longrightarrow OSRWLogic$$

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

The difference between Maude and MaudeLog does not consist of any change in the underlying logic; indeed, both languages are based on rewriting logic, and both have rewrite theories as programs. It resides, rather, in an enlargement of the set of *queries* that can be presented, so that, while keeping the same syntax and models, in MaudeLog we also consider queries involving existential formulas of the form

$$\exists \overline{x} \quad [u_1(\overline{x})] \longrightarrow [v_1(\overline{x})] \ \wedge \ \ldots \ \wedge \ [u_k(\overline{x})] \longrightarrow [v_k(\overline{x})].$$

Therefore, the sentences and the deductive rules and mechanisms that are now needed require further extensions of rewriting logic deduction. In particular, solving such existential queries requires performing *unification*, specifically, given Maude's typing structure, order-sorted E-unification for a set E of structural axioms [80].

3.5 The models of rewriting logic

We first sketch the construction of initial and free models for a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$. Such models capture nicely the intuitive idea of a "rewrite system" in the sense that they are systems whose states are E-equivalence classes of terms, and whose transitions are concurrent rewrites using the rules in R. By adopting a logical instead of a computational perspective, we can alternatively view such models as "logical systems" in which formulas are validly rewritten to other formulas by concurrent rewrites which correspond to proofs for the logic in question. Such models 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.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, the model that we are seeking is a category $\mathcal{T}_{\mathcal{R}}(X)$ whose objects are equivalence classes of terms $[t] \in T_{\Sigma,E}(X)$ and whose morphisms

are equivalence classes of "proof terms" representing proofs in rewriting deduction, i.e., concurrent $\mathcal{R}$-rewrites. The rules for generating such proof terms, with the specification of their respective domain and codomain, are given below; they just "decorate" with proof terms the rules 1-4 of rewriting logic. Note that we always use "diagrammatic" notation for morphism composition, i.e., $\alpha;\beta$ always means the composition of α *followed by* β.

1. **Identities**. For each $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] : [t] \longrightarrow [t]}.$$

2. **Σ-structure**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{\alpha_1 : [t_1] \longrightarrow [t'_1] \quad \ldots \quad \alpha_n : [t_n] \longrightarrow [t'_n]}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}.$$

3. **Replacement**. For each rewrite rule $r : [t(\overline{x}^n)] \longrightarrow [t'(\overline{x}^n)]$ in R,

$$\frac{\alpha_1 : [w_1] \longrightarrow [w'_1] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w'_n]}{r(\alpha_1, \ldots, \alpha_n) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}.$$

4. **Composition**.

$$\frac{\alpha : [t_1] \longrightarrow [t_2] \qquad \beta : [t_2] \longrightarrow [t_3]}{\alpha;\beta : [t_1] \longrightarrow [t_3]}.$$

Convention. In the case when the same label r appears in two different rules of R, the "proof terms" $r(\overline{\alpha})$ can sometimes be ambiguous. We assume that such ambiguity problems have been resolved by disambiguating the label r in the proof terms $r(\overline{\alpha})$ if necessary; with this understanding, we adopt the simpler notation $r(\overline{\alpha})$ to ease the exposition.

Each of the above rules of generation defines a different operation taking certain proof terms as arguments and returning a resulting proof term. In other words, proof terms form an algebraic structure $\mathcal{P}_{\mathcal{R}}(X)$ consisting of a graph with nodes $T_{\Sigma,E}(X)$, with identity arrows, and with operations f (for each $f \in \Sigma$), r (for each rewrite rule), and $_;_$ (for composing arrows). Our desired model $\mathcal{T}_{\mathcal{R}}(X)$ is the quotient of $\mathcal{P}_{\mathcal{R}}(X)$ modulo the following equations[6]:

1. **Category**.

   (a) *Associativity.* For all α, β, γ,

$$(\alpha;\beta);\gamma = \alpha;(\beta;\gamma).$$

   (b) *Identities.* For each $\alpha : [t] \longrightarrow [t']$,

$$\alpha;[t'] = \alpha \quad \text{and} \quad [t];\alpha = \alpha.$$

[6] In the expressions appearing in the equations, when compositions of morphisms are involved, we always implicitly assume that the corresponding domains and codomains match.

2. **Functoriality of the Σ-algebraic structure**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

   (a) *Preservation of composition.* For all $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$,

   $$f(\alpha_1; \beta_1, \ldots, \alpha_n; \beta_n) = f(\alpha_1, \ldots, \alpha_n); f(\beta_1, \ldots, \beta_n).$$

   (b) *Preservation of identities.*

   $$f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)].$$

3. **Axioms in E.** For each axiom $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ in E, for all $\alpha_1, \ldots, \alpha_n$,

   $$t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n).$$

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

   $$\frac{\alpha_1 : [w_1] \longrightarrow [w_1'] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w_n']}{r(\overline{\alpha}) = r(\overline{[w]}); t'(\overline{\alpha}) = t(\overline{\alpha}); r(\overline{[w']})}.$$

Note that the set X of variables is actually a parameter of these constructions, and we need not assume X to be fixed and countable. In particular, for $X = \emptyset$, we adopt the notation $\mathcal{T}_{\mathcal{R}}$. The equations in 1 make $\mathcal{T}_{\mathcal{R}}(X)$ a category, the equations in 2 make each $f \in \Sigma$ a functor, and 3 forces the axioms E. The exchange law states that any rewrite of the form $r(\overline{\alpha})$—which represents the *simultaneous* rewriting of the term at the top using rule r *and* "below," i.e., in the subterms matched by the variables, using the rewrites $\overline{\alpha}$—is equivalent to the sequential composition $r(\overline{[w]}); t'(\overline{\alpha})$, corresponding to first rewriting on top with r and then below on the subterms matched by the variables with $\overline{\alpha}$, and is also equivalent to the sequential composition $t(\overline{\alpha}); r(\overline{[w']})$ corresponding to first rewriting below with $\overline{\alpha}$ and then on top with r. Therefore, the exchange law states that rewriting at the top by means of rule r and rewriting "below" using $\overline{\alpha}$ are processes that are independent of each other and can be done either simultaneously or in any order. Since $[t(x_1, \ldots, x_n)]$ and $[t'(x_1, \ldots, x_n)]$ can be regarded as functors $\mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$, from the mathematical point of view the exchange law just asserts that r is a *natural transformation*, i.e.,

Lemma 10 [72] For each rule $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R, the family of morphisms

$$\{r(\overline{[w]}) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w}/\overline{x})] \mid \overline{[w]} \in T_{\Sigma,E}(X)^n\}$$

is a natural transformation $r : [t(x_1, \ldots, x_n)] \Rightarrow [t'(x_1, \ldots, x_n)]$ between the functors

$$[t(x_1, \ldots, x_n)], [t'(x_1, \ldots, x_n)] : \mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X).$$

The exchange law provides a way of *abstracting* a rewriting computation by considering immaterial the order in which rewrites are performed "above" and "below" in the term; further abstraction among proof terms is obtained from the functoriality equations. The equations 1-4 provide in a sense the *most abstract* "true concurrency" view of the computations of the rewrite theory $\mathcal{R}$ that can reasonably be given.

The category $\mathcal{T}_{\mathcal{R}}(X)$ is just one among many *models* that can be assigned to the rewrite theory $\mathcal{R}$. The general notion of model, called an $\mathcal{R}$-*system*, is defined as follows:

Definition 11 Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, an $\mathcal{R}$-*system* $\mathcal{S}$ is a category $\mathcal{S}$ together with:

- a (Σ, E)-algebra structure given by a family of functors

$$\{f_{\mathcal{S}} : \mathcal{S}^n \longrightarrow \mathcal{S} \mid f \in \Sigma_n, n \in \mathbb{N}\}$$

  satisfying the equations E, i.e., for any $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ in E we have an identity of functors $t_{\mathcal{S}} = t'_{\mathcal{S}}$, where the functor $t_{\mathcal{S}}$ is defined inductively from the functors $f_{\mathcal{S}}$ in the obvious way.

- for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R a natural transformation $r_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$.

An $\mathcal{R}$-*homomorphism* $F : \mathcal{S} \longrightarrow \mathcal{S}'$ between two $\mathcal{R}$-systems is then a functor $F : \mathcal{S} \longrightarrow \mathcal{S}'$ such that it is a Σ-algebra homomorphism—i.e., $f_{\mathcal{S}} * F = F^n * f_{\mathcal{S}'}$, for each f in Σ_n, $n \in \mathbb{N}$—and such that "F preserves R," i.e., for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R we have the identity of natural transformations[7] $r_{\mathcal{S}} * F = F^n * r_{\mathcal{S}'}$, where n is the number of variables appearing in the rule. This defines a category $\mathcal{R}$-**Sys** in the obvious way.

The above definition captures formally the idea that the models of a rewrite theory are *systems*. By a "system" we mean a machine-like entity that can be in a variety of *states*, and that can change its state by performing certain *transitions*. Such transitions are transitive, and it is natural and convenient to view states as "idle" transitions that do not change the state. In other words, a system can be naturally regarded as a *category*, whose objects are the states of the system and whose morphisms are the system's transitions.

For *sequential* systems such as labelled transition systems this is in a sense the end of the story; such systems exhibit *nondeterminism*, but do not have the required algebraic structure in their states and transitions to exhibit true concurrency. Indeed, what makes a system *concurrent* is precisely the existence of an additional *algebraic structure* [72]. First, the states themselves are distributed according to such a structure; for example, for Petri nets [96] the distribution takes the form of a multiset. Second, concurrent transitions are themselves distributed according to the same algebraic structure; this is what the notion of $\mathcal{R}$-system captures, and is for example manifested in the concurrent firing of Petri nets, the evolution of concurrent object-oriented systems [74] and, more generally, in any type of concurrent rewriting.

The expressive power of rewrite theories to specify concurrent transition systems is greatly increased by the possibility of having not only transitions, but also *parameterized transitions*, i.e., *procedures*. This is what rewrite rules with variables provide. The family of states to which the procedure applies is given by those states where a component of the (distributed) state is a substitution instance of the lefthand side of the rule in question. The rewrite rule is then a *procedure* which transforms the state *locally*, by replacing such a substitution instance by the corresponding substitution instance of the righthand side. The fact that this can take place concurrently with other transitions "below" is precisely

[7] Note that we use diagrammatic order for the *horizontal*, $\alpha * \beta$, and *vertical*, $\gamma ; \delta$, composition of natural transformations [59].

what the concept of a *natural transformation* formalizes. The following table summarizes our present discussion:

System	$\longleftrightarrow$	*Category*
State	$\longleftrightarrow$	*Object*
Transition	$\longleftrightarrow$	*Morphism*
Procedure	$\longleftrightarrow$	*Natural Transformation*
Distributed Structure	$\longleftrightarrow$	*Algebraic Structure*

A detailed proof of the following theorem on the existence of initial and free $\mathcal{R}$-systems for the more general case of conditional rewrite theories is given in [72], where the soundness and completeness of rewriting logic for $\mathcal{R}$-system models is also proved.

Theorem 12 $\mathcal{T}_{\mathcal{R}}$ is an initial object in the category $\mathcal{R}$-**Sys**. More generally, $\mathcal{T}_{\mathcal{R}}(X)$ has the following universal property: Given an $\mathcal{R}$-system $\mathcal{S}$, each function $F : X \longrightarrow |\mathcal{S}|$ extends uniquely to an $\mathcal{R}$-homomorphism $F^{\natural} : \mathcal{T}_{\mathcal{R}}(X) \longrightarrow \mathcal{S}$.

3.5.1 Preorder, poset, and algebra models

Since $\mathcal{R}$-systems are an "essentially algebraic" concept[8], we can consider classes Θ of $\mathcal{R}$-systems defined by the satisfaction of additional equations. Such classes give rise to full subcategory inclusions $\Theta \hookrightarrow \mathcal{R}$-**Sys**, and by general universal algebra results about essentially algebraic theories [8] such inclusions are *reflective* [59], i.e., for each $\mathcal{R}$-system $\mathcal{S}$ there is an $\mathcal{R}$-system $R_\Theta(\mathcal{S}) \in \Theta$ and an $\mathcal{R}$-homomorphism $\rho_\Theta(\mathcal{S}) : \mathcal{S} \longrightarrow R_\Theta(\mathcal{S})$ such that for any $\mathcal{R}$-homomorphism $F : \mathcal{S} \longrightarrow \mathcal{D}$ with $\mathcal{D} \in \Theta$ there is a unique $\mathcal{R}$-homomorphism $F^\diamondsuit : R_\Theta(\mathcal{S}) \longrightarrow \mathcal{D}$ such that $F = \rho_\Theta(\mathcal{S}); F^\diamondsuit$. The assignment $\mathcal{S} \longmapsto R_\Theta(\mathcal{S})$ extends to a functor $\mathcal{R}$-**Sys** $\longrightarrow \Theta$, called the *reflection functor*.

Therefore, we can consider subcategories of $\mathcal{R}$-**Sys** that are defined by certain equations and be guaranteed that they have initial and free objects, that they are closed by subobjects and products, etc. Consider for example the following equations:

$$\forall f, g \in \mathit{Arrows},\ f = g \ \ \mathit{if} \ \ \partial_0(f) = \partial_0(g) \wedge \partial_1(f) = \partial_1(g)$$
$$\forall f, g \in \mathit{Arrows},\ f = g \ \ \mathit{if} \ \ \partial_0(f) = \partial_1(g) \wedge \partial_1(f) = \partial_0(g)$$
$$\forall f \in \mathit{Arrows},\ \partial_0(f) = \partial_1(f),$$

where $\partial_0(f)$ and $\partial_1(f)$ denote the source and target of an arrow f respectively. The first equation forces a category to be a preorder, the addition of the second requires this preorder to be a poset, and the three equations together force the poset to be *discrete*, i.e., just a set. By imposing the first one, the first two, or all three, we get full subcategories

$$\mathcal{R}\text{-}\mathbf{Alg} \subseteq \mathcal{R}\text{-}\mathbf{Pos} \subseteq \mathcal{R}\text{-}\mathbf{Preord} \subseteq \mathcal{R}\text{-}\mathbf{Sys}.$$

A routine inspection of $\mathcal{R}$-**Preord** for $\mathcal{R} = (\Sigma, E, L, R)$ reveals that its objects are preordered Σ-algebras $(A, \leq)$—i.e., preordered sets with a Σ-algebra structure such that all

[8] In the precise sense of being specifiable by an "essentially algebraic theory" or a "sketch" [8]; see [72] for more details.

the operations in Σ are monotonic—that satisfy the equations E and such that for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R and for each $\overline{a} \in A^n$ we have $t_A(\overline{a}) \leq t'_A(\overline{a})$. The poset case is entirely analogous, except that the relation $\leq$ is a partial order instead of being a preorder. Finally, $\mathcal{R}$-**Alg** is the category of ordinary Σ-algebras that satisfy the equations $E \cup eq(R)$, where $eq(r : [t] \longrightarrow [t']) = \{t_1 = t_2 \mid t_1 \in [t] \ \textit{and} \ t_2 \in [t']\}$, and $eq(R) = \bigcup\{eq(r : [t] \longrightarrow [t']) \mid [t] \longrightarrow [t'] \in R\}$.

The reflection functor associated with the inclusion $\mathcal{R}$-**Preord** $\subseteq$ $\mathcal{R}$-**Sys** sends $\mathcal{T}_{\mathcal{R}}(X)$ to the familiar $\mathcal{R}$-*rewriting relation*[9] $\rightarrow_{\mathcal{R}(X)}$ on E-equivalence classes of terms with variables in X. Similarly, the reflection associated to the inclusion $\mathcal{R}$-**Pos** $\subseteq$ $\mathcal{R}$-**Sys** maps $\mathcal{T}_{\mathcal{R}}(X)$ to the partial order $\leq_{\mathcal{R}(X)}$ obtained from the preorder $\rightarrow_{\mathcal{R}(X)}$ by identifying any two $[t], [t']$ such that $[t] \rightarrow_{\mathcal{R}(X)} [t']$ and $[t'] \rightarrow_{\mathcal{R}(X)} [t]$. Finally, the reflection functor into $\mathcal{R}$-**Alg** maps $\mathcal{T}_{\mathcal{R}}(X)$ to $T_{\mathcal{R}}(X)$, the free Σ-algebra on X satisfying the equations $E \cup eq(R)$; therefore, the classical *initial algebra semantics* of (functional) equational specifications reappears here associated with a very special class of models which—when viewed as systems—have only trivial identity transitions.

4 Rewriting logic as a logical framework

The adequacy of rewriting logic as a logical framework in which other logics can be represented by means of maps of logics or of entailment systems is explored by means of relevant examples, including equational, Horn, and linear logic, a general approach to the treatment of quantifiers, and a very general method for representing sequent presentations of a logic.

4.1 Mapping equational logic

As mentioned in Section 3.2, one can get equational logic from rewriting logic by adding the symmetry rule. Moreover, the syntax of rewriting logic includes equations in order to impose structural axioms on terms. Therefore, it should not be surprising to find out that there are many connections between both logics.

Even in the case of equational logic it can be convenient to allow sometimes a distinction between structural axioms and equations, so that an equational theory can then be described as a triple (Σ, E, Q), with Q a set of equations of the form $[u]_E = [v]_E$. This increases the expressiveness of equational theories, because we can allow more flexible description of equations—for example, omitting parentheses in the case when E contains an associativity axiom—and also supports a built-in treatment of the structural axioms in equational deduction. Indeed, this is fully consistent with the distinction made in OBJ3 and in Maude's functional modules between the equational *attributes* of an operator—such as associativity, commutativity, etc.—which are declared together with the operator, and the equations given, which are used modulo such attributes.

In order to define a map of entailment systems

$$(\Phi, \alpha) : ent(OSEqtl) \longrightarrow ent(OSRWLogic)$$

[9] It is perhaps more suggestive to call $\rightarrow_{\mathcal{R}(X)}$ the *reachability relation* of the system $\mathcal{T}_{\mathcal{R}}(X)$.

in principle we need to map an equation $[u]_E = [v]_E$ to a sequent, and the obvious choices are either $[u]_E \longrightarrow [v]_E$ or $[v]_E \longrightarrow [u]_E$. However this choice involves giving a fixed orientation to an equation, with the well-known problems that this causes. To avoid this choice, we would like to give the equation *both* orientations. We can achieve this by slightly generalizing Definition 5 of map of entailment systems in such a way that a sentence is mapped to a set of sentences[10]. In our case, α maps an equation $[u]_E = [v]_E$ to the set of sequents $\{[u]_E \longrightarrow [v]_E, [v]_E \longrightarrow [u]_E\}$, and Φ maps an equational theory $T = (\Sigma, E, Q)$ to the rewrite theory $\Phi(T) = (\Sigma, E, L, \alpha(Q))$, where $\alpha(Q) = \bigcup\{\alpha(e) \mid e \in Q\}$, and L is a labelling of the rewrite rules such that, for example, each rule is labelled by itself. This map satisfies

$$(\Sigma, E, Q) \vdash_{EL} e \iff (\Sigma, E, L, \alpha(Q)) \vdash_{RL} \alpha(e).$$

This can be easily proved by induction on the deduction rules of equational logic, using the fact that all the rules of rewriting logic are also rules of equational logic and the following lemma.

Lemma 13

$$(\Sigma, E, L, \alpha(Q)) \vdash_{RL} [u] \to [v] \iff (\Sigma, E, L, \alpha(Q)) \vdash_{RL} [v] \to [u].$$

Therefore, we have a *conservative* map of entailment systems.

Regarding the institution component, in order to extend this map to a map of logics, a simple idea is to send a $\Phi(T)$-system $\mathcal{C}$ to $R_{\mathbf{Alg}}(\mathcal{C})$, where $R_{\mathbf{Alg}}$ is the reflection functor associated with the inclusion $\Phi(T)$-**Alg** $\subseteq$ $\Phi(T)$-**Sys**, as discussed in Section 3.5.1. By definition, $R_{\mathbf{Alg}}(\mathcal{C})$ is a model of the equational theory T. However, this map does *not* satisfy the condition in Definition 6 of map of institutions. The difficulty is that, in general, from an equation $t = t'$ one can deduce that there is a chain $t \to t_1 \leftarrow t_2 \cdots t_n \leftarrow t'$, but not that $t \to t'$, as the reader familiar with term rewriting knows. To solve this problem, we consider a different quotient of the underlying (Σ, E)-algebra $|\mathcal{C}|$ in which two objects A and B are identified if and only if there exist morphisms $f : A \to B$ and $g : B \to A$ in $\mathcal{C}$. In this way, we obtain a (Σ, E)-algebra $\beta_T(\mathcal{C})$ that satisfies all the sentences in Q. Moreover, the condition in Definition 6 of map of institutions holds for this map. In short, we have obtained a conservative map of logics

$$(\Phi, \alpha, \beta) : \mathit{OSEqtl} \longrightarrow \mathit{OSRWLogic}.$$

There is also another map of logics

$$(\Phi', \alpha', \beta') : \mathit{OSEqtl} \longrightarrow \mathit{OSRWLogic}$$

that, instead of sending equations to sequents, sends equations to equations. This requires making explicit the fact, left implicit in Section 3, that equations can also be considered as sentences of rewriting logic, where, by definition,

$$(\Sigma, E, L, R) \vdash_{RL} t = t' \iff E \vdash_{EL} t = t'.$$

[10] This generalization is also very useful in relating other logics; see for example [77].

From this point of view, Φ' maps an equational theory (Σ, E) to the rewrite theory $(\Sigma, E, \emptyset, \emptyset)$, and at the level of sentences α' is just an inclusion, trivially satisfying the requirement for a map of entailment systems. Note that in this context the distinction between structural axioms and equations is not necessary.

With respect to the models, β'_T maps a $(\Sigma, E, \emptyset, \emptyset)$-system $\mathcal{C}$ to the underlying (Σ, E)-algebra structure on $|\mathcal{C}|$, trivially satisfying also the condition in Definition 6 and being therefore a map of institutions. Notice that (Φ', α', β') is *conservative* in a straightforward way.

On the opposite direction there is also a map of logics

$$(\Psi, \gamma, \delta) : \mathit{OSRWLogic} \longrightarrow \mathit{OSEqtl}$$

which maps a rewrite theory (Σ, E, L, R) to the equational theory $(\Sigma, E, \gamma(R))$ where γ removes the labels from the rules and turns the sequent signs "$\longrightarrow$" into equality signs. For the models, $\delta_{\mathcal{R}}$ is the inclusion $\mathcal{R}$-**Alg** $\subseteq$ $\mathcal{R}$-**Sys** defined in Section 3.5.1.

Notice that the composition of maps of logics $(\Phi, \alpha, \beta); (\Psi, \gamma, \delta)$ is the identity.

4.2 Mapping Horn logic

Horn logic signatures are of the form (F, P), with F a set of function symbols and P a set of predicate symbols. In the order-sorted case such symbols have ranks $f : s_1 \dots s_n \to s$, and $p : s_1 \dots s_n$, specified by strings of sorts in the poset of sorts S. Models are F-algebras M together with, for each predicate symbol $p : s_1 \dots s_n$, a subset $p_M \subseteq M_{s_1} \times \dots \times M_{s_n}$, which can alternatively be viewed as a characteristic function $p_M : M_{s_1} \times \dots \times M_{s_n} \longrightarrow \mathit{Bool}$ to the two element Boolean algebra Bool. Satisfaction of a Horn clause

$$q_1(\overline{u_1}), \dots, q_n(\overline{u_n}) \;\Rightarrow\; p(\overline{t})$$

in a model M can be expressed as either the subset containment of the intersection of the interpretations of $q_1(\overline{u_1}), \dots, q_n(\overline{u_n})$ in M inside the corresponding interpretation of $p(\overline{t})$, or, in a characteristic function description, as the functional inequality

$$q_1(\overline{u_1})_M \;\; \mathit{and} \;\; \dots \;\; \mathit{and} \;\; q_n(\overline{u_n})_M \leq p(\overline{t})_M$$

between the corresponding interpretations in M of the conjunction of the premises and of the conclusion as characteristic functions, where the inequality between the functions means inequality of their values for each of the arguments in the Boolean algebra ordering. A *homomorphism* $f : M \to M'$ between two such models is an F-homomorphism which in addition satisfies $(f_{s_1} \times \dots \times f_{s_n})(p_M) \subseteq p_{M'}$ for each $p : s_1 \dots s_n$, or in characteristic function form the functional inequality

$$p_M \leq (f_{s_1} \times \dots \times f_{s_n}); p_{M'}.$$

Horn logic is a particularly simple logic that does not use the full power of classical first-order logic and is in fact compatible with a variety of other nonclassical interpretations such as for example intuitionistic logic. It is therefore reasonable to enlarge the class

of models just described by keeping the F-algebra parts as before, but allowing instead interpretations of the predicate symbols p as "characteristic functions"

$$p_M : M_{s_1} \times \ldots \times M_{s_n} \longrightarrow M_{Prop}$$

into a partially ordered set M_{Prop} of "propositions" which is not required to be fixed, i.e., it can vary from model to model. We require of any such poset the "bare minimum" structure of having a top element $true : Prop$ and a binary associative and commutative "conjunction" operator $_,_ : Prop\ Prop \longrightarrow Prop$ that is monotonic and has $true$ as its neutral element. Of course, $Bool$ is one such poset, where conjunction is interpreted as *and*. Satisfaction of Horn clauses can be defined by a functional inequality just as before, but changing $Bool$ by the appropriate poset M_{Prop} being chosen for the model.

The natural generalization of the notion of homomorphism $f : M \longrightarrow M'$ is to again require an F-homomorphism for the operations in F, whereas for predicate symbols $p : s_1 \ldots s_n$ we require the functional inequality

$$(\dagger) \qquad p_M ; f_{Prop} \leq (f_{s_1} \times \ldots \times f_{s_n}) ; p_{M'}$$

where $f_{Prop} : M_{Prop} \longrightarrow M'_{Prop}$ is an additional component of the homomorphism, namely, a monotonic function preserving $true$ and conjunction "up to inequality" between the posets of propositions M_{Prop} and M'_{Prop} chosen for the models M and M', in the sense that we have $f_{Prop}(true_M) \leq true_{M'}$, and $f_{Prop}(x, y) \leq f_{Prop}(x), f_{Prop}(y)$, for $x, y \in M_{Prop}$. This defines a category of models (F, P)-**Mod**.

In addition, we can consider the generalization to Horn theories of the form (F, P, E, H) where E is a set of F-equations, and H is a set of Horn clauses involving the predicates in P but not equations (again, equations in E can be viewed as structural axioms forming part of the signature). A model satisfies this theory when the underlying F-algebra satisfies all the equations in E and the model satisfies the Horn clauses in H, defining in this way a full subcategory (F, P, E, H)-**Mod** of (F, P)-**Mod**. We denote by $OSHorn^{=}$ the logic whose theories are such generalized Horn theories (F, P, E, H) with equational axioms E, and whose models we have just described.

The map of logics

$$(\Phi, \alpha, \beta) : OSHorn^{=} \longrightarrow OSRWLogic$$

that we define now is a considerable simplification and extension of the map described in [73].

A Horn theory (F, P, E, H) is mapped to a rewrite theory

$$\Phi(F, P, E, H) = (F \cup P^{\diamond}, E \cup ACI, \{*\} \cup H, \{x_{Prop} \longrightarrow true\} \cup H^{\diamond}),$$

where

- $F \cup P^{\diamond}$ is the order-sorted signature that extends F by adding the additional sort $Prop$, a constant $true : Prop$, a binary operator $_,_$ on $Prop$, and, for each predicate symbol $p : s_1 \ldots s_n$ in P, an operator $p : s_1 \ldots s_n \longrightarrow Prop$;

- ACI is the set of associativity, commutativity and identity ($true$) structural axioms for the conjunction operator $_,_$;
- "$*$" is the label for the rewrite rule $x_{Prop} \longrightarrow true$, where x_{Prop} is a variable of sort $Prop$;
- $H^\diamond$ is a set of rewrite rules labelled by the Horn clauses H themselves in such a way that a Horn clause of the form $q_1(\overline{u_1}), \ldots, q_n(\overline{u_n}) \Rightarrow p(\overline{t})$ labels the rewrite rule $q_1(\overline{u_1}), \ldots, q_n(\overline{u_n}) \longrightarrow p(\overline{t})$, whereas a Horn clause of the form $p(\overline{t})$ labels the rewrite rule $true \longrightarrow p(\overline{t})$.

At the level of sentences, α maps each Horn clause to its corresponding labelled rewrite rule in the above manner.

As to models, given a Horn theory T, a $\Phi(T)$-system consists of a category $\mathcal{C}_s$ for each sort s in the poset S, and a category $\mathcal{P}$ for the sort $Prop$, together with a collection of functors satisfying the equations in $\Phi(T)$ and natural transformations interpreting the rewrite rules in $\Phi(T)$. The functor β_T sends such a system to the T-model consisting of the underlying (order-sorted) algebra structure on the family of sets $\{|\mathcal{C}_s| \mid s \in S\}$, and the poset $R_{\mathbf{Pos}}(\mathcal{P})$, where $R_{\mathbf{Pos}}$ is the reflection functor associated to the inclusion $\Phi(T)$-**Pos** $\subseteq \Phi(T)$-**Sys**, discussed in Section 3.5.1. By definition of this reflection functor, $A \leq B$ in $R_{\mathbf{Pos}}(\mathcal{P})$ if and only if there is a morphism $A \to B$ in $\mathcal{P}$. Therefore, a Horn clause $q_1(\overline{u_1}), \ldots, q_n(\overline{u_n}) \Rightarrow p(\overline{t})$ is satisfied by this T-model if and only if there is a morphism in $\mathcal{P}$ interpreting the rewrite sequent $q_1(\overline{u_1}), \ldots, q_n(\overline{u_n}) \longrightarrow p(\overline{t})$ if and only if this sequent is satisfied by the original $\Phi(T)$-system. Thus, (Φ, α, β) is indeed a map of institutions.

Notice that, by the conditions for $\mathcal{R}$-homomorphisms in Definition 11, for the homomorphisms in the image of β_T the functional inequality (†) above becomes an equality. In addition, β_T maps free $\Phi(T)$-systems to (weakly) free Horn T-models; since the entailment relation coincides with satisfaction in free models (see the proof of Theorem 3.13 in [72]), this provides a short proof of the fact that (Φ, α) is indeed a map of entailment systems, and moreover, it is *conservative*.

The same discussion applies to the case of preorders instead of posets, by considering the reflection functor associated to the inclusion $\Phi(T)$-**Preord** $\subseteq \Phi(T)$-**Sys**, which would have given a slightly more general notion of model for a Horn theory in which propositions would form a preorder.

4.3 Mapping linear logic

In this section, we describe a map of logics $LinLogic \longrightarrow OSRWLogic$ 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 [29, 110, 62] for example. We need to point out, nonetheless, the way linear logic satisfies the conditions given in Definition 1 of 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 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 [101, 62], in which sequents are interpreted as morphisms, and leads to a very natural correspondence between the models of rewriting and linear logic.

4.3.1 Expressing linear logic in rewriting 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 `PROPO[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 `PROPO[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

--- linear logic syntax
fth PROPO[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
```

Note that the equations can be used as oriented rules from left to right at the implementation level in order to obtain a canonical form for expressions in `Prop0`.

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 [29]. 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 a later section, 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 [101, 62]. This style exploits the close connection between the models of linear logic and those of rewriting logic which are also categories, as we have explained in Section 3.5. Without going into details that the reader can find for example in [62] and the references therein, the tensor and linear implication connectives are interpreted in a closed symmetric monoidal category $\langle \mathcal{C}, \otimes, \multimap \rangle$. Negation is interpreted by means of a dualizing object $\bot$ and the definition $A^\bot = A \multimap \bot$ (with this definition of negation, $\mathcal{C}$ becomes a $*$-autonomous category [7]). The categorical product & interprets additive conjunction. The interpretation of the exponential ! is given by a comonad $\langle !A, !A \to A, !A \to !!A \rangle$ that maps the comonoid structure $\top \leftarrow A \to A \& A$ into a comonoid structure $1 \leftarrow !A \to !A \otimes !A$ via isomorphisms $!\top \cong 1$ and $!(A \& A) \cong !A \otimes !A$.

The dual connectives $\parr$, $\oplus$ and ? can be defined using negation: $A \parr B = (A^\bot \otimes B^\bot)^\bot = A^\bot \multimap B$, $A \oplus B = (A^\bot \& B^\bot)^\bot$, $?A = (!A^\bot)^\bot$. Without negation, $\oplus$ needs the presence of coproducts and ? is interpreted by means of a monad with a monoid structure.

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* [20]. 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[11]. Negation is added to a weakly distributive category

[11] Cockett and Seely develop in [20] the more general case in which the tensor products are not assumed to be symmetric.

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. Cockett and Seely then prove that the concepts of weakly distributive category with negation and of $*$-autonomous category are equivalent, providing in this way a categorical semantics for linear logic in which the *par* connective $\parr$ is primitive and is not defined in terms of tensor and negation.

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 the 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 !.

```
--- linear logic rules
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 $A^{\perp} \parr B$.

4.3.2 Representing a linear logic theory in rewriting logic

A *linear theory* T in propositional linear logic consists of a finite set C of propositional constants and 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. 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, using the equations in `PROP0[X]` and the structural axioms in `LINLOG[X]`, is equal to the term

```
([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

$$\begin{aligned} T \;=\; & \{a \otimes b, !c \oplus a \vdash a, (c \oplus b)^{\perp} \\ & \quad a \parr b, ?(c^{\perp}) \vdash (?b \parr !c)^{\perp}, a \oplus b\}, \end{aligned}$$

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
```

Note that this technique can also be used to interpret quantifier-free first-order linear logic formulas, where, instead of propositional constants, we have literals built using functions and predicates. In general, we can allow any abstract data type `ADT` defining constants, functions and predicates. Then, we define the instantiation

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

which is finally extended with the corresponding rules to a theory `LINLOG(T)` corresponding to the desired theory T.

The main result is the following conservativity theorem.

Theorem 14 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]` $\otimes$ ... $\otimes$ `[An]` $\longrightarrow$ `[B1]` $\parr$ ... $\parr$ `[Bm]`

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

To show that a linear logic proof can be translated into a rewriting logic proof, the idea is similar to the proof of the soundness theorem for the categorical semantics of linear logic, where a sequent is interpreted as a morphism (see [62, Theorem 40]). What is important to realize is that the categorical constructions of these morphisms can be seen as rewriting logic proofs; for example, functoriality corresponds to the *Congruence* rule of rewriting logic, something made completely explicit in the categorical semantics of rewriting logic, as outlined in Section 3.5 and developed in detail in [72].

4.3.3 The map of logics

The fully detailed development in the previous sections provides a map of entailment systems between linear logic and rewriting logic, which is conservative because of Theorem 14. We have already discussed briefly the models of linear logic in Section 4.3.1 by way of motivation to the rules in the theory `LINLOG[X]`. Now, in order to complete the construction of the map of logics *LinLogic* $\longrightarrow$ *OSRWLogic*, we need a way of getting a (categorical) model of a linear theory T from a rewrite system that is a model of the rewrite theory `LINLOG(T)`.

The first thing to note, recalling the definition of $\mathcal{R}$-system in Section 3.5, is that for each rewrite rule in R we require just a natural transformation in the system, but we do not impose any coherence or uniqueness conditions on these natural transformations. For this reason, a `LINLOG(T)`-system interprets $A\&B$ as a weak product instead of a product, for example. A way of obtaining uniqueness would be considering the generalized rewrite theories defined in [73], but we do not need that for our purposes here. On the other hand, the attributes of the operations, like associativity or commutativity, are interpreted as identities, instead of the more general natural isomorphisms, thus satisfying all coherence conditions automatically.

In general, given a linear theory $T = (C, S)$, a `LINLOG(T)`-system consists of an algebra $\mathcal{A}$ interpreting all the structure of the functional theory `PROPO[C]`, a category $\mathcal{C}$ with all the morphisms necessary to interpret the rewrite rules in the theory `LINLOG[C]` and the rules corresponding to all the sequents in S, and an injective homomorphism $\mathcal{A} \to |\mathcal{C}|$ that, without loss of generality, we can consider to be an inclusion. Note that, as $\mathcal{A}$ is closed under all the operations in the theory `LINLOG[C]`, the full subcategory of $\mathcal{C}$ generated by $\mathcal{A}$

has the same structure as $\mathcal{C}$, and, in addition, there is a function $(_)^{\perp} : \mathcal{A} \to \mathcal{A}$ interpreting negation. Therefore, this full subcategory is *almost* a weakly distributive category with negation, products, coproducts, a comonad with a comonoid structure, and a monad with a monoid structure. What is possibly missing is the satisfaction of a set of equations between morphisms which ensure that all this structure is really what we want.

Thus, in order to get a Girard category $\mathcal{L}$ from the original `LINLOG(T)`-system, we do the quotient of the full subcategory of $\mathcal{C}$ generated by $\mathcal{A}$ by this set of equations. Clearly, there is a morphism $A \to B$ in $\mathcal{L}$ if and only if there is a morphism $A \to B$ in $\mathcal{C}$, i.e., $\mathcal{L}$ satisfies a linear sequent if and only if $\mathcal{C}$ satisfies the rewriting logic version of that sequent. This is true because the constants in C are interpreted always as the corresponding constants in $\mathcal{A}$, and variables in a sequent are also interpreted as elements of $\mathcal{A}$ (note that variables appear in a theory `ADT` that is used to instantiate `PROPO[X]`). In summary, we have a *conservative* map of logics *LinLogic* $\longrightarrow$ *OSRWLogic*.

4.4 Quantifiers

In Section 4.3 we have defined a map of logics between quantifier-free linear logic and rewriting logic. In this section, we show how to extend that map at the level of entailment systems to quantifiers. The choice of linear logic to illustrate the treatment of quantifiers is irrelevant; we could have chosen any other logic. It has only the expository advantage of building upon an example already introduced in this paper. In fact, 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 [57, 58], 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], and to the work of Talcott on binding structures [106, 107].

We begin by presenting the example of the lambda abstraction binding mechanism in the lambda calculus, as defined by Laneve and Montanari in [57] (see also [58], where this technique is generalized to combinatory reduction systems). Since in this case the syntax is much simpler, the main ideas can become more explicit and clearer to the reader.

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.

```
--- variable names
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

--- lambda calculus syntax with substitution
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:

```
--- lambda calculus rules
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 similar approach, by first introducing substitution in the syntax together with the quantifiers, and then adding rewrite rules for the new connectives. In the same way that we had to duplicate the logical connectives in both theories `PROPO[X]` and `LINLOG[X]` in Section 4.3.1 in order to have a correct treatment of negation, we also have to duplicate the operations and equations for substitution in the two modules `FO-PROPO[X]` and `FO-LINLOG[X]` below. This technicality, due to the treatment of negation, makes the exposition somewhat longer, but should not obscure the main ideas about the treatment of quantification that have been illustrated more concisely before with the lambda calculus example.

We assume an abstract data type `ADT` defining constants, functions and predicates over a set `Var` of variable names. Substitution must also be defined in this module. For example, we can have something like the following module:

```
fmod ADT[X :: VAR] is
  extending SET[X] .

  sort Term .                  *** terms
  subsort Var < Term .         *** variables are terms
  op c : -> Term .             *** constant symbol
  op f : Term Term -> Term .   *** function symbol
  sort Atom .                  *** atomic formulas
  op p : Term Term -> Atom .   *** predicate symbol

  op va : Term -> Set .        *** set of variables
  op va : Atom -> Set .        *** set of variables
  op _[_/_] : Term Term Var -> Term .  *** substitution
  op _[_/_] : Atom Term Var -> Atom .  *** substitution

  vars X Y : Var .
  vars T U V : Term .
  var P : Atom .

  *** Set of variables
  eq va(X) = {X} .
  eq va(c) = emptyset .
  eq va(f(T,V)) = va(T) U va(V) .
```

```
  eq va(p(T,V)) = va(T) U va(V) .
  eq va(V[T/X]) = (va(V) - {X}) U va(T) .
  eq va(P[T/X]) = (va(P) - {X}) U va(T) .

  *** Substitution equations
  eq X[T/X] = T .
  ceq Y[T/X] = Y  if  not(X == Y) .
  eq c[T/X] = c .
  eq f(U,V)[T/X] = f(U[T/X],V[T/X]) .
  eq p(U,V)[T/X] = p(U[T/X],V[T/X]) .
endfm

--- linear logic syntax with quantifiers
fmod FO-PROP0[X :: VAR] is
  extending PROP0[ADT[X]] .
  op _[_/_] : Prop0 Term Var -> Prop0 .   *** substitution
  op fv : Prop0 -> Set .                  *** free variables
  op ∀_._ : Var Prop0 -> Prop0 .          *** universal quantifier
  op ∃_._ : Var Prop0 -> Prop0 .          *** existential quantifier

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

  *** Negation and quantifiers
  eq (∀X.A)^⊥ = ∃X.A^⊥ .
  eq (∃X.A)^⊥ = ∀X.A^⊥ .

  *** Free variables
  eq fv(P) = va(P) .
  eq fv(1) = emptyset .
  eq ...   *** similar equations for the other logical constants
  eq fv(A^⊥) = fv(A) .
  eq fv(A ⊗ B) = fv(A) U fv(B) .
  eq ...   *** similar equations for the other logical connectives
  eq fv(∀X.A) = fv(A) - {X} .
  eq fv(∃X.A) = fv(A) - {X} .
  eq fv(A[T/X]) = (fv(A) - {X}) U va(T) .

  *** Substitution equations
  eq 1[T/X] = 1 .
  eq ...   *** similar equations for the other logical constants
  eq A^⊥[T/X] = A[T/X]^⊥ .
  eq (A ⊗ B)[T/X] = A[T/X] ⊗ B[T/X] .
```

```
  eq ...   *** similar equations for the other logical connectives
  eq (∀X.A)[T/X] = ∀X.A .
  ceq (∀Y.A)[T/X] = ∀Y.(A[T/X])
       if  not(X == Y)  and  (not(Y is-in va(T)) or not(X is-in fv(A))) .
  ceq (∀Y.A)[T/X] = ∀(new(va(T) U fv(A))).((A[new(va(T) U fv(A))/Y])[T/X])
       if  not(X == Y)  and  Y is-in fv(T)  and  X is-in fv(A) .
  eq ...   *** similar equations for the existential quantifier
endfm

mod FO-LINLOG[X :: VAR] is
  extending LINLOG[ADT[X]] .  ***
  protecting FO-PROP0[X] .    *** Note that PROP0[ADT[X]] is shared

  op _[_/_] : Prop Term Var -> Prop0 .   *** substitution
  op fv : Prop -> Set .                  *** free variables
  op ∀_._ : Var Prop -> Prop .           *** universal quantifier
  op ∃_._ : Var Prop -> Prop .           *** existential quantifier

  var P Q : Prop .
  var A : Prop0 .
  var X : Var .
  var T : Term .
  ax [∀X.A] = ∀X.[A] .
  ax [∃X.A] = ∃X.[A] .

  *** Free variables
  ax fv(1) = emptyset .
  ax ...   *** similar axioms for the other logical constants
  ax fv(P ⊗ Q) = fv(P) U fv(Q) .
  ax ...   *** similar axioms for the other logical connectives
  ax fv(∀X.P) = fv(P) - {X} .
  ax fv(∃X.P) = fv(P) - {X} .
  ax fv(P[T/X]) = (fv(P) - {X}) U va(T) .
  ax fv([A]) = fv(A) .

  *** Substitution axioms
  ax 1[T/X] = 1 .
  ax ...   *** similar axioms for the other logical constants
  ax (P ⊗ Q)[T/X] = P[T/X] ⊗ Q[T/X] .
  ax ...   *** similar axioms for the other logical connectives
  ax (∀X.P)[T/X] = ∀X.P .
  cax (∀Y.P)[T/X] = ∀Y.(P[T/X])
       if  not(X == Y)  and  (not(Y is-in va(T)) or not(X is-in fv(P))) .
  cax (∀Y.P)[Q/X] = ∀(new(va(T) U fv(P))).((P[new(va(T) U fv(P))/Y])[T/X])
       if  not(X == Y)  and  Y is-in va(T)  and  X is-in fv(P) .
```

```
  ax ...   *** similar axioms for the existential quantifier
  ax [A][T/X] = [A[T/X]] .

  *** Rules for quantifiers
  rl ∀X.P => P[T/X] .
  rl P[T/X] => ∃X.P .
  crl P => ∀X.A ⅋ Q  if  P => A ⅋ Q
                         and  not(X is-in fv(P ⊗ Q)) .
  crl P ⊗ ∃X.A => Q  if  P ⊗ A => Q
                         and  not(X is-in fv(P ⊗ Q)) .
endm
```

In this way, we have defined a map of entailment systems

$$ent(FOLinLogic) \longrightarrow ent(OSRWLogic)$$

which is also *conservative*.

4.5 Mapping sequent systems

In Section 4.3, 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 `⊢ 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. The elements in the parameter are considered singleton multisets via a subsort declaration `Elem < Mset`, and there is a multiset union operator `_,_` which is associative, commutative, and has the empty multiset `null` as neutral element. Note that what makes the elements of `Mset` multisets instead of lists is the attribute `comm` of commutativity of the union operator `_,_`.

```
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[12] 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 .
```

This displaying trick that makes possible to write a sequent calculus rule in a similar way to the usual presentation in logical textbooks is due to K. Futatsugi.

```
--- one-sided sequent calculus for linear logic
mod LL-SEQUENT[X :: VAR] is
  protecting FO-PROPO[X] .
  extending MSET[FO-PROPO[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 : PropO .
  var T : Term .
  var X : Var .
```

[12] 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. Appropriate parameterized data types can similarly be used for this purpose. For example, we use later a data type of lists to define 2-sequents in which exchange is not assumed.

```
*** Identity
rl      empty
    => --- ----
       ⊢ P,P^⊥ .

*** Cut
rl     (⊢ M,A) (⊢ N,A^⊥)
    => --- -------------
           ⊢ M,N .
*** Tensor
rl     (⊢ M,A) (⊢ B,N)
    => --- -----------
        ⊢ M,A ⊗ B,N .

*** Par
rl       ⊢ M,A,B
    => --- -------
       ⊢ M,A ⅋ B .

*** Plus
rl        ⊢ M,A
    => --- -------
       ⊢ M,A ⊕ B .

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

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

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

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

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

  *** Bottom
  rl       ⊢ M
      => --- ---
         ⊢ M,⊥ .

  *** One
  rl     empty
      => --- -
         ⊢ 1 .

  *** Top
  rl      empty
      => --- ----
         ⊢ M,⊤ .

  *** Universal
  crl      ⊢ M,A
      => --- -----
         ⊢ M,∀X.A
      if  not(X is-in fv(M)) .

  *** Existential
  rl     ⊢ M,A[T/X]
      => --- ------
         ⊢ M.∃X.A .
endm
```

Note that in the module `FO-PROPO[X]` (via the reused theory `PROPO[X]`) we have imposed associativity and commutativity attributes for some connectives, making syntax a bit more abstract than usual. However, in this case, this has no significance at all, except for the convenient fact that we only need a rule for $\oplus$ instead of two; of course, these attributes can be removed if a less abstract presentation is preferred.

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 4.3.2, 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 15 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. In general, we need an operation

```
op _⊢_ : FormList FormList -> Sequent .
```

that turns two lists of formulas (multisets, or sets in some cases) into a term representing a sequent. Then we have a sort `Configuration` representing multisets of sequents, with a union operator written using empty syntax. A sequent calculus rule

$$\frac{G_1 \vdash D_1, \ldots, G_n \vdash D_n}{G \vdash D}$$

becomes a rewrite rule

```
rl (G1 ⊢ D1) ... (Gn ⊢ Dn) => (G ⊢ D) .
```

on the sort `Configuration`, that we have displayed above also as

```
rl    (G1 ⊢ D1) ... (Gn ⊢ Dn)
   => --- --------------------
              (G ⊢ D) .
```

in order to make even clearer that the rewrite rule and the sequent notations in fact capture the same idea. In the particular case of linear logic the situation is somewhat simplified by the use of one-sided sequents. Notice also that sometimes the rewrite rule can be conditional to the satisfaction of some auxiliary side conditions like, for example, in the rule for the universal quantifier in the module above.

As another example illustrating the generality of this approach, we sketch a presentation in rewriting logic of the *2-sequent calculus* defined by A. Masini and S. Martini in order to develop a proof theory for modal logics [64, 61]. In their approach, a *2-sequent* is an expression of the form $\Gamma \vdash \Delta$, where Γ and Δ are not lists of formulas as usual, but they are lists of lists of formulas, so that sequents are endowed with a vertical structure. For example,

$$\begin{array}{cc} A, B & \\ C & \vdash \end{array} \quad \begin{array}{c} D \\ E, F \\ G \end{array}$$

is a 2-sequent, which will be represented in rewriting logic as

$$A, B; C \vdash D; E, F; G.$$

In order to define 2-sequents, we first need a parameterized module for lists, assuming a module `NAT` defining a sort `Nat` of natural numbers with zero `0`, a successor function `s_`, an addition operation `_+_`, and an order relation `_<=_`, as well as a module `BOOL` defining a sort `Bool` of truth values `true`, `false` and corresponding Boolean operations.

```
fmod LIST[X :: ELEM] is
  protecting NAT BOOL .
  sort List .
  subsort Elem < List .
  op nil : -> List .
  op _;_ : List List -> List [assoc id: nil] .
  op length : List -> Nat .
  op _in_ : Elem List -> Bool .
  vars E E' : Elem .
  vars L L' : List .
  eq length(nil) = 0 .
  eq length(E) = s0 .
  eq length(L;L') = length(L) + length(L') .
  eq E in nil = false .
  eq E in E' = if E == E' then true else false .
  eq E in (L;L') = (E in L) or (E in L') .
endfm
```

This module is instantiated twice in order to get the module of 2-sequents, using a sort of formulas `Form` whose definition is not presented here, and that should have an operation

```
  op []_ : Form -> Form .
```

corresponding to the modality $\Box$.

```
make 2-LIST is
  LIST[LIST[Form]*(op _;_ to _,_)]*(sort List to 2-List,
                                    op length to depth)
endmk
```

Note that in the `2-LIST` module the concatenation operation `_;_` is renamed to `_,_` in the case of lists of formulas, whereas in the case of lists of lists of formulas, called 2-lists, the notation `_;_` is kept. Also, to emphasize the vertical structure of 2-sequents, the operation `length` for 2-lists is renamed to `depth`.

Now we can define 2-sequents as follows:

```
fmod 2-SEQUENT is
  protecting 2-LIST .
  sort 2-Sequent .
  op _⊢_ : 2-List 2-List -> 2-Sequent .
endfm
```

The basic rules for the modality $\Box$ are

$$\frac{\begin{array}{c}\Gamma\\ \alpha\\ \beta, A\\ \Gamma'\end{array} \vdash \Delta}{\begin{array}{c}\Gamma\\ \alpha, \Box A\\ \beta\\ \Gamma'\end{array} \vdash \Delta}\ (\Box\text{-}L) \qquad \frac{\Gamma \vdash \begin{array}{c}\Delta\\ \alpha\\ A\end{array}}{\Gamma \vdash \begin{array}{c}\Delta\\ \alpha, \Box A\end{array}}\ (\Box\text{-}R)$$

where Γ, Γ', Δ denote 2-lists, α, β denote lists of formulas, and the rule $\Box$-R has the side condition that $depth(\Gamma) \leq depth(\Delta) + 1$, i.e., the formula A is the only formula in the last level of the 2-sequent.

These rules are represented in rewriting logic as follows.

```
mod 2-SEQUENT-RULES is
  protecting 2-SEQUENT .
  sort Configuration .
  subsort 2-Sequent < Configuration .
  op empty : -> Configuration .
  op __ : Configuration Configuration -> Configuration
                                        [assoc comm id: empty] .
  vars R R' S : 2-List .
  vars L L' : List .
  var A : Form .
  rl     R ; L ; L',A ; R' ⊢ S
     => --- --------------------
        R ; L,[]A ; L' ; R' ⊢ S .

  rl    R ⊢ S ; L ; A
     => --- ----------
        R ⊢ S ; L,[]A
     if depth(R) <= s(depth(S)) .
endm
```

The dual rules for the modality $\Diamond$ are treated similarly.

This general method of viewing sequents as rewrite rules can even be applied to systems more general than traditional sequent calculi. Thus, besides the possibilities of being one-sided or two-sided, one-dimensional or two-dimensional, etc., 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 5.3 and any other use of the so-called structural operational semantics (see [45] and Section 5.4 later), including type-checking systems. 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.

4.6 Reflection in rewriting logic

Clavel and Meseguer have shown in [17, 18] that rewriting logic is reflective in the sense of Section 2.8. 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 $\mathcal{L} \rightarrow \mathit{OSRWLogic}$ 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 4.3.

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 [79], 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 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 -> Constants [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 .
```

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 4.3. First note that the rewrite theory `LINLOG` presented in Section 4.3 gives rise to a term in `RWLTheory` that we denote

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

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

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

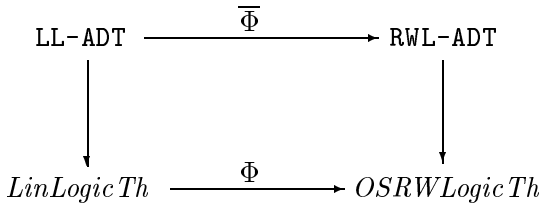
where the auxiliary operations `cons`, `tensor` and `par` are defined as follows, and correspond exactly to the description in Section 4.3.

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

var F : Formula .
vars F1 F2 : Formulas .
var I : Id .
var L : Ids .

eq tensor(null) = 1 .
eq tensor(F) = F .
eq tensor(F1,F2) = tensor(F1) ⊗ tensor(F2) .
eq par(null) = ⊥ .
eq par(F) = F .
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 [10], that it is specifiable by a finite set of Church-Rosser and terminating equations inside rewriting logic.

5 Rewriting logic as a semantic framework

After an overview of rewriting logic as a general model of computation that unifies many other existing models, the cases of concurrent object-oriented programming and of Milner's CCS are treated in greater detail. Structural operational semantics is discussed as a specification formalism similar in some ways to rewriting logic, but more limited in its expressive capabilities. Rewriting logic can also be very useful as a semantic framework for many varieties of constraint solving in logic programming and in automated deduction. Finally, the representation of action and change in rewriting logic and the consequent solution of the "frame problem" difficulties associated with standard logics are also discussed.

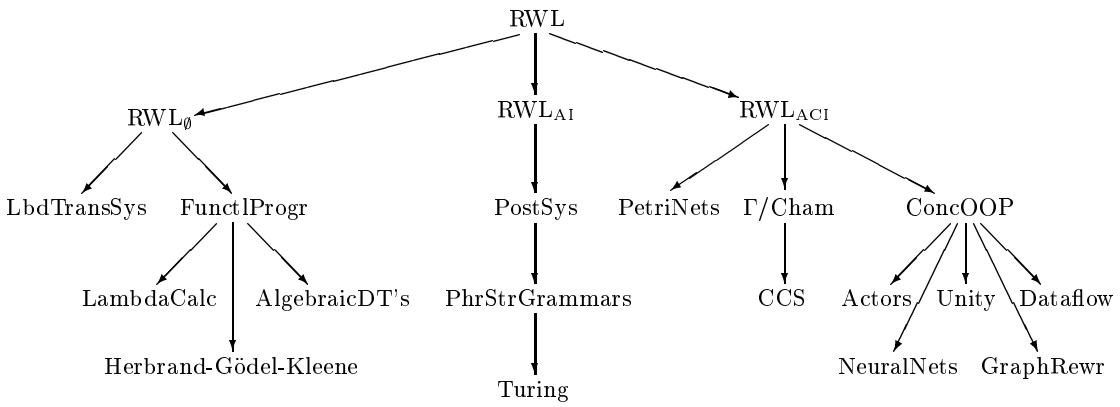


Figure 1: Unification of models of computation.

5.1 Generality of rewriting logic as a model of computation

Concurrent rewriting is a very general model of concurrency from which many other models can be obtained by specialization. Except for concurrent objet-oriented programming and CCS that are further discussed in Sections 5.2 and 5.3, respectively, we refer the reader to [72, 76] for a detailed discussion of the remaining models, and summarize here such specializations using Figure 1, where RWL stands for rewriting logic, the arrows indicate specializations, and the subscripts $\emptyset$, AI, and ACI stand for syntactic rewriting, rewriting modulo associativity and identity, and rewriting modulo associativity, commutativity, and identity, respectively.

Within syntactic rewriting we have labelled transition systems, which are used in interleaving approaches to concurrency; functional programming (in particular Maude's functional modules) corresponds to the case of *confluent*[13] rules, and includes the lambda calculus and the Herbrand-Gödel-Kleene theory of recursive functions. Rewriting modulo AI yields Post systems and related grammar formalisms, including Turing machines. Besides the general treatment by ACI-rewriting of concurrent object-oriented programming, briefly described in Section 5.2, that contains Actors [2], neural networks, graph rewriting and the dataflow model as a special case [76], rewriting modulo ACI includes Petri nets [96], the Gamma language of Banâtre and Le Mètayer [6], and Berry and Boudol's *chemical abstract machine* [11] (which itself specializes to CCS [84]; see [11] and also the treatment in Section 5.3), as well as Unity's model of computation [19].

The ACI case is quite important, since it contains as special subcases a good number of concurrency models that have already been studied. In fact, the associativity and commutativity of the axioms appear in some of those models as "fundamental laws of concurrency." However, from the perspective of this work the ACI case, while being important and useful, does not have a monopoly on the concurrency business. Indeed, "fundamental laws of concurrency" expressing associativity and commutativity are only valid in this particular case. They are for example meaningless for the tree-structured

[13] Although not reflected in the picture, rules confluent *modulo* equations E are also functional.

case of functional programming. The point is that the laws satisfied by a concurrent system cannot be determined *a priori*. They essentially depend on the actual distributed structure of the system, which is its algebraic structure.

5.2 Concurrent object-oriented programming

Concurrent object-oriented programming is a very active area of research. An important reason for this interest is the naturalness with which this style of programming can model concurrent interactions between objects in the real world. However, the field of concurrent object-oriented programming seems at present to lack a clear, agreed-upon semantic basis.

Rewriting logic supports a logical theory of concurrent objects that addresses these conceptual needs in a very direct way. We summarize here the key ideas regarding Maude's object-oriented modules; a full discussion of Maude's object-oriented aspects can be found in [74, 75].

An *object* in a given state can be represented as a term

```
< O : C | a1 : v1, ... , an : vn >
```

where `O` is the object's name, belonging to a set `OId` of object identifiers, `C` is its class, the `ai`'s are the names of the object's *attributes*, and the `vi`'s are their corresponding values, which typically are required to be in a sort appropriate for their corresponding attribute. The *configuration* is the distributed state of the concurrent object-oriented system and is represented as a multiset of objects and messages according to the following syntax:

```
subsorts Object Message < Configuration .
op __ : Configuration Configuration -> Configuration
                                    [assoc comm id: null] .
```

where the operator `__` is associative and commutative with identity `null` and is interpreted as multiset union, and where the sorts `Object` and `Message` are subsorts of `Configuration` and generate data of that sort by multiset union. The system evolves by concurrent *ACI*-rewriting of the configuration by means of rewrite rules specific to each particular system, whose lefthand and righthand sides may in general involve patterns for several objects and messages. By specializing to patterns involving only one object and one message, we can obtain an abstract, declarative, and truly concurrent version of the Actor model [2] (see [74, Section 4.7]).

Maude's syntax for object-oriented modules is illustrated by the object-oriented module `ACCNT` below which specifies the concurrent behavior of objects in a very simple class `Accnt` of bank accounts, each having a `bal`(ance) attribute, which may receive messages for crediting or debiting the account, or for transferring funds between two accounts. We assume an already defined functional module `INT` for integers with a subsort relation `Nat < Int` and an ordering predicate `_>=_`.

After the keyword `class`, the name of the class (`Accnt` in this case) is given, followed by a "`|`" and by a list of pairs of the form `a : S` separated by commas, where `a` is an attribute identifier and `S` is the sort inside which the values of such an attribute identifier must range in the given class. In this example, the only attribute of an account is its `bal`(ance), which

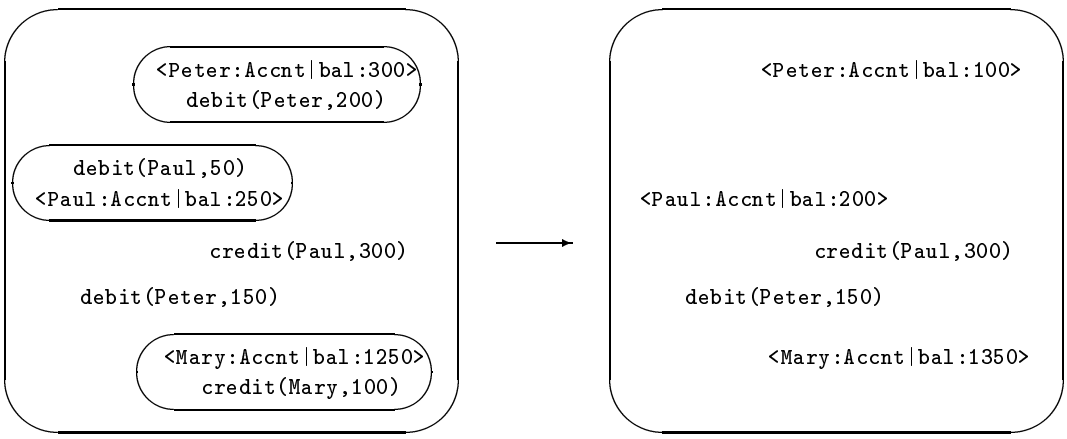


Figure 2: Concurrent rewriting of bank accounts.

is declared to be a value in `Nat`. The three kinds of messages involving accounts are `credit`, `debit`, and `transfer` messages, whose user-definable syntax is introduced by the keyword `msg`. The rewrite rules specify in a declarative way the behavior associated to the `credit`, `debit`, and `transfer` messages.

```
omod ACCNT is
  protecting INT .
  class Accnt | bal : Nat .
  msgs credit debit : OId Nat -> Msg .
  msg transfer_from_to_ : Nat OId OId -> Msg .
  vars A B : OId .
  vars M N N' : Nat .
  rl credit(A,M) < A : Accnt | bal: N > => < A : Accnt | bal: N + M > .
  rl debit(A,M) < A : Accnt | bal: N > => < A : Accnt | bal: N - M >
     if  N >= M .
  rl transfer M from A to B
       < A : Accnt | bal: N > < B : Accnt | bal: N' >
     => < A : Accnt | bal: N - M > < B : Accnt | bal: N' + M >
     if  N >= M .
endom
```

The multiset structure of the configuration provides the top level distributed structure of the system and allows concurrent application of the rules. For example, Figure 2 provides a snapshot in the evolution by concurrent rewriting of a simple configuration of bank accounts. To simplify the picture, the arithmetic operations required to update balances have already been performed. However, the reader should bear in mind that the values in the attributes of an object can also be computed by means of rewrite rules, and this adds yet another important level of concurrency to a concurrent object-oriented system, which might be called *intra-object concurrency*.

Intuitively, we can think of messages as "traveling" to come into contact with the objects to which they are sent and then causing "communication events" by application

of rewrite rules. In rewriting logic, this traveling is accounted for in a very abstract way by the ACI structural axioms. This abstract level supports both synchronous and asynchronous communication and provides great freedom and flexibility to consider a variety of alternative implementations at lower levels.

Although Maude provides convenient syntax for object-oriented modules, the syntax and semantics of such modules can be reduced to those of system modules, i.e., we can systematically translate an object-oriented module `omod` $\mathcal{O}$ `endom` into a corresponding system module `mod` $\mathcal{O}\#$ `endm`, where $\mathcal{O}\#$ is a theory in rewriting logic. A detailed account of this translation process can be found in [74].

5.3 CCS

Milner's *Calculus of Communicating Systems* (CCS) [83, 84, 85] is among the best well-known and studied concurrency models, and has become the paradigmatic example of an entire approach to "process algebras." We just give a very brief introduction to CCS, referring the reader to Milner's book [84] for motivation and a comprehensive treatment, before giving two alternative formulations of CCS in rewriting logic and showing the conservativity of these formulations.

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. The following modules have been motivated by, but are considerably different from, the corresponding examples in [78].

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

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

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

--- CCS syntax
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 [79], 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.

```
--- defining equations and contexts
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 [84, 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 4.5.

```
--- CCS transitions
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:

```
{A}(P | Q) ⟶  {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 [35], cannot be applied unless the resulting term $f(t_1, \ldots, t_n)$ is well formed according to the given order-sorted signature. To illustrate this point further, although `A.P` $\longrightarrow$ `{A}P` is a correct instance of the `Prefix` rewrite rule, we cannot use the *Congruence* rule to derive

```
(A.P) | Q ⟶ ({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 `{A1}...{Ak}P` can only be rewritten into another term of the same form `{A1}...{Ak}{B}P'`, assuming in this case that `P` $\longrightarrow$ `{B}P'` is 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 `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 16 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 `{a1}...{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.

Let us consider now the second possibility, using the same idea described in Section 4.5 for the linear logic sequent calculus, that, as we have already mentioned there, is applicable to many more cases, with a very broad understanding of the term "sequent."

```
--- CCS operational semantics
mod CCS2[X :: LABEL, Y :: PROCESSID, C :: CCS-CONTEXT*[X,Y]] is
  --- a configuration is a multiset of transitions
  sort Configuration .
  op (_:_-->_) : Act Process Process -> Configuration .
  op empty : -> Configuration .
  op __ : Configuration Configuration -> Configuration
                                       [assoc comm id: empty] .
  vars L M : Label .
  var A : Act .
  vars P P' Q Q' : Process .
  var X : ProcessId .
  *** Prefix
  rl              empty
      => --- -----------------
         (A : (A . P) --> P) .

  *** Summation
  rl        (A : P --> P')
      => --- ----------------
         (A : P + Q --> P') .

  *** Composition
  rl          (A : P --> P')
      => --- --------------------
         (A : P | Q --> P' | Q) .

  rl     (L : P --> P')(~L : Q --> Q')
      => --- -------------------------
          (tau : P | Q --> P' | Q') .

  *** Restriction
  crl         (A : P --> P')
      => --- ------------------
         (A : P \ L --> P' \ L)
      if  not(A == L)  and  not(A == ~L) .

  *** Relabelling
  rl             (L : P --> P')
      => --- -------------------------
         (M : P[M / L] --> P'[M / L]) .

  rl             (~L : P --> P')
      => --- --------------------------
```

```
        (~M : P[M / L] --> P'[M / L]) .

  crl           (A : P --> P')
      => --- ------------------------
         (A : P[M / L] --> P'[M / L])
      if  not(A == L)  and  not(A == ~L) .

  *** Definition
  crl    (A : P --> P')
      => --- ----------
         (A : X --> P')
      if (X =def P) in context .
endm
```

Except for the difference in the number of rules for some operators, as already pointed out above for the module `CCS1[X,Y,C[X,Y]]`, this presentation is closer to the original one, and therefore the following conservativity result is immediate.

Theorem 17 For CCS processes P and P', a transition $P \xrightarrow{A} P'$ is possible according to the structural operational semantics of CCS if and only if

```
empty ⟶ (A : P --> P')
```

is provable in rewriting logic from the rewrite theory `CCS2[X,Y,C[X,Y]]`.

5.4 Structural operational semantics

Structural operational semantics is an approach originally introduced by Plotkin [93] in which the operational semantics of a programming language is specified in a logical way, independent of machine architecture or implementation details, by means of rules that provide an inductive definition based on the structure of the expressions in the language. We refer the reader to Hennessy's book [45] for a clear introduction to this subject.

Within "structural operational semantics," two main approaches coexist:

- *Big-step semantics* (also called *natural semantics* by Kahn [54], Gunter [40], and Nielson and Nielson [88], and *evaluation semantics* by Hennessy [45]). In this approach, the main inductive predicate describes the overall result or value of executing a computation until its termination. For this reason, it is not well suited for languages like CCS where most programs are not intended to be terminating.

- *Small-step semantics* (also called *structural operational semantics* by Plotkin [93], and Nielson and Nielson [88], *computation semantics* by Hennessy [45], and *transition semantics* by Gunter [40]). In this approach, the main inductive predicate describes in more detail the execution of individual steps in a computation, with the overall computation roughly corresponding to the transitive closure of such small steps. The structural operational semantics of CCS presented at the beginning of Section 5.3 is an example.

Both big-step and small-step approaches to structural operational semantics can be naturally expressed in rewriting logic:

- Big-step semantics can be seen as a particular case of the mapping of sequent systems described in Section 4.5, where semantics rules are mapped to rewrite rules over a "configuration" of sequents or predicates, and the rewriting relation means provability of such a predicate.
- Small-step semantics corresponds to the use of conditional rewrite rules, where a rewrite $t \longrightarrow t'$ means a transition or computation step from a state t to a new state t' as in the explanation of rewriting logic given in Section 3.3. This is illustrated by the `CCS1[X,Y,C[X,Y]]` example in Section 5.3. However, as the `CCS2[X,Y,C[X,Y]]` example shows, the technique of sequent systems of Section 4.5 can also be used in this case.

Since the CCS example has already been discussed in detail in Section 5.3, we give here another example, describing the operational semantics of the functional language Mini-ML taken with slight modifications from Kahn's paper [54]. The first thing to point out about this example is that the specification of a language's syntax is outside of the structural operational semantics formalism. By contrast, thanks to the order-sorted type structure of rewriting logic, such specification is now given by a functional module in Maude, as follows:

```
fmod NAT is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
endfm

fmod TRUTH-VAL is
  sort TruthVal .
  ops true false : -> TruthVal .
endfm

--- syntax: values, patterns and expressions
fmod ML-SYNTAX[X :: VAR] is
  protecting NAT TRUTH-VAL .
  sorts Exp Value Pat NullPat Lambda .

  subsorts NullPat Var < Pat .
  op () : -> NullPat .
  op (_,_) : Pat Pat -> Pat .
  subsorts TruthVal Nat NullPat < Value .
  op (_,_) : Value Value -> Value .
  subsorts Value Var Lambda < Exp .
  op s : Exp -> Exp .
```

```
  op _+_ : Exp Exp -> Exp [comm] .
  op not : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
  op if_then_else_ : Exp Exp Exp -> Exp .
  op (_,_) : Exp Exp -> Exp .
  op __ : Exp Exp -> Exp .
  op λ_._ : Pat Exp -> Lambda .
  op let_=_in_ : Pat Exp Exp -> Exp .
  op letrec_=_in_ : Pat Exp Exp -> Exp .
endfm

--- environments are lists of pairs pattern-value
fmod AUX[X :: VAR] is
  protecting ML-SYNTAX[X] .
  sort Pair .
  op <_,_> : Pat Value -> Pair .
  protecting LIST[Pair]*(sort List to Env, op _;_ to __) .
  op Clos : Lambda Env -> Value .
endfm
```

The following module constitutes a direct translation of the natural semantics specification for Mini-ML given by Kahn in [54], using the general technique for sequent systems introduced in Section 4.5. Note that the natural semantics rules are particularly well suited for Prolog search, and indeed they are so executed in the system described in [54].

```
--- natural semantics a la Kahn
mod ML-NAT-SEMANT[X :: VAR] is
  extending AUX[X] .
  sort Config .
  op (_|-_-->_) : Env Exp Value -> Config .
  op empty : -> Config .
  op __ : Config Config -> Config [assoc comm id: empty] .

  vars V W : Env .
  vars E F G : Exp .
  vars X Y : Var .
  vars P Q : Pat .
  vars A B C : Value .
  vars N M : Nat .
  var T : TruthVal .

  *** Variables
  rl              empty
     => --- -------------------
        ((V <X,A>) |- X --> A).
```

```
crl       (V |- X --> A)
   => --- ------------------
      ((V <Y,B>) |- X --> A)
   if  not(X == Y) .

rl      (V <P,A> <Q,B> |- X --> C)
   => --- --------------------------
      (V <(P,Q),(A,B)> |- X --> C) .

*** Arithmetic expressions
rl         empty
   => --- ------------
      (V |- 0 --> 0) .

rl        (V |- E --> A)
   => --- ------------------
      (V |- s(E) --> s(A)) .

crl   (V |- E --> A)(V |- F --> B)
   => --- ------------------------
         (V |- E + F --> C)
   if  A + B => C .

rl 0 + N => N .
rl s(N) + s(M) => s(s(N + M)) .

*** Boolean expressions
rl            empty
   => --- ------------------
      (V |- true --> true) .

rl             empty
   => --- --------------------
      (V |- false --> false) .

rl        (V |- E --> true)
   => --- ---------------------
      (V |- not(E) --> false) .

rl       (V |- E --> false)
   => --- --------------------
      (V |- not(E) --> true) .
```

```
  crl   (V |- E --> A)(V |- F --> B)
     => --- ------------------------
           (V |- E and F --> C)
     if  (A and B) => C .

  rl T and true => T .
  rl T and false => false .

  *** Conditional expressions
  rl     (V |- E --> true)(V |- F --> A)
     => --- ----------------------------
        (V |- if E then F else G --> A) .

  rl    (V |- E --> false)(V |- G --> A)
     => --- ----------------------------
        (V |- if E then F else G --> A) .

  *** Pair expressions
  rl          empty
     => --- --------------
        (V |- () --> ()) .

  rl    (V |- E --> A)(V |- F --> B)
     => --- ------------------------
          (V |- (E,F) --> (A,B)) .

  *** Lambda expressions
  rl                empty
     => --- --------------------------
        (V |- λP.E --> Clos(λP.E,V)) .

  rl    (V |- E --> Clos(λP.G,W))(V |- F --> A)(W <P,A> |- G --> B)
     => --- ------------------------------------------------------
                              (V |- E F --> B) .

  *** Let and letrec expressions
  rl    (V |- F --> A)(V <P,A> |- E --> B)
     => --- ----------------------------
           (V |- let P = F in E --> B) .

  rl    (V <P,A> |- F --> A)(V <P,A> |- E --> B)
     => --- ----------------------------------
           (V |- letrec P = F in E --> B) .
endm
```

The following module gives an alternative description of the semantics of the Mini-ML language in terms of the small-step approach. The rules can be directly used to perform reduction on Mini-ML expressions, and therefore constitute a very natural functional interpreter for the language.

```
--- sos semantics
mod ML-SOS-SEMANT[X :: VAR] is
  extending AUX[X] .
  op [[_]]_ : Exp Env -> Value .

  vars V W : Env .
  vars E F G : Exp .
  vars X Y : Var .
  vars P Q : Pat .
  vars A B : Value .

  *** Variables
  rl [[X]](V <X,A>) => A .
  crl [[X]](V <Y,B>) => [[X]]V  if  not(X == Y) .
  rl [[X]](V <(P,Q),(A,B)>) => [[X]](V <P,A> <Q,B>) .

  *** Arithmetic expressions
  rl 0 + E => E .
  rl s(E) + s(F) => s(s(E + F)) .
  rl [[0]]V => 0 .
  rl [[s(E)]]V => s([[E]]V) .
  rl [[E + F]]V => [[E]]V + [[F]]V .

  *** Boolean expressions
  rl not(false) => true .
  rl not(true) => false .
  rl E and true => E .
  rl E and false => false .
  rl [[true]]V => true .
  rl [[false]]V => false .
  rl [[not(E)]]V => not([[E]]V) .
  rl [[E and F]]V => [[E]]V and [[F]]V .

  *** Conditional expressions
  rl if true then E else F => E .
  rl if false then E else F => F .
  rl [[if E then F else G]]V => if [[E]]V then [[F]]V else [[G]]V .

  *** Pair expressions
  rl [[()]]V => () .
```

```
  rl [[(E,F)]]V => ([[E]]V,[[F]]V) .

  *** Lambda expressions
  rl [[λP.E]]V => Clos(λP.E,V) .
  rl [[E F]]V => [[E]]V [[F]]V .
  rl Clos(λP.E,W) [[F]]V => [[E]](W <P,[[F]]V>) .

  *** Let and letrec expressions
  rl [[let P = E in F]]V => [[F]](V <P,[[E]]V>) .
  crl [[letrec P = E in F]]V => [[F]](V <P,A>)  if  [[E]]((V <P,A>) => A .
endm
```

This concludes our discussion of structural operational semantics. Compared with rewriting logic, one of its limitations is the lack of support for structural axioms yielding more abstract data representations. Therefore, the rules must follow a purely syntactic structure, and more rules may in some cases be necessary than if an abstract representation had been chosen. In the case of multiset representations (corresponding to associativity, commutativity, and identity axioms), this has led Milner to favor multiset rewriting presentations [86] in the style of the chemical abstract machine of Berry and Boudol [11] over the traditional syntactic presentation of structural operational semantics.

5.5 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 [53]), which corresponds to solving equations in a given equational theory.
- *Sorted unification*, either many-sorted or order-sorted [111, 112, 99, 80, 104, 53], where type constraints are added to variables in equations.
- *Higher-order unification* [49, 82], which corresponds to solving equations between λ-expressions.

- *Disunification* [22], which corresponds to solving not only equalities but also negated equalities.

- *Solution of equalities and inequalities in a theory*, as for example the solution of numerical constraints built into the constraint logic programming language $CLP(\mathcal{R})$ [50] 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 [60], 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 [34, 53, 21, 22, 90]). In particular, the survey by Jouannaud and Kirchner [53] 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 [55] (who use a general notion of constraint language proposed by Smolka [103]), Bachmair, Ganzinger, Lynch, and Snyder [5], Nieuwenhuis and Rubio [89], and Giunchiglia, Pecchiari, and Talcott [30]. In particular, the ELAN language of C. Kirchner, H. Kirchner, and M. Vittek [56, 12] 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.

Exploring the use of rewriting logic as a semantic framework for languages and theorem-proving systems using constraints seems a worthwhile research direction not only for systems used in automated deduction, but also for parallel logic programming languages such as those surveyed in [102], the Andorra language [52], concurrent constraint programming [98], and the Oz language [46].

5.6 Action and change in rewriting logic

In the previous sections, we have shown the advantages of rewriting logic as a logical framework in which other logics can be represented, and as a semantic framework for the specification of languages and systems. We would like the class of systems that can be represented to be as wide as possible, and their representation to be as natural and direct as possible. In particular, an important point that has to be considered is the representation of action and change in rewriting logic. In our paper [63], we show that rewriting logic overcomes the frame problem, and subsumes and unifies a number of previously proposed logics of change. In this section, we illustrate this claim by means of an example, referring the reader to the cited paper for more examples and discussion.

The frame problem [69, 44, 51] consists in formalizing the assumption that facts are preserved by an action unless the action explicitly says that a certain fact becomes true or false. In the words of Patrick Hayes [44],

> "There should be some economical and principled way of succintly saying what changes an action makes, without having to explicitly list all the things it doesn't change as well [...]. *That* is the frame problem."

Recently, some new logics of action and change have been proposed, among which we can point out the approach of Hölldobler and Schneeberger [48, 38, 39], based on Horn logic with equations, and the approach of Masseron, Tollu, and Vauzeilles [65, 66], based on linear logic. The main interest of these formalisms is that they need not explicitly state frame axioms, because they treat facts as resources which are produced and consumed. Having proved in Sections 4.2 and 4.3, respectively, that Horn logic with equations and linear logic can be conservatively mapped into rewriting logic, it is not surprising that the advantages of the two previously mentioned approaches are also shared by rewriting logic. In particular, the rewriting logic rules automatically take care of the task of preserving context, making unnecessary the use of any frame axioms stating the properties that do not change when a rule is applied to a certain state.

We illustrate this point by means of a blocksworld example, borrowed from [48, 65].

```
fth BLOCKS is
  sort BlockId .
endft

mod BLOCKWORLD[X :: BLOCKS] is
  sort Prop .
  op table : BlockId -> Prop .       *** block is on the table
  op on : BlockId BlockId -> Prop .  *** block A is on block B
```

```
  op clear : BlockId -> Prop .        *** block is clear
  op hold : BlockId -> Prop .         *** robot hand is holding the block
  op empty : -> Prop .                *** robot hand is empty

  sort State .
  subsort Prop < State .
  op 1 : -> State .
  op _⊗_ : State State -> State [assoc comm id: 1] .

  vars X Y : BlockId .
  rl pickup(X) : empty ⊗ clear(X) ⊗ table(X) => hold(X) .
  rl putdown(X) : hold(X) => empty ⊗ clear(X) ⊗ table(X) .
  rl unstack(X,Y) : empty ⊗ clear(X) ⊗ on(X,Y) => hold(X) ⊗ clear(Y) .
  rl stack(X,Y) : hold(X) ⊗ clear(Y) => empty ⊗ clear(X) ⊗ on(X,Y) .
endm
```

In order to create a world with three blocks $\{a, b, c\}$, we consider the following instantiation of the previous parameterized module.

```
fmod BLOCKS3 is
  sort BlockId .
  ops a b c : -> BlockId .
endfm

  make WORLD is BLOCKWORLD[BLOCKS3] endmk
```

Consider the states described in Figure 3; the state I on the left is the initial one, described by the following term of sort `State` in the rewrite theory (Maude program) `WORLD`

```
  empty ⊗ clear(c) ⊗ clear(b) ⊗ table(a) ⊗ table(b) ⊗ on(c,a) .
```

Analogously, the final state F on the right is described by the term

```
  empty ⊗ clear(a) ⊗ table(c) ⊗ on(a,b) ⊗ on(b,c) .
```

The fact that the plan

```
  unstack(c,a);putdown(c);pickup(b);stack(b,c);pickup(a);stack(a,b)
```

moves the blocks from state I to state F corresponds directly to the following `WORLD`-rewrite (proof in rewriting logic), where we also show the use of the structural axioms of associativity and commutativity:

```
empty ⊗ clear(c) ⊗ clear(b) ⊗ table(a) ⊗ table(b) ⊗ on(c,a)
 =
empty ⊗ clear(c) ⊗ on(c,a) ⊗ clear(b) ⊗ table(a) ⊗ table(b)
⟶      Cong[Repl[unstack(c,a)],Refl]
```

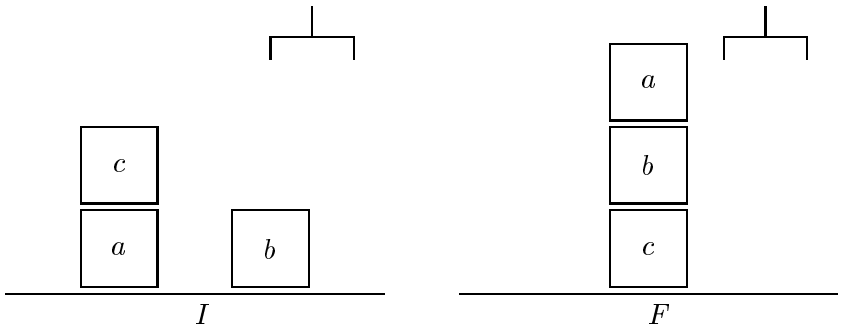


Figure 3: Two states of a blocksworld.

```
hold(c) ⊗ clear(a) ⊗ clear(b) ⊗ table(a) ⊗ table(b)
−→      Cong[Repl[putdown(c)],Refl]
empty ⊗ clear(c) ⊗ table(c) ⊗ clear(a) ⊗ clear(b) ⊗ table(a) ⊗ table(b)
 =
empty ⊗ clear(b) ⊗ table(b) ⊗ clear(c) ⊗ table(c) ⊗ clear(a) ⊗ table(a)
−→      Cong[Repl[pickup(b)],Refl]
hold(b) ⊗ clear(c) ⊗ table(c) ⊗ clear(a) ⊗ table(a)
−→      Cong[Repl[stack(b,c)],Refl]
empty ⊗ clear(b) ⊗ on(b,c) ⊗ table(c) ⊗ clear(a) ⊗ table(a)
 =
empty ⊗ clear(a) ⊗ table(a) ⊗ clear(b) ⊗ on(b,c) ⊗ table(c)
−→      Cong[Repl[pickup(a)],Refl]
hold(a) ⊗ clear(b) ⊗ on(b,c) ⊗ table(c)
−→      Cong[Repl[stack(a,b)],Refl]
empty ⊗ clear(a) ⊗ on(a,b) ⊗ on(b,c) ⊗ table(c)
 =
empty ⊗ clear(a) ⊗ table(c) ⊗ on(a,b) ⊗ on(b,c)
```

Hopefully this notation is self-explanatory. For example, *Cong*[*Repl*[`pickup(b)`],*Refl*] means the application of the *Congruence* rule of rewriting logic to the two `WORLD`-rewrites obtained by using *Replacement* with the rewrite rule `pickup(b)` and *Reflexivity*. Of course, *Transitivity* is used several times to go from the initial state I to the final state F.

Große, Hölldobler and Schneeberger prove in [38] (see also [39, 47]) that, in the framework of conjunctive planning, there is an equivalence between plans generated by linear logic proofs as used by Masseron *et al.* [65, 66], and the equational Horn logic approach of Hölldobler and Schneeberger [48]. In the light of the example above, it is not surprising that we can add to the above equivalence the plans generated by proofs in rewriting logic [63]. Moreover, this result extends to the case of disjunctive planning [14, 63]. In our opinion, rewriting logic compares favorably with these formalisms, not only because it subsumes them, but also because it is intrinsically concurrent, and it is more flexible and general, supporting user-definable logical connectives, which can be chosen to fit the problem at hand. In the words of Reichwein, Fiadeiro and Maibaum [95],

> "It is not enough to have a convenient formalism in which to represent action and change: the representation has to reflect the structure of the represented system."

In this respect, we show in [63] that the object-oriented point of view supported by rewriting logic becomes very helpful in order to represent action and change.

6 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 [70] 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 [105] and Feferman [25].

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. 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.

Although this paper has studied the use of rewriting logic as a logical framework, and not as a metalogical one in which metalevel resoning about an object logic is performed, this second use is not excluded and is indeed one of the most interesting research directions that we plan to study. For this purpose, as stressed by Constable [3], we regard *reflection* as a key technique to be employed. Some concrete evidence for the usefulness of reflection has been given in Section 4.6.

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.

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.
- *Abstract user-definable syntax*, which can be specified as an order-sorted equational data type with the desired structural axioms.
- *Modularity and parameterization*[14], 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 [16]. A very important additional feature that the Maude interpreter has is good support for flexible and expressive strategies of evaluation [16, 18], so that the user can explore the space of rewritings in intelligent ways.

Acknowledgements

We would like to thank Manuel Clavel, Robert Constable, Harmut Ehrig, Fabio Gadducci, Claude Kirchner, Hélène Kirchner, Patrick Lincoln, Ugo Montanari, Natarajan Shankar, Sam Owre, Gordon Plotkin, Axel Poigné, and Carolyn Talcott for their comments and suggestions that have helped us improve the final version of this paper. We are also grateful to the participants of the May 1993 Dagstuhl Seminar on Specification and Semantics, where this work was first presented, for their encouragement of, and constructive comments on, these ideas.

References

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

[2] G. Agha, *Actors*, The MIT Press, 1986.

[3] W. E. Aitken, R. L. Constable, and J. L. Underwood, *Metalogical frameworks II: Using reflected decision procedures*, Technical report, Computer Science Department, Cornell University, 1995. Also, lecture by R. L. Constable at the Max Planck Institut für Informatik, Saarbrücken, Germany, July 21, 1993.

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

[4] E. Astesiano and M. Cerioli, Relationships between logical frameworks, in: M. Bidoit and C. Choppy (eds.), *Recent Trends in Data Type Specification*, LNCS 655, Springer-Verlag, 1993, pages 126–143.

[5] 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, June 1992*, LNCS (subseries LNAI) 607, Springer-Verlag, 1992, pages 462–476.

[6] J.-P. Banâtre and D. Le Mètayer, The Gamma model and its discipline of programming, *Science of Computer Programming* **15**, 1990, pages 55–77.

[7] M. Barr, $*$-*Autonomous Categories*, Lecture Notes in Mathematics 752, Springer-Verlag, 1979.

[8] M. Barr and C. Wells, *Toposes, Triples and Theories*, Springer-Verlag, 1985.

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

[10] 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, pages 76–90.

[11] G. Berry and G. Boudol, The chemical abstract machine, *Theoretical Computer Science* **96**, 1992, pages 217–248.

[12] 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.

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

[14] S. Brüning, G. Große, S. Hölldobler, J. Schneeberger, U. Sigmund, and M. Thielscher, Disjunction in plan generation by equational logic programming, in: A. Horz (ed.), *Beiträge zum 7. Workshop Plannen und Konfigurieren*, Arbeitspapiere der GMD 723, 1993, pages 18–26.

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

[16] 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.

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

[18] 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.

[19] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[20] 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, pages 45–65.

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

[22] 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, pages 322–359.

[23] 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, pages 243–320.

[24] H. Ehrig, M. Baldamus, and F. Cornelius, Theory of algebraic module specification including behavioural semantics, constraints and aspects of generalized morphisms, in *Proc. Second Int. Conf. on Algebraic Methodology and Software Technology*, Iowa City, Iowa, 1991, pages 101–125.

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

[26] 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, July 1990*, LNCS (subseries LNAI) 449, Springer-Verlag, 1990, pages 221–235.

[27] J. Fiadeiro and A. Sernadas, Structuring theories on consequence, in: D. Sannella and A. Tarlecki (eds.), *Recent Trends in Data Type Specification*, LNCS 332, Springer-Verlag, 1988, pages 44–72.

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

[29] J.-Y. Girard, Linear logic, *Theoretical Computer Science* **50**, 1987, pages 1–102.

[30] F. Giunchiglia, P. Pecchiari, and C. Talcott, *An architecture for open mechanized reasoning systems*, manuscript, August 1993.

[31] 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, pages 221–256.

[32] J. A. Goguen and R. M. Burstall, A study in the foundations of programming methodology: Specifications, institutions, charters and parchments, in: D. Pitt *et al.* (eds.), *Proc. Workshop on Category Theory and Computer Programming, Guildford, UK, September 1985*, LNCS 240, Springer-Verlag, 1986, pages 313–333.

[33] 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, pages 95–146.

[34] J. A. Goguen and J. Meseguer, Software for the Rewrite Rule Machine, in: *Proc. of the Int. Conf. on Fifth Generation Computer Systems, Tokyo, Japan*, ICOT, 1988, pages 628–637.

[35] 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, pages 217–273.

[36] 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, pages 69–86.

[37] 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.

[38] G. Große, S. Hölldobler, and J. Schneeberger, Linear deductive planning, *Journal of Logic and Computation* **6**(2), 1996, pages 233–262.

[39] G. Große, S. Hölldobler, J. Schneeberger, U. Sigmund, and M. Thielscher, Equational logic programming, actions, and change, in: K. Apt (ed.), *Proc. Int. Joint Conf. and Symp. on Logic Programming*, The MIT Press, 1992, pages 177–191.

[40] C. Gunter, Forms of semantic specification, *Bulletin of the EATCS* **45**, October 1991, pages 98–113.

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

[42] 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, pages 226–237.

[43] R. Harper, D. Sannella, and A. Tarlecki, Logic representation in LF, in: D. H. Pitt *et al.* (eds.), *Category Theory and Computer Science, Manchester, UK, September 1989*, LNCS 389, Springer-Verlag, 1989, pages 250–272.

[44] P. J. Hayes, What the frame problem is and isn't, in: Z. W. Pylyshyn (ed.), *The Robot's Dilemma: The Frame Problem in Artificial Intelligence*, Ablex Publishing Corp., 1987, pages 123–137.

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

[46] M. Henz, G. Smolka, and J. Würtz, Object-oriented concurrent constraint programming in Oz, in: V. Saraswat and P. van Hentenryck, eds., *Principles and Practice of Constraint Systems: The Newport Papers*, The MIT Press, 1995, pages 29–48.

[47] S. Hölldobler, On deductive planning and the frame problem, in: A. Voronkov (ed.), *Logic Programming and Automated Reasoning, St. Petersburg, Russia, July 1992*, LNAI 624, Springer-Verlag, 1992, pages 13–29.

[48] S. Hölldobler and J. Schneeberger, A new deductive approach to planning, *New Generation Computing* **8**, 1990, pages 225–244.

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

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

[51] L.-E. Janlert, Modeling change—The frame problem, in: Z. W. Pylyshyn (ed.), *The Robot's Dilemma: The Frame Problem in Artificial Intelligence*, Ablex Publishing Corp., 1987, pages 1–40.

[52] S. Janson and S. Haridi, Programming paradigms of the Andorra kernel language, in: V. Saraswat and K. Ueda (eds.), *Proc. 1991 Int. Symp. on Logic Programming*, The MIT Press, 1991, pages 167–186.

[53] 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, pages 257–321.

[54] G. Kahn, *Natural semantics*, Technical report 601, INRIA Sophia Antipolis, February 1987.

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

[56] 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, pages 133–160.

[57] 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, September 1992*, LNCS 632, Springer-Verlag, 1992, pages 350–363.

[58] C. Laneve and U. Montanari, Axiomatizing permutation equivalence, *Mathematical Structures in Computer Science* **6**(3), 1996, pages 219–249.

[59] S. Mac Lane, *Categories for the Working Mathematician*, Springer-Verlag, 1971.

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

[61] S. Martini and A. Masini, *A computational interpretation of modal proofs*, Technical report TR-27/93, Dipartimento di Informatica, Università di Pisa, November 1993.

[62] 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, pages 297–399.

[63] N. Martí-Oliet and J. Meseguer, *Action and change in rewriting logic*, Technical report SRI-CSL-94-07, Computer Science Laboratory, SRI International, 1994. To appear in: B. Fronhöfer and R. Pareschi (eds.), *Theoretical Approaches to Dynamic Worlds in Artificial Intelligence and Computer Science*, Kluwer Academic Publishers.

[64] A. Masini, *A Proof Theory of Modalities for Computer Science*, Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa, 1993.

[65] M. Masseron, C. Tollu, and J. Vauzeilles, Generating plans in linear logic, in: K. V. Nori and C. E. Veni Madhavan (eds.), *Foundations of Software Technology and Theoretical Computer Science, Bangalore, India, December 1990*, LNCS 472, Springer-Verlag, 1990, pages 63–75.

[66] M. Masseron, C. Tollu, and J. Vauzeilles, Generating plans in linear logic I: Actions as proofs, *Theoretical Computer Science* **113**(2), 1993, pages 349–370.

[67] 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, pages 61–82.

[68] B. Mayoh, *Galleries and institutions*, Technical report DAIMI PB-191, Computer Science Department, Aarhus University, 1985.

[69] J. McCarthy and P. J. Hayes, Some philosophical problems from the standpoint of artificial intelligence, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 4*, Edinburgh University Press, 1969, pages 463–502.

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

[71] J. Meseguer, A logical theory of concurrent objects, in: N. Meyrowitz (ed.), *Proc. OOPSLA-ECOOP'90*, ACM Press, 1990, pages 101–115.

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

[73] J. Meseguer, Multiparadigm logic programming, in: H. Kirchner and G. Levi (eds.), *Proc. Third Int. Conf. on Algebraic and Logic Programming, Volterra, Italy, September 1992*, LNCS 632, Springer-Verlag, 1992, pages 158–200.

[74] 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, pages 314–390.

[75] J. Meseguer, Solving the inheritance anomaly in concurrent object-oriented programming, in: O. M. Nierstrasz (ed.), *Proc. ECOOP'93, 7th. European Conf., Kaiserslautern, Germany, July 1993*, LNCS 707, Springer-Verlag, 1993, pages 220–246.

[76] J. Meseguer, Rewriting logic as a semantic framework for concurrency: A progress report, in: U. Montanari and V. Sassone (eds.), *Proc. CONCUR'96*, LNCS 1119, Springer-Verlag, 1996, pages 331–372.

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

[78] J. Meseguer, K. Futatsugi, and T. Winkler, Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents, in: *Proc. IMSA'92, Int. Symp. on New Models for Software Architecture*, Tokyo, 1992.

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

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

[81] 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, pages 253–293.

[82] D. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, *Journal of Logic and Computation* **1**(4), 1991, pages 497–536.

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

[84] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[85] R. Milner, Operational and algebraic semantics of concurrent processes, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Vol. B*, The MIT Press/Elsevier, 1990, pages 1201–1242.

[86] R. Milner, Functions as processes, *Mathematical Structures in Computer Science* **2**(2), 1992, pages 119–141.

[87] 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, pages 810–827.

[88] H. R. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*, John Wiley and Sons, 1992.

[89] 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, June 1992*, LNCS (subseries LNAI) 607, Springer-Verlag, 1992, pages 477–491.

[90] T. Nipkow, Functional unification of higher-order patterns, in: *Proc. Eighth Annual IEEE Symp. on Logic in Computer Science*, Montreal, Canada, June 1993, pages 64–74.

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

[92] 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, pages 313–322.

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

[94] A. Poigné, Foundations are rich institutions, but institutions are poor foundations, in: H. Ehrig *et al.* (eds.), *Categorical Methods in Computer Science with Aspects from Topology*, LNCS 393, Springer-Verlag, 1989, pages 82–101.

[95] G. Reichwein, J. L. Fiadeiro, and T. Maibaum, Modular reasoning about change in an object-oriented framework, Abstract presented at the *Workshop Logic & Change at GWAI'92*, September 1992.

[96] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1985.

[97] A. Salibra and G. Scollo, A soft stairway to institutions, in: M. Bidoit and C. Choppy (eds.), *Recent Trends in Data Type Specification*, LNCS 655, Springer-Verlag, 1993, pages 310–329.

[98] V. J. Saraswat, *Concurrent Constraint Programming*, The MIT Press, 1992.

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

[100] D. Scott, Completeness and axiomatizability in many-valued logic, in: L. Henkin *et al.* (eds.), *Proceedings of the Tarski Symposium*, American Mathematical Society, 1974, pages 411–435.

[101] 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, American Mathematical Society, 1989, pages 371–382.

[102] E. Shapiro, The family of concurrent logic programming languages, *ACM Computing Surveys* **21**, 1989, pages 412–510.

[103] G. Smolka, *Logic Programming over Polymorphically Order-Sorted Types*, Ph.D. Thesis, Fachbereich Informatik, Universität Kaiserslautern, 1989.

[104] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer, Order-sorted equational computation, in: M. Nivat and H. Aït-Kaci (eds.), *Resolution of Equations in Algebraic Structures, Volume 2*, Academic Press, 1989, pages 297–367.

[105] R. M. Smullyan, *Theory of Formal Systems*, Annals of Mathematics Studies 47, Princeton University Press, 1961.

[106] C. Talcott, A theory of binding structures and applications to rewriting, *Theoretical Computer Science* **112**, 1993, pages 99–143.

[107] C. Talcott, *Binding syntax structures*, manuscript, August 1993.

[108] A. Tarlecki, Free constructions in algebraic institutions, in M. P. Chytil and V. Koubek (eds.), *Proc. Mathematical Foundations of Computer Science '84*, LNCS 176, Springer-Verlag, 1984, pages 526–534.

[109] A. Tarlecki, On the existence of free models in abstract algebraic institutions, *Theoretical Computer Science* **37**(3), 1985, pages 269–304.

[110] A. S. Troelstra, *Lectures on Linear Logic*, CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford University, 1992.

[111] C. Walther, A mechanical solution to Schubert's steamroller by many-sorted resolution, *Artificial Intelligence* **26**(2), 1985, pages 217–224.

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