Theoretical Computer Science 96 (1992) 73–155
Elsevier

Conditional rewriting logic as a unified model of concurrency

José Meseguer*

SRI International, Menlo Park, CA 94025, USA, and Center for the Study of Language and Information, Stanford University, Stanford, CA 94305, USA

Dedicated with affection to my mother, Fuensanta Guaita de Meseguer, on the occasion of her 80th birthday

Abstract

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

Rewriting with conditional rewrite rules modulo a set E of structural axioms provides a general framework for unifying a wide variety of models of concurrency. Concurrent rewriting coincides with logical deduction in *conditional rewriting logic*, a logic of actions whose models are concurrent systems. This logic is sound and complete and has initial models. In addition to general models interpreted as concurrent systems which provide a more operational style of semantics, more restricted semantics with an incresingly denotational flavor such as preorder, poset, cpo, and standard algebraic models appear as special cases of the model theory. This permits dealing with operational and denotational issues within the same model theory and logic. A programming language called Maude whose modules are rewriting logic theories is defined and given denotational and operational semantics. Maude provides a simple unification of concurrent programming with functional and object-oriented programming and supports high level declarative programming of concurrent systems.

Contents

*Supported by Office of Naval Research Contracts N00014-90-C-0086, N00014-88-C-0618 and N00014-86-C-0450, and NSF Grant CCR-8707155.

1. Introduction

The main goal of this paper is to propose a general and precise answer to the question:

What is a concurrent system?

It seems fair to say that this question has not yet received a satisfactory answer, and that the resulting situation is one of *conceptual fragmentation* within the field of concurrency. The field seems in need of *internal unification* because it is at present hard to relate very different approaches—each with a different set of basic concepts, models, and problems—such as for example Petri nets, algebraic approaches originating in CCS and CSP, Actors, and temporal-logic-based approaches. The problem of conceptual fragmentation appears not only across different approaches, but also within a given approach. For example, within the algebraic approach some problems are dealt with using denotational semantics methods, whereas other problems less amenable to a denotational treatment are dealt with using an operational semantics; however, a clear account of how to unify the operational and denotational viewpoints within a common semantic basis seems to be lacking. This work addresses this double need for conceptual unification

across and within different approaches. The need for a conceptual unification of concurrency is being felt quite strongly, and has also been addressed in other recent proposals such as the π-calculus [101] and the chemical abstract machine [16].

A related problem—which might be understood as the problem of achieving an *external unification* of concurrency with other areas—is the *integration of concurrent programming with other programming paradigms*, such as functional and object-oriented programming. Integration attempts typically graft an existing concurrency model on top of an existing language, but such *ad hoc* combinations often lead to monstrous deformities which are extremely difficult to understand. Instead, this paper proposes a *semantic integration* of those paradigms based on a common *logic and model theory*. Such an integration is embodied in a programming language called Maude whose basic syntax and mathematical semantics are also discussed in the paper. In Maude, concurrent computation and logical deduction coincide. Actually, Maude is a logic programming language in the general axiomatic sense made precise in [91]. Maude contains a *functional sublanguage* entirely similar to OBJ3 [46] as well as more general *system modules*, and also *object-oriented modules* that provide notational convenience for object-oriented applications but are reducible to system modules. The language's semantics is directly based on the model theory of rewriting logic and yields the desired semantic integration of concurrency with functional and object-oriented programming.

Rewriting logic is implicit in term rewriting systems but has passed for the most part unnoticed due to our overwhelming tendency to associate term rewriting with equational logic. Its proof theory exactly corresponds to concurrent computation, and the model theory proposed for it in this paper provides the general concept of concurrent system that we are seeking. The key ideas of this semantics were developed in December of 1987 [89] and were disseminated among a small group of researchers; they were first presented to a general audience in the San Miniato conference. This semantics generalizes the categorical semantics of Petri nets that we developed in joint work with Ugo Montanari [96,97].

The resulting notion of concurrent system specializes to a wide variety of concurrency models in a natural way. Section 5 discusses several of those specializations including: labelled transition systems, functional programming—in several flavors such as the general recursive functions of Herband–Gödel–Kleene, the lambda calculus, and algebraic data types—, Post systems and phrase-structure grammars, Petri nets, the chemical abstract machine [16] and CCS [99], as well as a new logical theory of concurrent object-oriented programming that encompasses Actors [3] and the UNITY model of computation [21] as particular cases. The models discussed are meant to illustrate quite different lines of work, but no claim to be exhaustive is made. Indeed, in this matter being exhaustive seems

hardly possible.

The paper is organized as follows. In the rest of this introduction we discuss informally the nature of rewriting logic as a logic of action, and motivate the intrinsically concurrent character of rewriting. We also show that the functional interpretation of rewriting corresponding to equational logic is too narrow and breaks down even for very simple examples, motivating in this way the need for rewriting logic. Section 2 presents the rules of deduction for conditional rewriting logic and makes precise the formal identification of concurrent rewriting with logical deduction. Section 3 proposes a model theoretic semantics for rewriting logic and gives a computational interpretation of such a semantics as the sought-after general notion of concurrent system. An initiality theorem and theorems proving the soundness and completeness of the logic with respect to this semantics are presented. A discussion of equationally definable subclasses of models is also given, yielding as special cases the logic of inequalities for preorder and poset models and the classical initial algebra semantics of equational logic. Section 4 discusses the Maude language, its syntax, its operational and denotational semantics, and its module structure. Section 5 contains a detailed discussion of how the concurrent rewriting model specializes to the various models already mentioned. Section 6 discusses related work by a variety of authors and presents some concluding remarks summarizing the main points of the paper.

The exposition assumes familiarity with elementary concepts of category theory such as category, functor, natural transformation, horizontal and vertical composition of natural transformations, and adjoint functors. MacLane's book [83] is an excellent source for all those notions.

1.1. Rewriting logic as a logic of action

The strong historical influence of mathematics on logic during the 19th and 20th centuries, while providing logic with high standards of rigor, has had the limiting effect of developing logic in a timeless, Platonic, direction that is not well suited for the dynamical nature of computation. For logic programming languages this is felt as an inadequacy to deal, within pure logic, with dynamic aspects of computation such as input–output, concurrency, or asserting new database facts. This applies to functional languages—based on a first order or higher order version of equational logic—and also to relational languages such as Prolog. This state of affairs poses an unhealthy *dualistic dilemma*, forcing one to choose between a clean, timeless, world of logic and the dirty material world of change and chaos.

This work proposes conditional rewriting logic as a way out of this dilemma which seems ideally suited for computational applications. This requires a reinterpretation of rewriting beyond equational logic, which has

been up to now its almost exclusive locus of activity. Indeed, equational logic is entirely Platonic and—in spite of noble and tenacious attempts to deal with dynamic aspects of computation within it, for example in functional programming—will just not do the job.

Although the rules of rewriting logic resemble those of equational logic, their meaning is very different; rewriting logic is a logic to reason about *change* in a concurrent system, *not about equality*. Each rewrite rule is a general pattern for a *basic action* that can occur concurrently with other actions in a concurrent system. Rewriting logic then allows us to reason about what other complex changes are possible in a system, given that changes corresponding to the basic actions axiomatized by the rules are possible. In this way, we can reason about concurrent programs in a logic *intrinsic* to their computations.

The *models* of rewriting logic are precisely *concurrent systems* in the intuitive sense of the word, i.e., they are machine-like entities whose state is distributed and can change by actions taking place simultaneously. Such models are formalized as categories with algebraic structure and this yields a general *triangular correspondence* between logic, concurrency and category theory by which ideas and methods can be transferred between these fields.

1.2. Rewriting is naturally concurrent

The literature on rewrite systems has for the most part formalized rewriting in terms of a sequential rewriting relation. However, the most natural understanding of rewriting is as an intrinsically concurrent activity. In fact, rewriting logic *identifies* concurrent rewriting with deduction. This section motivates the basic ideas with examples written in Maude [93,92], a language based on rewriting logic that contains OBJ [39,46] as its functional sublanguage.

The idea of concurrent rewriting is very simple. It is the idea of *equational simplification* that we are all familiar with from our secondary school days, *plus* the obvious remark that we can do many of those simplifications independently, i.e., in *parallel*. Consider for example the *functional modules* in Maude, written with an OBJ3 like syntax, in Fig. 1.

The first module defines the natural numbers in Peano notation, and the second defines a function to reverse a binary tree whose leaves are natural numbers. Each module begins with the keyword `fmod` followed by the module's name, and ends with the keyword `endfm`. A module contains sort and subsort declarations introduced by the keywords `sort` and `subsorts` stating the different sorts of data manipulated by the module and how those sorts are related. As in OBJ3, Maude's functional modules are based on a particularly flexible variant of equational logic, namely *order-sorted logic* [50], in which it is possible to declare one sort as a subsort of another;

```
fmod NAT is
    sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
endfm
```

```
fmod REVERSE is
    protecting NAT .
  subsorts Nat < Tree .
  op _^_ : Tree Tree -> Tree .
  op rev : Tree -> Tree .
  var N : Nat .
  vars T T' : Tree .
  eq rev(N) = N
  eq rev(T ^ T') = rev(T') ^ rev(T) .
endfm
```

Fig. 1.

for example, the declaration `Nat < Tree` states that every natural number is a tree consisting of a single node. Each of the functions provided by the module, as well as the sorts of their arguments and the sort of their result, is introduced using the keyword `op`. The syntax is user-definable, and permits specifying function symbols in "prefix", (in the `NAT` example the function `s_`), "infix" (`_+_`) or any "mixfix" combination as well as standard parenthesized notation (`rev`). Variables to be used for defining equations are declared with their corresponding sorts, and then equations are given; such equations provide the actual "code" of the module. The statement `protecting NAT` imports the `NAT` module and asserts that the natural numbers are not modified in the sense that no new data of sort `Nat` is added and different numbers are not identified by the new equations declared in the module `REVERSE`.

To compute with such modules, one performs equational simplification by using the equations from left to right until no more simplifications are possible. Note that this can be done *concurrently*, i.e., applying several equations at once, as in the example of Fig. 2, in which the places where the equations have been matched at each step are marked. Notice that the function symbol `_+_` was declared to be commutative by the attribute[1] `[comm]`. This not only asserts that the equation

```
N + M = M + N
```

is satisfied in the intended semantics, but it also means that when doing simplification we are allowed to apply the rules for addition not just to *terms*—in a purely syntactic way—but to *equivalence classes* of terms *modulo* the commutativity equation. In the example of Fig. 2, the rule

```
eq N + 0 = N .
```

is applied (modulo commutativity) with `0` both on the right *and* on the left.

[1] In Maude as in OBJ it is possible to declare several attributes of this kind for an operator, including also associativity and identity, and then do rewriting modulo such properties.

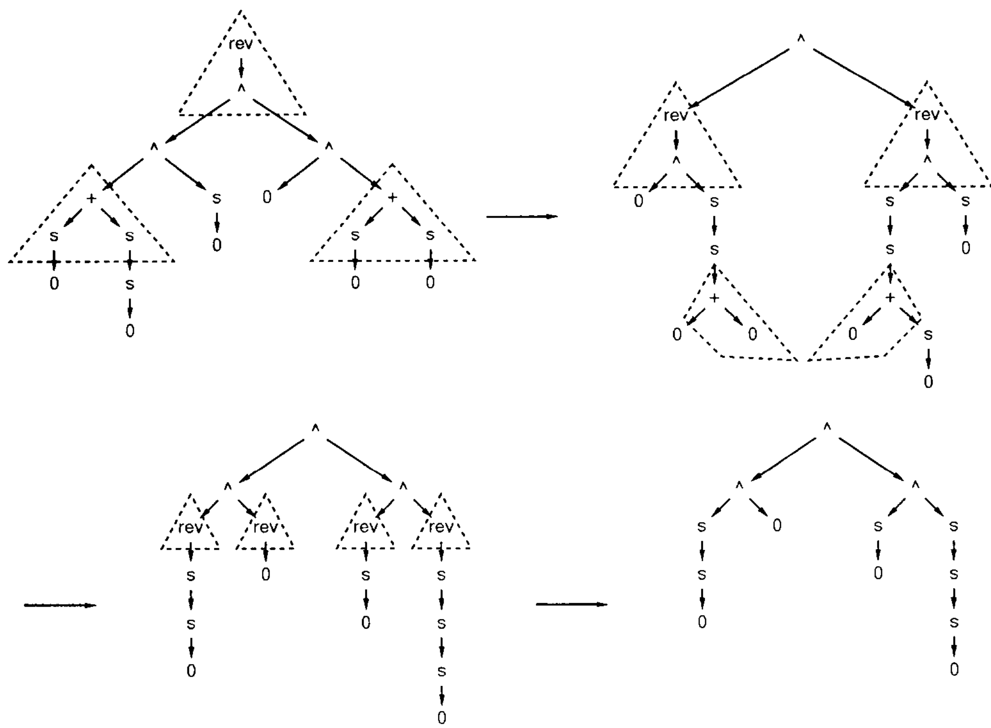


Fig. 2. Concurrent rewriting of a tree of numbers.

Two particularly appealing features of this style of concurrent programming are the *implicit* nature of the parallelism, which avoids having to program it explicitly, and the *logical* nature of the computation, which is just (directed) substitution of equals for equals.

The equations in the two modules above are Church–Rosser and terminating (more about this in Section 2.3). This implies that the *order* in which the rules are applied does not affect at all the final result which is uniquely determined by the original term being submitted for rewriting. Therefore, such modules are *functional* (thus the keyword `fmod`) and for them we can view the process of rewriting a term until no more rewritings are possible as the process of evaluating a functional expression to its result.

This functional interpretation of rewriting is of course intimately connected with equational logic. From the proof-theoretic point of view, we can view rewriting as a special efficient form of equational deduction which—when the rules are Church–Rosser and terminating—provides a decision procedure for equality. Model theoretically, to a given signature Σ of function symbols and set of rewrite rules R (perhaps modulo some equations E, such a commutativity, associativity, etc.) we can associate the class of all Σ-algebras that satisfy the equations R (and also the equations E, if we are

rewriting modulo E).

The *denotational semantics* of a Maude functional module such as the two ones discussed above is—exactly as in OBJ—given by the *initial algebra* in the class of all algebras satisfying the given (possibly conditional) equations. A nice property of the initial algebra is that in it the model theoretic and proof theoretic points of view come into full agreement, in the precise sense that two ground terms are provably equal iff they denote identical elements in the initial algebra, and for Church–Rosser and terminating rules we can decide this equality efficiently by rewriting. All this goes back to the original ADJ proposal [52] and to Goguen's work on the relationship between initial algebras and rewriting [44], and generalizes nicely to order-sorted equational logic, on which OBJ modules and Maude's functional modules are based [50,45,74].

Indeed, both in the applications to automated deduction that have been a constant source of stimulus for term rewriting techniques as well as in functional programming applications, equational logic has so entirely dominated our thinking that the overwhelming tendency has been to regard term rewriting as a technique for equational deduction.

A central aim of this paper is to suggest that, while the use of term rewriting within equational logic is all well and fine, many important applications—applications for which equational logic has no room at all—fit very well within a broader interpretation of rewriting as a logic in its own right.

1.3. Beyond equational logic

Consider for example the following Maude *system module*, which adds a nondeterministic choice operator to the natural numbers:

```
mod NAT-CHOICE is
  extending NAT .
  op _?_ : Nat Nat -> Nat .
  vars N M : Nat .
  rl N ? M => N .
  rl N ? M => M .
endm
```

The intuitive *operational behavior* of this module is quite clear. Natural number addition remains unchanged and is computed using the two rules in the `NAT` module. Notice that any occurrence of the choice operator in an expression can be eliminated by choosing either of the arguments. In the end, we can reduce any ground expression to a natural number in Peano notation. The *mathematical semantics* of the module is much less clear. If we adopt an initial algebra semantics, it follows by the rules of equational deduction with the above two equations that `N = M`, i.e., everything collapses

to one point and the module NAT is destroyed. To indicate that this is *not* the semantics intended, the keyword mod (instead of the previous fmod) has been used, indicating that the module *is not functional.* Similarly, the rewrite rules are introduced by a new keyword rl—instead of the usual eq—to suggest that they must be understood as "rules" and not as equations in the usual sense. Of course, at the operational level the equations introduced by the keyword eq in a functional module are also implemented as rewrite rules; the difference however lies in the *mathematical semantics* given to the module, which for modules like the one above should *not* be the initial algebra semantics.

But how can we make precise the intended meaning? Whatever it is, if we want to assert that NAT is not destroyed, as the behavior of the module shows, it is clear that the module cannot be regarded as a theory in equational logic, and that the associated initial algebra does not at all provide an adequate semantics; therefore, both must be abandoned. The proposal put forward in this paper is to seek a logic and a model theory that are the perfect match for this problem. For this solution to be in harmony with the old one, the new logic and the new model theory should in some sense *generalize* the old ones.

2. Rewriting logic

This section defines the syntax and proof theory of rewriting logic, and defines concurrent rewriting as deduction in such a logic. We first briefly recall some basic universal algebra needed in the exposition.

2.1. Basic universal algebra

For the sake of making the exposition simpler, the *unsorted* case is treated; the many-sorted and order-sorted cases can be given a similar treatment. Therefore, 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 $f_A : A^n \to A$ for each $f \in \Sigma_n$ with $n \in \mathbb{N}$. As usual (see, e.g., [95]) T_Σ denotes the Σ-algebra of ground Σ-terms, and $T_\Sigma(X)$ 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 (i.e., modulo provable equality using 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. We let $t =_E t'$ denote the congruence modulo E of two terms t, t', and $[t]_E$ or just $[t]$ denote the E-equivalence class of t.

For $t \in T_\Sigma(\{x_1, \ldots, x_n\})$, and $u_1, \ldots, u_n$, a sequence of terms, we denote by $t(u_1/x_1, \ldots, u_n/x_n)$ the term obtained from t by *simultaneously substituting*

u_i for x_i, $i = 1,\ldots,n$. To simplify notation, we will often denote a sequence of objects $a_1,\ldots,a_n$ by $\overline{a}$, or, if we want to emphasize the length of the sequence, by $\overline{a}^n$; also, in many contexts we will find it convenient to identify a sequence $a_1,\ldots,a_n$ of length n and its associated n-tuple $(a_1,\ldots,a_n)$. With this notation, $t(u_1/x_1,\ldots,u_n/x_n)$ can be abbreviated to $t(\overline{u}/\overline{x})$.

2.2. Deduction in rewrite theories

The syntax of rewriting logic is given by signatures. A *signature* is a pair (Σ, E) with Σ a ranked alphabet of function symbols[2] and E a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo a given set of equations E. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility, thanks to the "structural axioms" E, 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. The idea of rewriting in equivalence classes is well known (see, e.g., [61,33]).

Given a signature (Σ, E), the *sentences* that we consider are sequents of the form $[t]_E \rightarrow [t']_E$ with t, t' Σ-terms, where t and t' may possibly involve some variables from the countably infinite set $X = \{x_1,\ldots,x_n,\ldots\}$.

The notion of rewrite theory presented below is very general and expressive. In the first place, as already mentioned, it allows rewriting modulo "structural axioms" E, thus increasing the expressive power. In addition, it allows *conditional* rules of a very general form, where the conditions need not require equalities to hold but only the existence of rewritings among pairs of terms in the condition, which further increases the expressive power. Finally, it allows *labelling* of the rewrite rules; this is quite 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 (see Section 5). The categorical semantics of Section 3 will further clarify why this last extra generality is natural and desirable.

Definition 2.1. A (*labelled*) *rewrite theory*[3] $\mathcal{R}$ 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,

[2] As already mentioned, we could consider an *order-sorted* family Σ of function symbols; however, for the sake of a simpler exposition we treat the unsorted case.

[3] I consciously depart from the standard terminology, that would call $\mathcal{R}$ a *rewrite system*. The reason for this departure is quite specific. I want to keep the term "rewrite system" for the *models* of such a theory, which will be defined in Section 3 and which really are systems with a dynamic behavior. Strictly speaking, $\mathcal{R}$ is not a system; it is only a static, linguistic, *presentation* of a class of systems—including the initial and free systems that most directly formalize our dynamic intuitions about rewriting.

L is a set called the 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 nonempty sequence of pairs 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*[4]. For a rewrite rule $(r, ([t], [t'])([u_1], [v_1]) \ldots ([u_k], [v_k]))$ we use the notation

$$r : [t] \to [t'] \text{ if } [u_1] \to [v_1] \wedge \cdots \wedge [u_k] \to [v_k].$$

We call the part $[u_1] \to [v_1] \wedge \cdots \wedge [u_k] \to [v_k]$ the *condition* of the rule, and may abbreviate it with the letter C. To indicate that $\{x_1, \ldots, x_n\}$ is a set of variables occurring in either t, t', or C, we write[5] $r : [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$ *if* $C(x_1, \ldots, x_n)$, or in abbreviated notation $r : [t(\overline{x}^n)] \to [t'(\overline{x}^n)]$ *if* $C(\overline{x}^n)$.

Rules of the form $(r, ([t], [t']))$, i.e., with an empty condition, are called *unconditional rewrite rules*, and we use for them the notation

$$r : [t] \to [t'].$$

A rewrite theory where all the rules are unconditional is called an *unconditional rewrite theory.*

Example 2.2. The module `NAT-CHOICE` can be viewed as a rewrite theory with $\Sigma = \{0, s\_, \_+\_, \_?\_\}$, $E = \{N + M = M + N\}$ and with rules:

```
r1: N + 0 -> N
r2: (s N) + (s M) -> s s (N + M)
r3: N ? M -> N
r4: N ? M -> M
```

Alternatively, if we want to take Peano arithmetic as built in, we could add `r1` and `r2` to the set E of equations, and keep only `r3` and `r4` in R. In a similar vein, we can view `NAT` and `REVERSE` as rewrite theories with $E = \{N + M = M + N\}$ and with rules their corresponding equations, oriented from left to right; note that `REVERSE` is an order-sorted rewrite theory.

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

$$\mathcal{R} \vdash [t] \to [t']$$

[4]I.e., all rules are assumed *conditional* unless said otherwise.

[5]Note that, in general, the set $\{x_1, \ldots, x_n\}$ will depend on the representatives t, t', u_i, v_i chosen; therefore, we allow any possible such qualification with explicit variables.

if and only if $[t] \to [t']$ can be obtained by finite application of the following *rules of deduction*:

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

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

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

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

(3) *Replacement.* For each rewrite rule

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ if}$$

$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R,

$$\frac{\begin{array}{c}[w_1] \to [w'_1] \quad \dots \quad [w_n] \to [w'_n] \\ [u_1(\overline{w}/\overline{x})] \to [v_1(\overline{w}/\overline{x})] \quad \dots \quad [u_k(\overline{w}/\overline{x})] \to [v_k(\overline{w}/\overline{x})]\end{array}}{[t(\overline{w}/\overline{x})] \to [t'(\overline{w'}/\overline{x})]}.$$

That is, if for a substitution $x_i \mapsto w_i$, $1 \leqslant i \leqslant n$, we can deduce sequents

$$[u_j(\overline{w}/\overline{x})] \to [v_j(\overline{w}/\overline{x})], \quad 1 \leqslant j \leqslant k,$$

then, if in addition we can deduce $[w_i] \to [w'_i]$, $1 \leqslant i \leqslant n$, we are then allowed to deduce $[t(\overline{w}/\overline{x})] \to [t'(\overline{w'}/\overline{x})]$.

(4) *Transitivity.*

$$\frac{[t_1] \to [t_2] \quad [t_2] \to [t_3]}{[t_1] \to [t_3]}.$$

Note that for unconditional rules the rule of replacement specializes to the simpler:

(5) *Unconditional replacement.* For each

$$r : [t(x_1, \dots, x_n)] \to [t'(x_1, \dots, x_n)]$$

in R,

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

These rules are quite simple—yet fairly powerful. For example, they make unnecessary a general rule of substitution which can be proved to be a derived rule of (1)–(4) by a simple induction on the depth of derivations.

Proposition 2.3. *Any sequent derivable by adding to* (1)–(4) *the additional rule*

(6) *Substitution.*

$$\frac{[t(x_{i_1},\ldots,x_{i_n})] \to [t'(x_{i_1},\ldots,x_{i_n})] \quad [w_1] \to [w_1'] \quad \ldots \quad [w_n] \to [w_n']}{[t(w_1/x_{i_1},\ldots,w_n/x_{i_n})] \to [t'(w_1'/x_{i_1},\ldots,w_n'/x_{i_n})]}$$

can be derived using only the original rules (1)–(4).

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

(7) *Symmetry.*

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

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

For the moment we have only considered the rules of deduction for rewriting logic. Therefore, this logic might at first sight seem somewhat of an empty formal game. Such an impression would be mistaken. The importance of this logic will become apparent when we study its semantics in Section 3. However, a few remarks are in order at present. First, note that a sequent $[t] \to [t']$ should not be read as "$[t]$ *equals* $[t']$", but as "$[t]$ *becomes* $[t']$". Therefore, rewriting logic is a logic of *becoming* or *change*, not a logic of equality in a static Platonic sense. Adding the symmetry rule is 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 second related observation is that $[t]$ should not be understood as a *term* in the usual first-order logic sense, but as a *proposition*—built up using the *logical connectives* in Σ—that asserts being in a certain *state* having a certain *structure*. The rules of rewriting logic are therefore rules to reason about *change in a concurrent system*. They allow us to draw valid conclusions about the evolution of the system from certain basic types of change known to be possible thanks to the rules R.

2.3. Concurrent rewriting

We can now give a precise definition of concurrent rewriting. 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 2.4. Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, a (Σ, E)-sequent $[t] \rightarrow [t']$ is called:

- a 0-*step concurrent* $\mathcal{R}$-*rewrite* iff it can be derived from $\mathcal{R}$ by finite application of the rules (1) and (2) of rewriting deduction (in which case $[t]$ and $[t]'$ necessarily coincide);
- a *one-step concurrent* $\mathcal{R}$-*rewrite* iff it can be derived from $\mathcal{R}$ by finite application of the rules (1)–(3) (but *allowing all rules* (1)–(4) in the derivation of the sequents for the substitution instance of the condition whenever rule (3) is applied for r a conditional rule[6]) with at least one application of rule (3); if rule (3) was applied exactly once, we then say that the sequent is a one-step *sequential* $\mathcal{R}$-rewrite;
- a *concurrent* $\mathcal{R}$-*rewrite* (or just a *rewrite*) iff it can be derived from $\mathcal{R}$ by finite application of the rules (1)–(4).

We call the rewrite theory $\mathcal{R}$ *sequential* if all one-step $\mathcal{R}$-rewrites are necessarily sequential. A sequential rewrite theory $\mathcal{R}$ is in addition called *deterministic* if for each $[t]$ there is at most one one-step (necessarily sequential) rewrite $[t] \rightarrow [t']$. The notions of sequential and deterministic rewrite theory can be made relative to a given subset $S \subseteq T_{\Sigma,E}(X)$ by requiring that the corresponding property holds for each $[t']$ "reachable from S", i.e., for each $[t']$ such that for some $[t] \in S$ there is a concurrent $\mathcal{R}$-rewrite $[t] \rightarrow [t']$.

Example 2.5. All rewrite steps in Fig. 2 are one-step concurrent rewrites, but none are sequential. For example, the first such step can be obtained by first applying (unconditional) replacement twice for the second rule in NAT to 0-step sequents given by the substitutions $(\mathtt{N} \mapsto \mathtt{0}, \mathtt{M} \mapsto \mathtt{s0})$, and $(\mathtt{N} \mapsto \mathtt{0}, \mathtt{M} \mapsto \mathtt{0})$, to get sequents

$$[\mathtt{s0 + ss0}] \rightarrow [\mathtt{ss(0 + s0)}], \qquad [\mathtt{s0 + s0}] \rightarrow [\mathtt{ss(0 + 0)}];$$

then, applying congruence to each of these sequents and appropriate 0-step sequents to get sequents

$$[\mathtt{(s0 + ss0)\,\hat{}\,s0}] \rightarrow [\mathtt{ss(0 + s0)\,\hat{}\,s0}],$$

$$[\mathtt{0\,\hat{}\,(s0 + s0)}] \rightarrow [\mathtt{0\,\hat{}\,ss(0 + 0)}];$$

and finally, applying (unconditional) replacement to those two sequents for the second rule in the REVERSE module. Note that transitivity was never used.

[6] I.e., we view derivations for conditions as "scratchpad" deductions and do not include them in this classification. This of course means that a single rewriting step may implicitly contain many steps of deduction needed to establish its condition.

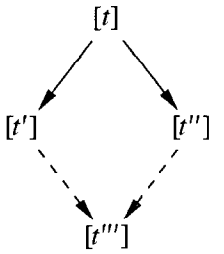


Fig. 3. The Church–Rosser property.

Traditionally, rewriting is defined by repeating one-step sequential rewrites. This makes the notion of a *sequence* of rewrites—an "operational" and even "sequential" notion indeed—the basic notion. By contrast, the basic view of rewriting stressed in this paper is that of a *logical deduction.* If we had followed the more traditional and operational view, we would have defined concurrent rewriting as a sequence of one-step concurrent rewrites. Implicit in such a view is the idea of an *observation* of "snapshots", but of course many different observations, all consistent with each other, may be possible. Section 3 will develop an algebraic theory of equivalence between such observations that can free us from an interleaving or snapshot view and bring us to the *paradiso* of "true concurrency". For the moment we state the following lemma, whose proof is postponed until Section 3.1, where a more precise formulation will be given (Lemma 3.6).

Lemma 2.6. *For each concurrent $\mathcal{R}$-rewrite $[t] \to [t']$, either $[t] = [t']$ or there is an $n \in \mathbb{N}$ and a chain of one-step (concurrent) rewrites*

$$[t] \to [t_1] \to \cdots \to [t_n] \to [t'].$$

We call such a chain a step sequence *for $[t] \to [t']$. In addition, we can always choose all the steps to be sequential; we then call it an* interleaving *or* firing *sequence for $[t] \to [t']$.*

We call a rewrite theory $\mathcal{R}$ *terminating* if there is no infinite chain of one-step rewrites (whether sequential or concurrent)

$$[t] \to [t_1] \to \cdots \to [t_n] \to \cdots.$$

We say that $[t']$ is an $\mathcal{R}$-*normal form* of $[t]$ if $[t] \to [t']$ is an $\mathcal{R}$-rewrite and there does not exist any one-step $\mathcal{R}$-rewrite of the form $[t'] \to [t'']$. If each $[t]$ has at least one normal form, we call the theory $\mathcal{R}$ *weakly terminating.*

We say that a rewrite theory $\mathcal{R}$ is *Church–Rosser* or *confluent* if given any two concurrent rewrites $[t] \to [t']$, $[t] \to [t'']$, there is a $[t''']$ and concurrent rewrites $[t'] \to [t''']$, $[t''] \to [t''']$. Likewise, we call $\mathcal{R}$ *ground*

Church–Rosser when the property is only asserted for equivalence classes of ground terms. This situation is shown in Fig. 3. Note that, by Lemma 2.6, this notion coincides with the usual one defined in terms of sequential rewriting. As expected, we then have

Theorem 2.7. *If $\mathcal{R}$ is Church–Rosser, then an equation $[t] \leftrightarrow [t']$ is provable from $\mathcal{R}$ by equational deduction, i.e., by using the rules* (1)–(4) *and rule* (7), *iff there is a $[t'']$ and $\mathcal{R}$-rewrites $[t] \to [t'']$, $[t'] \to [t'']$. In addition, if $\mathcal{R}$ is terminating, any $[t]$ has a unique normal form, called its* canonical form *and denoted $can_{\mathcal{R}}[t]$; under these conditions, an equation $[t] \leftrightarrow [t']$ is provable from $\mathcal{R}$ by equational deduction iff $can_{\mathcal{R}}[t] = can_{\mathcal{R}}[t']$.*

The case of Church–Rosser rewrite rules provides a straightforward specialization of the notion of conditional rewrite rule that we have been considering to the more traditional notion in which the condition is a finite conjunction of equalities. The specialization is as follows. Consider a Church–Rosser rewrite theory $\mathcal{R}$ such that all the rules are of the form

$$\begin{aligned} r : [t(\overline{x})] \to [t'(\overline{x})] \quad & \textit{if} \\ & [u_1(\overline{x})] \to [y_1] \wedge [v_1(\overline{x})] \to [y_1] \wedge \cdots \wedge [u_k(\overline{x})] \to [y_k] \\ & \quad \wedge [v_k(\overline{x})] \to [y_k]. \end{aligned} \tag{†}$$

with the y_j and the x_i pairwise disjoint. Then, application of the replacement rule (3) is possible relative to instances

$$[w_1] \to [w_1'] \quad \ldots \quad [w_n] \to [w_n']$$

of the variables $x_1, \ldots, x_n$ if and only if we can find terms $p_1, \ldots, p_k$ such that

$$\begin{aligned} & [u_1(\overline{w}/\overline{x})] \to [p_1] \wedge [v_1(\overline{w}/\overline{x})] \to [p_1] \\ & \quad \wedge \cdots \wedge [u_k(\overline{w}/\overline{x})] \to [p_k] \wedge [v_k(\overline{w}/\overline{x})] \to [p_k], \end{aligned}$$

and by Theorem 2.7 this holds iff

$$\mathcal{R} \vdash [u_1(\overline{w}/\overline{x})] \leftrightarrow [v_1(\overline{w}/\overline{x})] \wedge \cdots \wedge [u_k(\overline{w}/\overline{x})] \leftrightarrow [v_k(\overline{w}/\overline{x})].$$

Therefore, for a Church–Rosser rewrite theory $\mathcal{R}$ with rules of the form (†) above, we can express the rules with the alternative notation

$$\begin{aligned} & r : [t(\overline{x})] \to [t'(\overline{x})] \ \textit{if} \\ & \quad [u_1(\overline{x})] \leftrightarrow [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \leftrightarrow [v_k(\overline{x})], \end{aligned}$$

which is fully justified, since checking the condition is in this case equivalent to checking provable equality from $\mathcal{R}$ under the rules of equational logic. If, in addition, the theory $\mathcal{R}$ is terminating, we can implement application

of the replacement rule by reducing the two sides of each condition to canonical form, i.e., by checking the identities

$$can_{\mathcal{R}}[u_1(\overline{w}/\overline{x})] = can_{\mathcal{R}}[v_1(\overline{w}/\overline{x})]$$
$$\wedge \cdots \wedge can_{\mathcal{R}}[u_k(\overline{w}/\overline{x})] = can_{\mathcal{R}}[v_k(\overline{w}/\overline{x})]$$

which is of course the nicest possible situation from an implementation point of view. Careful examination of the above argument (and some of the reasoning that goes into the proof of Theorem 2.7) shows that all we have really used is something much weaker than requiring $\mathcal{R}$ to be Church–Rosser. All we need to require is that the Church–Rosser property holds for all terms p that can be proved equal by the rules of equational logic (using $\mathcal{R}$) to a substitution instance of a term $[u_j(\overline{x})]$ or a term $[v_j(\overline{x})]$ appearing in a condition of a rule (†) in $\mathcal{R}$.

Before finishing this section, the following four topics related to concurrent rewriting should be mentioned briefly.

Tree vs. graph

Instead of rewriting terms, we can rewrite graphs which represent terms in a more economic way by allowing *sharing*. This is an active area of research and a quite practical matter. Besides implementation aspects, there are also interesting and subtle semantic issues. For example, the disjoint union of two terminating rewriting systems need not be terminating [123], but it is terminating if the rules are implemented as graph rewriting rules with maximal sharing in the righthand sides [109]. An issue worth remarking is that labelled graphs can be characterized as elements of an algebraic data type axiomatized by equations [14] in shuch a way that graph rewriting becomes rewriting *modulo* those equations; therefore, it appears that graph rewriting can be reduced to a special case of term rewriting modulo equations. Another issue worth exploring is the relationship between the categorical semantics given in this paper and the various categorical approaches to graph rewriting [35,113,71,72]. Finally, the recent approach to actors using graph grammars [64,37] should be compared with the logical theory of concurrent objects sketched in Section 5.3.4 and further developed in [92]. A detailed discussion of these and other issues concerning the relationship between tree and graph rewriting will have to wait for a future occasion.

Effectiveness

For rewriting to be a *model of computation* we cannot accept just any collection E of axioms. We must demand the existence of an *algorithm* that, given a finite rewrite theory $\mathcal{R}$, will generate for us all possible one-step rewrites using $\mathcal{R}$. This can take a variety of forms. A very good one is to assume a *matching algorithm* modulo the axioms, and to develop techniques

to study properties such as the Church–Rosser property in terms of it; see [68] for an elegant general theory of this kind. A subtle point is that in a matching algorithm the left-hand side is assumed to be at the top of the redex and this requires that some additional rules, called *extensions*, be added in order to actually get the effect of rewriting in equivalence classes. For axioms in which all equivalence classes are finite, another possibility is searching the equivalence class. It is also possible to have axioms that, viewed as rules, are Church–Rosser, and such that rewriting modulo them can be accomplished by reduction to normal form using the axioms; one instance of this case appears in Example 2.2 for the variant in which the rules for addition are adopted as axioms.

Strategies and fairness

The role of strategies is to cut down the number of rewrite sequences that must be considered. This is sometimes done for reasons of efficiency, because a given strategy is known to yield sequences that terminate faster, or to ensure that a normal form will be found if one exists, as in the λ-calculus. Of course, if the rules are Church–Rosser and terminating only efficiency is affected and any fair strategy will do. There is a large literature treating *sequential* strategies (see, e.g., [104,12]) and a lot of activity in the area of *strictness analysis* to allow compilers for functional programs to automatically detect good strategies. Parallel strategies have received less attention; they are important to improve the performance of parallel computations and permit achieving balanced tradeoffs between space and time for such computations. The paper [47] proposes a variety of parallel strategies that can be useful for machines implementing concurrent rewriting, like the rewrite rule machine (RRM). The issue of fairness for term rewriting has already received attention by several authors, including Francez and Porat [110] and Tison [122]; however, this is a rather new area where more theoretical developments are needed. Fairness issues will not be covered in this paper; they will be treated elsewhere.

Parallel architecture

One of the most valuable uses of an abstract model of computation is as a guide to its physical realization in a new architecture. In fact, the limitations of a number of current parallel machines derive in part from their being more or less subservient to the von Neumann model of computation. Basing an architectural design as directly as possible on a fully parallel model of computation may require a fair amount of architectural innovation, but may also yield solutions with better performance, more scalable and easier to program than those offered by more conservative approaches. Concurrent rewriting—in a variety of guises, and often called *graph reduction* in this

context—has been recognized as a promising model of this kind by a variety of researchers, specially in connection with the implementation of functional languages (see for example [69] for a collection of research articles). Some of the work, including the G-machine [66], the categorical abstract machine [27], and the Standard ML compiler [8], has been concerned with sequential implementations of reduction, including hardware design [73], and this has led to a variety of efficient sequential implementations of functional languages. Other work has explored parallel implementations. Work within the dataflow tradition includes the Alice project [56] and its Flagship continuation [124]. Two difficulties associated with standard dataflow machines are the restriction of reduction to the leaves of the tree, and the latency problems in the traffic between processors and memory which makes them hard to scale up. The Grip machine [67] is a coarse grain parallel graph reduction machine. The rewrite rule machine [47,49,81,5,6,42] is a massively parallel architecture that exploits the parallelism of concurrent rewriting at all levels. Fine grain parallelism is exploited at the chip level operating in SIMD mode, and coarse grain parallelism is exploited at the network level where thousands of chips can cooperate in performing complex computations working in MIMD mode.

3. Semantics

As such, a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ is a *static description* of what a system can do. The *meaning* of the theory should be given by a model of its actual *behavior*. Since our approach has emphasized that the *concurrent computations* are nothing other than *deductions* in rewriting logic, it is natural, in the spirit of initial model semantics, to construct a model with a precise algebraic structure which is the most informative model possible in the sense that, in it, behavior exactly corresponds to deduction. The procedure for building such a model is straightforward. As in the Curry–Howard correspondence—generalized here to a correspondence between proofs and morphisms in a category—we can just read the rules (1)–(4) of rewriting logic as *rules of generation*. As already hinted at in Section 2.3, the question of when are two proofs (and therefore two computations) equal still remains. This is answered by postulating a set of equations that are natural and express intuitive identities between computations. What they allow us to get at, in a precise axiomatic way, is the intuitive notion of "true concurrency" by abstracting away from particular "snapshot" descriptions. The model obtained in this way is just the initial (or free when we allow variables) model of a wide class of models containing the functional models of classical initial algebra semantics as a very particular case. We also explore this general class of models—and some of its important subclasses—and give

them a computational interpretation as concurrent systems of a very general nature. The relationships between rewriting logic and equational logic are also clarified and are formalized by means of a map of logics in the sense of [91].

3.1. The $\mathcal{T}_{\mathcal{R}}(X)$ construction

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 the equivalence classes of terms $[t] \in T_{\Sigma,E}(X)$ and whose morphisms are equivalence classes of 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 are obtained from the rules of deduction (1)–(4) by decorating the sequents appearing in each rule with appropriate proof terms. Note that in the rest of this paper 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 $[t] \in T_{\Sigma,E}(X)$,

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

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

$$\frac{\alpha_1 : [t_1] \to [t_1'] \quad \ldots \quad \alpha_n : [t_n] \to [t_n']}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \to [f(t_1', \ldots, t_n')]}.$$

(3) *Replacement.* For each rewrite rule

$$r : [t(\overline{x})] \to [t'(\overline{x})] \ \textit{if}$$

$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R,

$$\frac{\begin{array}{c}\alpha_1 : [w_1] \to [w_1'] \quad \ldots \quad \alpha_n : [w_n] \to [w_n'] \\ \beta_1 : [u_1(\overline{w}/\overline{x})] \to [v_1(\overline{w}/\overline{x})] \quad \ldots \quad \beta_k : [u_k(\overline{w}/\overline{x})] \to [v_k(\overline{w}/\overline{x})]\end{array}}{r(\overline{\alpha}^n, \overline{\beta}^k) : [t(\overline{w}/\overline{x})] \to [t'(\overline{w'}/\overline{x})]}.$$

(4) *Composition.*

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

For the case of equational logic we can also define a similar model as a category $T_{\mathcal{R}}^{\leftrightarrow}(X)$ (actually a *groupoid*[7]) by using the rule of symmetry to generate additional terms:

(5) *Inversion.*

$$\frac{\alpha : [t_1] \to [t_2]}{\alpha^{-1} : [t_2] \to [t_1]}.$$

Convention and warning. In the case when the same label r appears in two different rules of R, the "proof terms" $r(\overline{\alpha}, \overline{\beta})$ can be *ambiguous*, either because different sources or targets are assigned to the same term by different rules, or because of the even more ambiguous situation whereby the same source and target are assigned to the same proof term $r(\overline{\alpha}, \overline{\beta})$ by two different rules, which can happen when the set E of axioms is nonempty. We will always assume that such ambiguity problems *have been resolved* by disambiguating the label r in the proof terms $r(\overline{\alpha}, \overline{\beta})$. With this understanding, the simpler notation $r(\overline{\alpha}, \overline{\beta})$ is adopted 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 $\mathcal{P}_{\mathcal{R}}(X)$ defined by the generation rules (1)–(4) above. Specifically, this structure is a graph[8] with set of nodes $T_{\Sigma,E}(X)$ and with set of arrows the set $\mathcal{P}_{\mathcal{R}}(X)$ of all proof terms generated by the above rules, with source and target maps

$$\partial_0, \partial_1 : \mathcal{P}_{\mathcal{R}}(X) \to T_{\Sigma,E}(X)$$

as specified by the rules; in particular, rule (1) requires each node $[t]$ to be also an arrow $[t] : [t] \to [t]$, i.e., it makes the graph reflexive[9]. In addition, by rule (2), the set $\mathcal{P}_{\mathcal{R}}(X)$ has a Σ-algebra structure such that the source and target maps ∂_0 and ∂_1 are Σ-homomorphisms; in other words, we have

[7] A category C is called a *groupoid* iff any morphism $f : A \to B$ in C has an inverse morphism $f^{-1} : B \to A$ such that $f ; f^{-1} = 1_A$, and $f^{-1} ; f = 1_B$.

[8] By a *graph* G we will always mean a directed graph, i.e., a structure $G = (\partial_0, \partial_1 : Arrows \to Nodes)$. A *graph homomorphism* $(f, g) : (\partial_0, \partial_1 : Arrows \to Nodes) \to (\partial_0', \partial_1' : Arrows' \to Nodes')$ is then defined as a pair of functions $f : Arrows \to Arrows'$, $g : Nodes \to Nodes'$ such that $f ; \partial_i' = \partial_i ; g$, $i = 0, 1$.

[9] A *reflexive graph* is a graph $G = (\partial_0, \partial_1 : Arrows \to Nodes)$ together with a function $j : Nodes \to Arrows$ such that $j ; \partial_0 = j ; \partial_1 = 1_{Nodes}$.

a Σ-algebra structure *on a graph* instead than just on a set. Rule (4) defines an operation on composable[10] pairs of arrows:

$$\_;\_ : Composable(\mathcal{P}_{\mathcal{R}}(X)) \to \mathcal{P}_{\mathcal{R}}(X)$$

with $\partial_0(\alpha;\beta) = \partial_0(\alpha)$ and $\partial_1(\alpha;\beta) = \partial_1(\beta)$. Finally, rule (3) defines for each rewrite rule

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ if}$$
$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R, an operation[11]

$$r(\_,\_) : \{(\overline{\alpha}^n, \overline{\beta}^k) \in \mathcal{P}_{\mathcal{R}}(X)^{n+k} \mid \partial_0^k(\overline{\beta}^k) = \overline{[u(\partial_0^n(\overline{\alpha}^n)/\overline{x})]}^k$$
$$\wedge\, \partial_1^k(\overline{\beta}^k) = \overline{[v(\partial_0^n(\overline{\alpha}^n)/\overline{x})]}^k\} \to \mathcal{P}_{\mathcal{R}}(X)$$

with source and target requirements as specified by rule (3). The structure of $\mathcal{P}_{\mathcal{R}}(X)$ just made explicit is an instance of the following general concept.

Definition 3.1. Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, an $\mathcal{R}$-*presystem* is a reflexive graph $G = (\partial_0, \partial_1 : Arrows \to Nodes, j)$ together with:

(i) a Σ-algebra structure on G such that the Σ-algebra *Nodes* of nodes satisfies the equations E;

(ii) an operation

$$\_;\_ : Composable(G) \to Arrows$$

with $\partial_0(a;b) = \partial_0(a)$ and $\partial_1(a;b) = \partial_1(b)$;

(iii) for each rewrite rule

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ if}$$
$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R, an operation

$$r(\_,\_) : \{(\overline{a}^n, \overline{b}^k) \in Arrows^{n+k} \mid \partial_0^k(\overline{b}^k) = \overline{u_{Nodes}(\partial_0^n(\overline{a}^n))}^k$$
$$\wedge\, \partial_1^k(\overline{b}^k) = \overline{v_{Nodes}(\partial_0^n(\overline{a}^n))}^k\} \to Arrows$$

[10] Given a graph $G = (\partial_0, \partial_1 : Arrows \to Nodes)$, the set of its *composable pairs of arrows* is the set $Composable(G) = \{(a, a') \in Arrows^2 \mid \partial_1(a) = \partial_0(a')\}$.

[11] Recall again that we implicitly assume that the operator $r(\_,\_)$ has been disambiguated if this is necessary in order to keep the proof terms unambiguous.

(where for $t(x_1, \ldots, x_n)$ a Σ-term, t_{Nodes} denotes the n-ary derived operation on nodes associated to the term t) such that

$$r(\overline{a}^n, \overline{b}^k) : t_{Nodes}(\partial_0^n(\overline{a}^n)) \to t'_{Nodes}(\partial_1^n(\overline{a}^n)).$$

Given two $\mathcal{R}$-presystems G and G', an *$\mathcal{R}$-prehomomorphism* is a graph homomorphism that preserves all the operations, i.e., the operations in Σ, the operation $\_;\_$ and the operations $r(\_,\_)$ for $r \in R$. This defines a category $\underline{\mathcal{R}\text{-}PreSys}$ in the obvious way.

The definition of $\mathcal{R}$-presystem just given is "essentially algebraic". The style of universal algebra involved is slightly more subtle than standard universal algebra (because operations such as $\_;\_$ or the operations $r(\_,\_)$ for $r \in R$ are only defined in a *subset* of tuples of elements determined by some *equations*) but well known. It can be formalized by means of "essentially algebraic theories" [38,115] or "sketches" [13].

Note that in the $\mathcal{P}_{\mathcal{R}}(X)$ construction the set X of variables is actually a parameter, and we need not assume X to be fixed and countable. The following proposition can be proved easily by induction on the depth of proof terms.

Proposition 3.2. *For $\mathcal{R}$ a rewrite theory, given a set X and a function $F : X \to Nodes$ to the set Nodes of nodes of an $\mathcal{R}$-presystem G, there exists a unique $\mathcal{R}$-prehomomorphism $F^\sharp : \mathcal{P}_{\mathcal{R}}(X) \to G$ whose map of nodes, say $F^\sharp_{nodes}$, extends F, i.e., $F^\sharp_{nodes}([x]) = F(x)$ for each $x \in X$.*

In the language of adjoint functors [83], this proposition can be rephrased by saying that the forgetful functor

$$Nodes : \underline{\mathcal{R}\text{-}PreSys} \to \underline{Set}$$

sending each $\mathcal{R}$-presystem to its set of nodes has a left adjoint.

$\mathcal{P}_{\mathcal{R}}(X)$ provides an algebra of proofs for the rewriting logic deductions of a rewrite theory $\mathcal{R}$. However, $\mathcal{P}_{\mathcal{R}}(X)$ is of course a completely *syntactic* structure, namely a term algebra. As a consequence, it makes too many distinctions. Such distinctions are forced upon us by syntax but are clearly irrelevant for a variety of purposes. From a proof-theoretic point of view we may want to characterize when two proofs are "essentially the same", despite being represented by different proof terms due to the rigidity of syntax. However, since proofs in rewriting logic correspond to concurrent computations, a computational perspective affords a different reformulation of the same question, namely, when are two concurrent computations essentially the same? This question has been addressed by advocates of "true

concurrency" in a variety of contexts. Typically, some partial order representation of a computation is proposed as the way of abstracting a class of essentially equivalent descriptions. Our answer here is completely axiomatic; it generalizes the equational axiomatization of Petri net computations developed in joint work with Pierpaolo Degano and Ugo Montanari[12] in [31,32] and is similar in style to the elegant treatment of true concurrency by equations between proofs given by Boudol and Castellani [20]. The answer is provided by the system $\mathcal{T}_{\mathcal{R}}(X)$ which is defined as the quotient presystem of $\mathcal{P}_{\mathcal{R}}(X)$ modulo the following equations (in the expressions appearing in the equations, when compositions of morphisms are involved, we always implicitly assume that the corresponding domains and codomains match):

(1) *Category.*

(a) Associativity. For all α, β, γ,

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

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

$$\alpha ; [t'] = \alpha, \qquad [t] ; \alpha = \alpha.$$

(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 $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ an axiom in E, for all $\alpha_1, \ldots, \alpha_n$,

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

(4) *Decomposition.* For each rewrite rule

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ if}$$

$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R,

[12] Indeed, for Petri nets (see Section 5.3.1) this kind of axiomatic approach has been shown to yield the partial order approach as a particular case in [31,32]. This provides additional evidence to support the claim that an axiomatic treatment of this kind is quite general and flexible. Indeed, not only can it be applied to a wide variety of models of concurrency (see Section 5) but it also makes very easy the exploration of different equivalences by adding or subtracting axioms, as done in [31,32] for Petri nets. Therefore, the equations below—although quite natural—should not be seen as the only possible equivalence that can be defined.

$$\frac{\begin{array}{cccc}\alpha_1 : [w_1] \to [w_1'] & \dots & \alpha_n : [w_n] \to [w_n'] \\ \beta_1 : [u_1(\overline{w}/\overline{x})] \to [v_1(\overline{w}/\overline{x})] & \dots & \beta_k : [u_k(\overline{w}/\overline{x})] \to [v_k(\overline{w}/\overline{x})]\end{array}}{r(\overline{\alpha}^n, \overline{\beta}^k) = r(\overline{[w]}^n, \overline{\beta}^k) ; t'(\overline{\alpha}^n)}.$$

(5) *Exchange.* For each rewrite rule

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ if}$$

$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R,

$$\frac{\begin{array}{cccc}\alpha_1 : [w_1] \to [w_1'] & \dots & \alpha_n : [w_n] \to [w_n'] \\ \beta_1 : [u_1(\overline{w}/\overline{x})] \to [v_1(\overline{w}/\overline{x})] & \dots & \beta_k : [u_k(\overline{w}/\overline{x})] \to [v_k(\overline{w}/\overline{x})] \\ \beta_1' : [u_1(\overline{w'}/\overline{x})] \to [v_1(\overline{w'}/\overline{x})] & \dots & \beta_k' : [u_k(\overline{w'}/\overline{x})] \to [v_k(\overline{w'}/\overline{x})] \\ \beta_1 ; v_1(\overline{\alpha}) = u_1(\overline{\alpha}) ; \beta_1' & \dots & \beta_k ; v_k(\overline{\alpha}) = u_k(\overline{\alpha}) ; \beta_k'\end{array}}{r(\overline{[w]}^n, \overline{\beta}^k) ; t'(\overline{\alpha}) = t(\overline{\alpha}) ; r(\overline{[w']}^n, \overline{\beta'}^k)}.$$

Similarly, the groupoid $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}(X)$ is obtained by identifying the terms generated by rules (1)–(5) modulo the above equations plus the additional:

(6) *Inverse.* For any $\alpha : [t] \to [t']$ in $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}(X)$,

$$\alpha ; \alpha^{-1} = [t], \qquad \alpha^{-1} ; \alpha = [t'].$$

Note again 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 notations $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}$, respectively.

Some comments can help clarify the intended meaning behind the above equations. The equations in (1) make $\mathcal{T}_{\mathcal{R}}(X)$ into a category, the equations in (2) give it a Σ-algebra structure *as a category*, i.e., each $f \in \Sigma_n$ determines a functor

$$f : \mathcal{T}_{\mathcal{R}}(X)^n \to \mathcal{T}_{\mathcal{R}}(X)$$

and the equations in (3) force such Σ-algebra structure to satisfy the equations E. Note that it follows easily (by induction) from the functoriality of the basic operations and from the satisfaction of the equations E that each $[t(x_1, \dots, x_n)]$ defines a functor

$$[t(x_1, \dots, x_n)] : \mathcal{T}_{\mathcal{R}}(X)^n \to \mathcal{T}_{\mathcal{R}}(X)$$

in the obvious way; this is just what the algebraic notion of a *derived operation* means in this context.

The decomposition law states that any rewriting of the form $r(\overline{\alpha}, \overline{\beta})$—which represents the *simultaneous* rewriting of the term at the top using rule r (once proofs $\overline{\beta}$ for the conditions have been provided) *and* "below",

i.e., in the subterms matched by the rule—is equivalent to the sequential composition $r(\overline{[w]},\overline{\beta})\,;t'(\overline{\alpha})$ corresponding to first rewriting on top with r and then below on the matched subterms.

The intuitive meaning of the exchange law is that—under certain conditions—rewriting at the top by means of a rule r and rewriting "below", i.e., in the subterms matched by the rule, are independent processes and therefore can be done in any order. Since the exchange law is somewhat complicated by the conditions in the conditional rule, it is simpler to first make precise the meaning of this law for *unconditional* rules, for which it can be combined together with the decomposition law into the single law:

(7) *Unconditional exchange.* For each

$$r : [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

in R,

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

Therefore, in the unconditional case rewritings on top using r and rewritings "below" can always be exchanged. Since $[t(x_1,\ldots,x_n)]$ and $[t'(x_1,\ldots,x_n)]$ can be regarded as functors $\mathcal{T}_{\mathcal{R}}(X)^n \to \mathcal{T}_{\mathcal{R}}(X)$, the exchange law asserts that r is a natural transformation [83], i.e., we have the following lemma.

Lemma 3.3. *For each* $r : [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$ *in* R, *the family of morphisms*

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

is a natural transformation

$$r : [t(x_1,\ldots,x_n)] \Rightarrow [t'(x_1,\ldots,x_n)]$$

between the functors

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

In the conditional case, the exchange law is subject to certain requirements. Such requirements essentially mean that the proofs of the conditions "before" and "after" rewriting below with the α's should be equivalent, i.e., that we have in essence provided the same proof for each of the conditions. This is important, because for a non-Church–Rosser rewrite theory it may even be *impossible* to prove the conditions *after* the term has been rewritten below by the α's, and therefore exchanging rewritings on top and "below" would in this case be illegal and ill defined.

Now that the intuitive meaning of the exchange law has been explained, we can make precise its mathematical meaning. For this, we need the notion of a *subequalizer* [78]; this notion generalizes that of an equalizer of two functors, by requiring a natural transformation instead of just an identity of functors.

Definition 3.4. Given functors $F, G : \mathcal{A} \to \mathcal{B}$, the *subequalizer* of F and G is a category $Subeq(F, G)$ together with a functor

$$J : Subeq(F, G) \to \mathcal{A}$$

and a natural transformation[13] $\alpha : J * F \Rightarrow J * G$ satisfying the following universal property: Given a functor $H : \mathcal{C} \to \mathcal{A}$ and a natural transformation $\beta : H * F \Rightarrow H * G$, there exists a unique functor $(H, \beta) : \mathcal{C} \to Subeq(F, G)$ such that

$$(H, \beta) * J = H \quad \text{and} \quad (H, \beta) * \alpha = \beta.$$

Similarly, given a family of pairs of functors $\{F_i, G_i : \mathcal{A} \to \mathcal{B}_i \mid i \in I\}$, the (simultaneous) *subequalizer* of this family is a category $Subeq((F_i, G_i)_{i \in I})$ together with a functor

$$J : Subeq((F_i, G_i)_{i \in I}) \to \mathcal{A}$$

and a family of natural transformations $\{\alpha_i : J * F_i \Rightarrow J * G_i \mid i \in I\}$ satisfying the following universal property: Given a functor $H : \mathcal{C} \to \mathcal{A}$ and a family of natural transformations $\{\beta_i : H * F_i \Rightarrow H * G_i \mid i \in I\}$, there exists a unique functor $(H, \{\beta_i\}_{i \in I}) : \mathcal{C} \to Subeq((F_i, G_i)_{i \in I})$ such that

$$(H, \{\beta_i\}_{i \in I}) * J = H \quad \text{and} \quad (H, \{\beta_i\}_{i \in I}) * \alpha_i = \beta_i \quad (i \in I).$$

The construction of $Subeq((F_i, G_i)_{i \in I})$ is quite simple. Its objects are pairs $(A, \{b_i\}_{i \in I})$ with A an object in $\mathcal{A}$ and $b_i : F_i(A) \to G_i(A)$ a morphism in $\mathcal{B}_i$. Morphisms

$$a : (A, \{b_i\}_{i \in I}) \to (A', \{b'_i\}_{i \in I})$$

are morphisms $a : A \to A'$ in $\mathcal{A}$ such that for each $i \in I$, $b_i ; G_i(a) = F_i(a) ; b'_i$. The functor J is just projection into the first component. The natural transformation α_j is defined by

$$\alpha_j(A, \{b_i\}_{i \in I}) = b_j.$$

[13]Note that we use diagrammatic order for the composition $J * F$ of two functors. More generally, we will also use diagrammatic order for the *horizontal*, $\alpha * \beta$, and *vertical*, $\gamma ; \delta$, composition of natural transformations (see [83]).

For our present purposes, the subequalizers of interest are of the form

$$Subeq(([u_j(\overline{x})], [v_j(\overline{x})])_{1 \leqslant j \leqslant k})$$

associated to rewrite rules[14]

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ if}$$
$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R. An object of such a subequalizer is a pair $(\overline{[w]}^n, \overline{[\beta]}^k)$ with $\overline{[w]} \in T_{\Sigma,E}(X)^n$ and with

$$[\beta_i] : [u_i(\overline{w}/\overline{x})] \to [v_i(\overline{w}/\overline{x})], \quad 1 \leqslant i \leqslant k,$$

equivalence classes of proof terms modulo the equations (1)–(5), i.e., morphisms of $\mathcal{T}_{\mathcal{R}}(X)$. Therefore, the conditions

$$\beta_1 ; v_1(\overline{\alpha}) = u_1(\overline{\alpha}) ; \beta'_1, \ldots, \beta_k ; v_k(\overline{\alpha}) = u_k(\overline{\alpha}) ; \beta'_k$$

in the exchange law just state that $\overline{\alpha}^n : (\overline{[w]}^n, \overline{[\beta]}^k) \to (\overline{[w']}^n, \overline{[\beta']}^k)$ is a morphism in

$$Subeq(([u_j(\overline{x})], [v_j(\overline{x})])_{1 \leqslant j \leqslant k})$$

and the equation in the conclusion just states that r is a natural transformation. Hence, we have the following result.

Lemma 3.5. *For each rewrite rule*

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ if}$$
$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R, the family of morphisms

$$\{r(\overline{[w]}^n, \overline{\beta}^k) : [t(\overline{w}/\overline{x})] \to [t'(\overline{w}/\overline{x})] \mid$$
$$(\overline{[w]}^n, \overline{[\beta]}^k) \in Subeq(([u_j(\overline{x})], [v_j(\overline{x})])_{1 \leqslant j \leqslant k})\}$$

is a natural transformation

$$r : J * [t(x_1, \ldots, x_n)] \Rightarrow J * [t'(x_1, \ldots, x_n)]$$

where $J : Subeq(([u_j(\overline{x})], [v_j(\overline{x})])_{1 \leqslant j \leqslant k}) \to \mathcal{T}_{\mathcal{R}}(X)^n$ *is the subequalizer functor.*

[14] If the condition of such a rule is abbreviated to C, then we will use the notation $Subeq(C)$ for the corresponding subequalizer.

The $\mathcal{T}_{\mathcal{R}}$ construction just explained is very general and yields other well known constructions as particular cases. For example, for $\mathcal{R}$ a labelled transition system (see Section 5.1.1) $\mathcal{T}_{\mathcal{R}}$ is the well known *path category* associated to the transition system $\mathcal{R}$. For the case of Petri nets (see Section 5.3.1) the $\mathcal{T}_{\mathcal{R}}$ construction specializes to the construction of the monoidal category $\mathcal{T}[\mathcal{R}]$ associated to a Petri net $\mathcal{R}$ in joint work with Ugo Montanari [96,97].

What the decomposition and exchange equations provide in general is a way of *abstracting* a rewriting computation by considering immaterial the order in which rewrites are performed "above" and "below" in the term (provided that the appropriate requirements are met); further abstraction among proof representations is of course provided by the functoriality equations and by the equations E. The equations (1)–(5) provide in a sense the *most abstract* view of the computations of the rewrite theory $\mathcal{R}$ that can reasonably be defined. In particular, we can prove that all proof terms are equivalent to step sequences and also to interleaving sequences.

Lemma 3.6. *For each* $[\alpha] : [t] \to [t']$ *in* $\mathcal{T}_{\mathcal{R}}(X)$, *either* $[t] = [t']$ *and* $[\alpha] = [[t]]$, *or there is an* $n \in \mathbb{N}$ *and a chain of morphisms* $[\alpha_i]$, $0 \leqslant i \leqslant n$ *whose terms* α_i *describe one-step (concurrent) rewrites*

$$[t] \xrightarrow{\alpha_0} [t_1] \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} [t_n] \xrightarrow{\alpha_n} [t']$$

such that $[\alpha] = [\alpha_0 ; \ldots ; \alpha_n]$. *In addition, we can always choose all the* α_i *corresponding to sequential rewrites.*

Proof. The proof is by induction on the depth of proof terms, with proof terms of the form $[t] : [t] \to [t]$ having depth 0, and any other proof term having depth 1 plus the maximum of the depths of its argument subterms. For proof terms of depth 0 the result is obvious. Let us assume the result for proof terms of depth less or equal than n, and show that it holds for proof terms of depth $n + 1$. Beforehand, we should make explicit what it means for a proof term α to be a one-step concurrent rewrite, or a one-step sequential rewrite. The proof term α is a *one-step concurrent rewrite* if it contains at least one subterm of the form $r(\overline{\alpha}^n, \overline{\beta}^k)$ and all occurrences of the operator $\_ ; \_$ in α are within the $\overline{\beta}^k$ part of such subterms (i.e., transitivity is only used to establish conditions); if, in addition, there is exactly one subterm of the form $r(\overline{\alpha}^n, \overline{\beta}^k)$ in α, then α is called a *one-step sequential rewrite.* To prove the induction step we reason by cases.

Case $\alpha = f(\alpha_1, \ldots, \alpha_n)$. If all the α_i are equivalent to identities, say $[\alpha_i] = [[t_i]]$, then, by the functoriality axiom (2-b) of preservation of identities we have $[\alpha] = [[f(t_1, \ldots, t_n)]]$ and we are done. Otherwise, at least one of the α_i is not equivalent to an identity. Using the induction hypothesis and adding identities when needed by means of the identities

axiom (1-b), we can decompose each α_i, $1 \leqslant i \leqslant n$, as $[\alpha_i] = [\alpha_i^1; \ldots; \alpha_i^m]$, where all the α_i^j are either one-step concurrent rewrites or identities, and where for at least one α_i none of the α_i^j are identities. The result then follows by repeated application of the functoriality axiom (2-a) of preservation of composition by first decomposing α as

$$[\alpha] = [f(\alpha_1^1, \ldots, \alpha_n^1); \ldots; f(\alpha_1^m, \ldots, \alpha_n^m)]$$

and then (using again axiom (2-b) together with the identities axiom (1-b)) decomposing each $f(\alpha_1^j, \ldots, \alpha_n^j)$ into a composition of one-step rewrites by "firing a nonidentity α_i^j at a time and leaving the rest fixed." Since by the induction hypothesis we may in addition assume that the nonidentity α_i^j are one-step sequential rewrites, the one-step rewrites just described can always be made sequential.

Case $\alpha = r(\overline{\alpha}^n, \overline{\beta}^k)$. By the decomposition axiom (4) we have the equality

$$r(\overline{\alpha}^n, \overline{\beta}^k) = r(\overline{[w]}^n, \overline{\beta}^k); t'(\overline{\alpha}^n)$$

where $r(\overline{[w]}^n, \overline{\beta}^k)$ is by construction a one-step sequential rewrite. The result now follows using the functoriality of t' and the induction hypothesis and reasoning exactly as in the case $\alpha = f(\alpha_1, \ldots, \alpha_n)$ to decompose $t'(\overline{\alpha}^n)$ into either an identity or a sequence of one-step rewrites; again, all such steps can be made sequential.

Case $\alpha = \alpha_1; \alpha_2$. Trivial. □

If one desires to make finer distinctions among the computations of a system, a different set of equations, perhaps weaker, perhaps incorporating also "symmetries"—which relax for morphisms the strictness with which the equations E are imposed—can be given. For the special case of Petri nets, Pierpaolo Degano, Ugo Montanari and I have studied such possibilities—giving an axiomatic account of familiar distinctions among Petri net computations—in [31,32]. The general technical concept relevant for studying relaxations of this kind is that of *coherence* for a certain structure on a category [83].

3.2. $\mathcal{R}$-systems

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

Definition 3.7. Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, an *$\mathcal{R}$-system* $\mathcal{S}$ is a category $\mathcal{S}$ together with:

(i) a (Σ, E)-algebra structure, i.e., for each $f \in \Sigma_n$, $n \in \mathbb{N}$, a functor $f_{\mathcal{S}} : \mathcal{S}^n \to \mathcal{S}$, in such a way that the equations E are satisfied, 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;

(ii) for each rewrite rule

$$r : [t(\overline{x})] \to [t'(\overline{x})] \ \textit{if}$$
$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R, a natural transformation

$$r : J_{\mathcal{S}} * t_{\mathcal{S}} \Rightarrow J_{\mathcal{S}} * t'_{\mathcal{S}},$$

where $J_{\mathcal{S}} : \mathit{Subeq}((u_{j\mathcal{S}}, v_{j\mathcal{S}})_{1 \leqslant j \leqslant k}) \to \mathcal{S}^n$ is the subequalizer functor.

An *$\mathcal{R}$-homomorphism* $F : \mathcal{S} \to \mathcal{S}'$ between two $\mathcal{R}$-systems is then a functor $F : \mathcal{S} \to \mathcal{S}'$ 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

$$r : [t(\overline{x})] \to [t'(\overline{x})] \ \textit{if}\ C$$

in R we have the identity of natural transformations:

$$r_{\mathcal{S}} * F = F^{\bullet} * r_{\mathcal{S}'}$$

where $F^{\bullet} : \mathit{Subeq}(C_{\mathcal{S}}) \to \mathit{Subeq}(C_{\mathcal{S}'})$ is the unique functor induced by the universal property of $\mathit{Subeq}(C_{\mathcal{S}'})$ by the composition functor

$$\mathit{Subeq}(C_{\mathcal{S}}) \xrightarrow{J_{\mathcal{S}}} \mathcal{S}^n \xrightarrow{F^n} \mathcal{S}'^n$$

and the natural transformations

$$\alpha_j * F, \quad 1 \leqslant j \leqslant k,$$

where *if* C has k conditions $[u_j] \to [v_j]$, $1 \leqslant j \leqslant k$, α_j is the jth natural transformation

$$\alpha_j : J_{\mathcal{S}} * u_{j\mathcal{S}} \to J_{\mathcal{S}} * v_{j\mathcal{S}}$$

associated to the subequalizer $\mathit{Subeq}(C_{\mathcal{S}})$. Note that for this to make sense we have used the identities

$$u_{j\mathcal{S}} * F = F^n * u_{j\mathcal{S}'}, \qquad v_{j\mathcal{S}} * F = F^n * v_{j\mathcal{S}'},$$

which hold for derived operations because F is a Σ-homomorphism and that allow us to express $\alpha_j * F$ as a natural transformation

$$\alpha_j * F : J_{\mathcal{S}} * F^n * u_{j\mathcal{S}'} \Rightarrow J_{\mathcal{S}} * F^n * v_{j\mathcal{S}'}.$$

Despite the somewhat complicated definition of $F^\bullet$, its behavior on objects is quite simple; it is given by the equation

$$F^\bullet(\overline{C}^n, \overline{c}^k) = (F^n(\overline{C}^n), F^k(\overline{c}^k)).$$

This defines a category $\underline{\mathcal{R}\text{-}Sys}$ in the obvious way.

This category has the additional property that the homsets $\underline{\mathcal{R}\text{-}Sys}(\mathcal{S}, \mathcal{S}')$ are themselves categories with morphisms, called *modifications*, given by natural transformations $\delta : F \to G$ between $\mathcal{R}$-homomorphisms $F, G : \mathcal{S} \to \mathcal{S}'$ satisfying the identities

$$\delta^n * f_{\mathcal{S}'} = f_{\mathcal{S}} * \delta$$

for each $f \in \Sigma_n$, $n \in \mathbb{N}$. This category structure actually makes $\underline{\mathcal{R}\text{-}Sys}$ into a 2-category [83,70].

An *$\mathcal{R}$-groupoid* is an $\mathcal{R}$-system $\mathcal{S}$ whose category structure is actually a groupoid. This defines a full subcategory $\underline{\mathcal{R}\text{-}Grpd} \subseteq \underline{\mathcal{R}\text{-}Sys}$.

3.3. Computational interpretation of $\mathcal{R}$-systems

What the definition of $\mathcal{R}$-system captures formally is the idea that the models of a rewrite theory *are systems*. By a "system" I of course 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 of course 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, this is in a sense the end of the story (see Section 5.1.1). As the examples discussed in Section 5 will make clear, what makes a system *concurrent* is precisely the existence of an additional *algebraic structure*. Ugo Montanari and the author first observed this fact for the particular case of Petri nets for which the algebraic structure is precisely that of a commutative monoid [96,97]. However, this observation holds in full generality for *any algebraic structure whatsoever*. What the algebraic structure captures is twofold. Firstly, *the states themselves are distributed according to such a structure*; for Petri nets (see Section 5.3.1) the distribution takes the form of a *multiset* that we can visualize with tokens and places; for a functional program involving just syntactic rewriting, the distribution takes the form of a *labelled tree structure* which can be spatially distributed in such a way that many transitions (i.e., rewrites) can happen concurrently in a way analogous to the concurrent firing of transitions in a Petri net. Secondly, *concurrent transitions are themselves distributed according to the same algebraic structure*; this is what the notion of $\mathcal{R}$-system captures, and

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

Fig. 4. The mathematical structure of concurrent systems.

is for example manifested in the concurrent firing of Petri nets 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 and—in addition—a proof of the corresponding substitution instance of the rule's condition can be obtained. The rewrite rule is then a *procedure*[15] 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 what the concept of a *natural transformation* formalizes. The table of Fig. 4 summarizes our present discussion; the most crucial correspondence listed in that figure is the identification of a system's distributed structure with its algebraic structure. The best way to exhibit that identification is by example; in Section 5 we first put this in evidence by discussing labelled transition systems, where both concurrency and algebraic structure clearly shine by their absence; later in that section we discuss truly concurrent systems such as parallel functional programs, Petri nets and concurrent object-oriented systems as well as a variety of other models that can be obtained by specialization from concurrent rewriting and show that concurrent rewriting is indeed a very general and flexible model of concurrent computation.

3.4. Initial and free $\mathcal{R}$-systems

As a further confirmation that the definitions of $\mathcal{R}$-system and $\mathcal{R}$-homomorphism are very natural we leave checking the following lemma to the reader.

Lemma 3.8. *The full subcategory of* $\underline{\mathcal{R}\text{-PreSys}}$ *determined by those presystems that satisfy the equations*[16] (1)–(5) *is isomorphic to the category* $\underline{\mathcal{R}\text{-Sys}}$.

[15]Its *actual parameters* are precisely given by a substitution.

[16]More precisely, the slightly generalized equations obtained from (1)–(5) by replacing nodes $[t] \in T_{\Sigma,E}(X)$ by nodes C in the set of nodes of an $\mathcal{R}$-presystem.

Therefore, the notion of $\mathcal{R}$-system is also "essentially algebraic" and can be specified by adding equations of the type (1)–(5) to the essentially algebraic theory defining $\mathcal{R}$-presystems. As usual in universal algebra, whenever we have a full subcategory defined by a collection of equations, that subcategory is *reflective* inside the bigger one, i.e., its inclusion has a left adjoint called a *reflection* (see, e.g., [115] or [13, Theorem 4.4.1] for an even more general result implying this property). In our case, this means that the full subcategory inclusion

$$\underline{\mathcal{R}\text{-}Sys} \hookrightarrow \underline{\mathcal{R}\text{-}PreSys}$$

has a left adjoint. In fact, for presystems of the form $\mathcal{P}_{\mathcal{R}}(X)$ it is quite easy to see that the quotient $\mathcal{R}$-prehomomorphism $Q_X : \mathcal{P}_{\mathcal{R}}(X) \to \mathcal{T}_{\mathcal{R}}(X)$ associated to the congruence defining $\mathcal{T}_{\mathcal{R}}(X)$ is in fact the reflection map or "unit" of such an adjunction. Indeed, given an $\mathcal{R}$-system $\mathcal{S}$ and an $\mathcal{R}$-prehomomorphism $F : \mathcal{P}_{\mathcal{R}}(X) \to \mathcal{S}$, since $\mathcal{S}$, being an $\mathcal{R}$-system, satisfies equations just like those used in the definition of $\mathcal{T}_{\mathcal{R}}(X)$, there is a containment of the congruences associated to Q_X and F as follows:

$$Ker(Q_X) \subseteq Ker(F);$$

this defines a unique $\mathcal{R}$-prehomomorphism (therefore an $\mathcal{R}$-homomorphism) $F^\dagger : \mathcal{T}_{\mathcal{R}}(X) \to \mathcal{S}$ such that $F = Q_X * F^\dagger$. The following important result becomes now a trivial consequence of this discussion.

Theorem 3.9 (Initiality). *The $\mathcal{R}$-system $\mathcal{T}_{\mathcal{R}}$ is an initial object in the category $\underline{\mathcal{R}\text{-}Sys}$, and the $\mathcal{R}$-groupoid $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}$ is an initial object in the category $\underline{\mathcal{R}\text{-}Grpd}$. More generally, the system $\mathcal{T}_{\mathcal{R}}(X)$ has the following universal property: Given an $\mathcal{R}$-system $\mathcal{S}$, each function $F : X \to Obj(\mathcal{S})$ extends uniquely to an $\mathcal{R}$-homomorphism $F^\natural : \mathcal{T}_{\mathcal{R}}(X) \to \mathcal{S}$. The groupoid $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}(X)$ has the same universal property with respect to $\mathcal{R}$-groupoids.*

Proof. All the theorem says is that the composition functor

$$\underline{\mathcal{R}\text{-}Sys} \hookrightarrow \underline{\mathcal{R}\text{-}PreSys} \xrightarrow{Nodes} \underline{Set}$$

has a left adjoint sending each X to $\mathcal{T}_{\mathcal{R}}(X)$. We have just discussed the left adjoint for the full inclusion $\underline{\mathcal{R}\text{-}Sys} \hookrightarrow \underline{\mathcal{R}\text{-}PreSys}$ and shown that it maps $\mathcal{P}_{\mathcal{R}}(X)$ to $\mathcal{T}_{\mathcal{R}}(X)$. Proposition 3.2 showed that the left adjoint for the functor $Nodes : \underline{\mathcal{R}\text{-}PreSys} \to \underline{Set}$ maps X to $\mathcal{P}_{\mathcal{R}}(X)$. The theorem then follows from the well known fact that the composition of two left adjoints is a left adjoint [83]. The groupoid case is entirely analogous. □

This theorem suggests a generalized *initial model semantics* for rewrite theories that associates to a rewrite theory $\mathcal{R}$ the initial $\mathcal{R}$-system $\mathcal{T}_{\mathcal{R}}$.

Traditionally, the model associated to a rewrite theory $\mathcal{R}$ is the initial algebra $T_{\mathcal{R}}$ determined by the equations in $\mathcal{R}$ (i.e., by both the axioms E and the rules R in $\mathcal{R}$). While this is perfectly adequate for rewrite theories that are Church–Rosser and terminating, we have already seen in Section 1.3 that the traditional initial algebra semantics is entirely inadequate for many other rewrite theories. The system $\mathcal{T}_{\mathcal{R}}$ provides the analogue of $T_{\mathcal{R}}$ for the general case. In addition, we shall see in Section 3.6 that the old and the new semantics are nicely related by means of an adjoint functor—actually a reflection—yielding a surjective counit map $\mathcal{T}_{\mathcal{R}} \to T_{\mathcal{R}}$ from which the old semantics can be recovered as a quotient of the new.

3.5. Satisfaction, soundness and completeness

Since rewriting logic has a notion of *sentence*, namely a sequent $[t] \to [t']$, we should consider what it means for an $\mathcal{R}$-system $\mathcal{S}$ to satisfy such a sequent. Intuitively, the sequent indicates the existence of an arrow in the category. However, since the terms t and t' may have variables, it is not a fixed arrow, but a "variable" one. Of course, the obvious candidate for the intuitive notion of a "variable arrow" is the concept of natural transformation. Lemmas 3.3 and 3.5 suggest that such an interpretation is indeed the right one.

Definition 3.10. A sequent $[t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$ is *satisfied* by an $\mathcal{R}$-system $\mathcal{S}$ if there exists a natural transformation

$$\alpha : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$$

between the functors $t_{\mathcal{S}}, t'_{\mathcal{S}} : \mathcal{S}^n \to \mathcal{S}$. We use the notation

$$\mathcal{S} \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

to denote the satisfaction relation. Similarly, for $\Theta \subseteq \underline{\mathcal{R}\text{-}Sys}$ a class of $\mathcal{R}$-systems,

$$\Theta \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

states that the sequent is satisfied by all $\mathcal{R}$-systems in Θ. Finally,

$$\mathcal{R} \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

states that the sequent is satisfied by all $\mathcal{R}$-systems.

Theorem 3.11 (Soundness). *For $\mathcal{R}$ a rewrite theory,*

$$\mathcal{R} \vdash [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

implies

$$\mathcal{R} \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)].$$

Proof. For $\mathcal{S}$ an arbitrary $\mathcal{R}$-system we have to exhibit for each proof term

$$\gamma : [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

a natural transformation

$$\gamma_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}.$$

We define $\gamma_{\mathcal{S}}$ by induction on the depth of proof terms; as before, proof terms of the form $[t] : [t] \to [t]$ have depth 0, and any other proof term has depth 1 plus the maximum of the depths of its argument subterms.

For γ a proof term of depth 0, say, $\gamma = [t(x_1,\ldots,x_n)] : [t(x_1,\ldots,x_n)] \to [t(x_1,\ldots,x_n)]$, we define $\gamma_{\mathcal{S}}$ as the identity natural transformation

$$t_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t_{\mathcal{S}}.$$

For $\gamma = f(\overline{\alpha}) : [f(\overline{t})] \to [f(\overline{t'})]$ we have by induction hypothesis natural transformations

$$\alpha_{i\mathcal{S}} : t_{i\mathcal{S}} \Rightarrow t'_{i\mathcal{S}} : \mathcal{S}^m \to \mathcal{S}, \quad 1 \leqslant i \leqslant n,$$

and therefore—by the universal property of a product of categories—a natural transformation

$$\overline{\alpha_{\mathcal{S}}} : \overline{t_{\mathcal{S}}} \Rightarrow \overline{t'_{\mathcal{S}}} : \mathcal{S}^m \to \mathcal{S}^n,$$

which, composed horizontally with the functor $f_{\mathcal{S}} : \mathcal{S}^n \to \mathcal{S}$, yields a natural transformation

$$\overline{\alpha_{\mathcal{S}}} * f_{\mathcal{S}} : f(\overline{t})_{\mathcal{S}} \Rightarrow f(\overline{t'})_{\mathcal{S}} : \mathcal{S}^m \to \mathcal{S}$$

which is our desired $\gamma_{\mathcal{S}}$.

For $\gamma = r(\overline{\alpha}^n, \overline{\beta}^k) : [t(\overline{w}/\overline{x})] \to [t'(\overline{w'}/\overline{x})]$ we have by induction hypothesis natural transformations

$$\alpha_{i\mathcal{S}} : w_{i\mathcal{S}} \Rightarrow w'_{i\mathcal{S}} : \mathcal{S}^m \to \mathcal{S}, \quad 1 \leqslant i \leqslant n,$$

which yield a single natural transformation

$$\overline{\alpha_{\mathcal{S}}}^n : \overline{w_{\mathcal{S}}}^n \Rightarrow \overline{w'_{\mathcal{S}}}^n : \mathcal{S}^m \to \mathcal{S}^n.$$

Also by the induction hypothesis we have natural transformations

$$\beta_{j\mathcal{S}} : u_j(\overline{w}/\overline{x})_{\mathcal{S}} \Rightarrow v_j(\overline{w}/\overline{x})_{\mathcal{S}} : \mathcal{S}^m \to \mathcal{S}, \quad 1 \leqslant j \leqslant k,$$

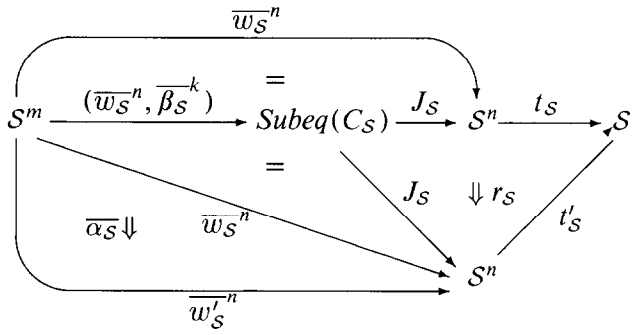
which by the universal property of $Subeq((u_{j\mathcal{S}}, v_{j\mathcal{S}})_{1 \leqslant j \leqslant k})$ yield a unique functor

$$(\overline{w_{\mathcal{S}}}^n, \overline{\beta_{\mathcal{S}}}^k) : \mathcal{S}^m \to Subeq((u_{j\mathcal{S}}, v_{j\mathcal{S}})_{1 \leqslant j \leqslant k})$$

such that

$$(\overline{w_{\mathcal{S}}}^n, \overline{\beta_{\mathcal{S}}}^k) * J_{\mathcal{S}} = \overline{w_{\mathcal{S}}}^n.$$

The desired natural transformation $\gamma_{\mathcal{S}}$ is given by the pasting diagram



which uses the fact that $\mathcal{S}$ is an $\mathcal{R}$-system, so that the natural transformation $r_{\mathcal{S}}$ exists by hypothesis.

Finally, for $\gamma = \alpha ; \beta$, we define $\gamma_{\mathcal{S}} = \alpha_{\mathcal{S}} ; \beta_{\mathcal{S}}$. □

Fact 3.12. *Although the definition of $\gamma_{\mathcal{S}}$ has been given in terms of proof terms, it is easy to check that all representatives β in an equivalence class of proof terms*

$$[\gamma] : [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$$

in $\mathcal{T}_{\mathcal{R}}(X)$ are assigned the same natural transformation $\gamma_{\mathcal{S}}$. In particular, for $\mathcal{S} = \mathcal{T}_{\mathcal{R}}(X)$ the instantiation of the natural transformation $\gamma_{\mathcal{T}_{\mathcal{R}}(X)}$ to a sequence of objects $\overline{[w]}$ is denoted $\gamma_{\mathcal{T}_{\mathcal{R}}(X)}(\overline{[w]}) = \gamma(\overline{[w]}/\overline{x})$, and called the substitution *of $\overline{[w]}$ in γ.*

Theorem 3.13 (Completeness). *For $\mathcal{R}$ a rewrite theory,*

$$\mathcal{R} \models [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$$

implies

$$\mathcal{R} \vdash [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)].$$

Proof. Since $\mathcal{R} \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$, we have in particular that

$$\mathcal{T}_{\mathcal{R}}(\{x_1,\ldots,x_n\}) \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

and therefore a natural transformation

$$\alpha : [t(x_1,\ldots,x_n)] \Rightarrow [t'(x_1,\ldots,x_n)] :$$
$$\mathcal{T}_{\mathcal{R}}(\{x_1,\ldots,x_n\})^n \to \mathcal{T}_{\mathcal{R}}(\{x_1,\ldots,x_n\}),$$

which, when instantiated for the objects $[x_1],\ldots,[x_n]$, yields an equivalence class of proof terms

$$\alpha([x_1],\ldots,[x_n]) : [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)].$$

Therefore, the sequent $[t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$ is provable from $\mathcal{R}$, as desired. □

As a direct consequence of the proof of the Completeness Theorem, together with soundness, we get the following corollary.

Corollary 3.14. *For $\mathcal{R}$ a rewrite theory,*

$$\mathcal{R} \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

iff

$$\mathcal{R} \vdash [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

iff there is a morphism

$$[t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

in $\mathcal{T}_{\mathcal{R}}(\{x_1,\ldots,x_n\})$.

3.6. Equationally defined classes of models

Since $\mathcal{R}$-systems are an "essentially algebraic" concept, we can consider classes Θ of $\mathcal{R}$-systems defined by the satisfaction of additional equations. Such classes give rise to full subcategory inclusions

$$\Theta \hookrightarrow \underline{\mathcal{R}\text{-}Sys}$$

and by general universal algebra results about essentially algebraic theories (see, e.g., [115,13]) such inclusions are reflective. More generally, we wish also to consider subcategories Θ which are reflective, i.e., their inclusion functor has a left adjoint, but are neither definable by additional equations, nor full, i.e., their morphisms are more restrictive; continuous poset models furnish an example of this latter kind of subcategory. In general, $\Theta \hookrightarrow$

$\underline{\mathcal{R}\text{-}Sys}$ reflective means that for each $\mathcal{R}$-system $\mathcal{S}$ there is an $\mathcal{R}$-system $R_\Theta(\mathcal{S}) \in \Theta$ and an $\mathcal{R}$-homomorphism $\rho_\Theta(\mathcal{S}) : \mathcal{S} \to R_\Theta(\mathcal{S})$ such that for any $\mathcal{R}$-homomorphism $F : \mathcal{S} \to \mathcal{D}$ with $\mathcal{D} \in \Theta$ there is a unique morphism $F^\diamond : R_\Theta(\mathcal{S}) \to \mathcal{D}$ in Θ such that $F = \rho_\Theta(\mathcal{S}) ; F^\diamond$.

The full subcategory $\underline{\mathcal{R}\text{-}Grpd} \subseteq \underline{\mathcal{R}\text{-}Sys}$ also has a reflection functor, but it is not equationally definable. This situation generalizes that of the inclusion of the category of groups into the category of monoids. What we have in this case is an inclusion that is a *forgetful functor* from a category of algebras with additional operations (in this case the inversion operation). However, for any equationally definable (full) subcategory $\Theta \subseteq \underline{\mathcal{R}\text{-}Sys}$, defined by a collection of equations H, the intersection $\Theta \cap \underline{\mathcal{R}\text{-}Grpd}$ has a very simple description, since it is just the full subcategory of $\underline{\mathcal{R}\text{-}Grpd}$ definable by the equations H.

Therefore, we can consider subcategories of $\underline{\mathcal{R}\text{-}Sys}$ or $\underline{\mathcal{R}\text{-}Grpd}$ 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. More generally, we can consider reflective subcategories; they also have intial objects because reflections preserve them. In this way, we can conceive of not one, but *several initial model semantics* for a rewrite theory $\mathcal{R}$, depending on how restrictive we want (or can afford) our semantics to be, just by restricting all the models to be in the subcategory of our choice. We show below that many important classes of models can be obtained in this way, including classes of preordered or ordered algebras defined by inequalities (which are for example used extensively in the domain theory of denotational semantics, and—when the order is a subset ordering—in algebraic approaches to nondeterminism) as well as $\mathcal{R}$-groupoids, $\mathcal{R}$-cquivalence relations, and finally $\mathcal{R}$-algebras, which are the traditional models when the rules in $\mathcal{R}$ are interpreted as equations.

Consider for example the following conditional equations:

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

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

The first equation forces a category to be a preorder, and the addition of the second requires this preorder to be a poset. By imposing the first one, or by imposing both, we get full subcategories

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

Recall that, as pointed out in Section 2.2, rewriting logic should be thought of as a very general form of *propositional logic*, in which the choice of propositional connectives is specified by the signature Σ. Therefore, from a logical point of view preorder and poset models are very natural models to consider; they are the analogue of Boolean or Heyting algebras for classical

or intuitionistic propositional logic. However, these models have also a computational interpretation as discussed below.

A routine inspection of $\underline{\mathcal{R}\text{-}Preord}$ for $\mathcal{R} = (\Sigma, E, L, R)$ reveals that its objects are preordered Σ-algebras $(A, \leqslant)$ (i.e., preordered sets with a Σ-algebra structure such that all the operations in Σ are monotonic) that satisfy the equations E and such that for each rewrite rule

$$r : [t(\overline{x})] \to [t'(\overline{x})] \text{ if}$$
$$[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$$

in R and for each $\overline{a} \in A^n$ such that $u_{jA}(\overline{a}) \leqslant v_{jA}(\overline{a})$ for $1 \leqslant j \leqslant k$, we have

$$t_A(\overline{a}) \leqslant t'_A(\overline{a}).$$

The poset case is entirely analogous, except that the relation $\leqslant$ is a partial order instead of being a preorder. In this context, sequents are interpreted as inequalities and it is more suggestive to express the above rewrite rule r in the form

$$r : [t(\overline{x})] \leqslant [t'(\overline{x})] \text{ if}$$
$$[u_1(\overline{x})] \leqslant [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \leqslant [v_k(\overline{x})].$$

It is also easy to see that our general notion of satisfaction of a sequent

$$[t(\overline{x})] \to [t'(\overline{x})]$$

by an $\mathcal{R}$-system specializes in the preorder case to satisfaction of the corresponding inequality

$$[t(\overline{x})] \leqslant [t'(\overline{x})].$$

The reflection functor associated to the inclusion $\underline{\mathcal{R}\text{-}Preord} \subseteq \underline{\mathcal{R}\text{-}Sys}$, being a left adjoint, preserves colimits. Therefore, it sends the initial $\mathcal{R}$-system $\mathcal{T}_{\mathcal{R}}$ to the initial $\mathcal{R}$-preorder, denoted $\to_{\mathcal{R}}$. This preorder is the familiar *rewriting relation* on ground E-equivalence classes associated to $\mathcal{R}$, except that now that its algebraic structure has been made apparent it is not viewed just as a relation. Similarly, for X a set of variables, $\mathcal{T}_{\mathcal{R}}(X)$ is mapped to the $\mathcal{R}$-rewriting relation $\to_{\mathcal{R}(X)}$ on E-equivalence classes of terms with variables in X. In general, the reflection functor associated to the inclusion $\underline{\mathcal{R}\text{-}Preord} \subseteq \underline{\mathcal{R}\text{-}Sys}$ maps each $\mathcal{R}$-system to the system's *reachability relation* which is conceived not just as a relation but as an algebraic structure reflecting the distributed structure of the system. Therefore, from the computational point of view we can think of preorder models as more abstract models in which only reachability information has been preserved about computations, but where the structure of states has been preserved intact.

Since the rewriting relation holds between $[t]$ and $[t']$ iff there exists a morphism $\alpha : [t] \to [t']$ in $\mathcal{T}_{\mathcal{R}}(X)$, Corollary 3.14 immediately gives us the following soundness and completeness result.

Corollary 3.15 (Completeness relative to preorders). *For $\mathcal{R}$ a rewrite theory,*

$$\underline{\mathcal{R}\text{-}Preord} \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

iff

$$\mathcal{R} \vdash [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

iff

$$[t(x_1,\ldots,x_n)] \to_{\mathcal{R}(X)} [t'(x_1,\ldots,x_n)]$$

for $X = \{x_1,\ldots,x_n\}$.

Similarly, the reflection associated to the inclusion $\underline{\mathcal{R}\text{-}Pos} \subseteq \underline{\mathcal{R}\text{-}Sys}$ maps $\mathcal{T}_{\mathcal{R}}(X)$ to the partial order $\leqslant_{\mathcal{R}(X)}$ obtained from the preorder $\to_{\mathcal{R}(X)}$ by identifying any two $[t], [t']$ such that $[t] \to_{\mathcal{R}(X)} [t']$ and $[t'] \to_{\mathcal{R}(X)} [t]$. From the computational point of view the reflection into posets is a coarser construction than the reflection into preorders because—in addition to throwing away all the details of a system's computation—it collapses all loops to identities thus giving an acyclic picture of a system's transitions. However, such models are still quite useful and have been used extensively in the context of denotational semantics where, following Scott's interpretation, the partial order $\leqslant$ is understood as an *approximation relation* so that each step in a system's computation brings us closer to the final solution. In fact, the model $\leqslant_{\mathcal{R}(X)}$ is nothing but the *free ordered algebra* on X associated to the set of inequalities defined by the rules in $\mathcal{R}$ which (in the unconditional case) has been studied—in connection with its extension by limits—in the denotational semantics literature (see, e.g., [17,54,87,88]). Actually, we can obtain the traditional continuos models of denotational semantics by considering the (nonfull) reflective subcategory inclusion

$$\underline{\mathcal{R}\text{-}Pos(\omega)} \hookrightarrow \underline{\mathcal{R}\text{-}Pos},$$

where $\underline{\mathcal{R}\text{-}Pos(\omega)}$ is the subcategory obtained by requiring that posets are ω-complete, i.e., countable ascending chains have limits, and that all operations in Σ and all homomorphisms are ω-continuous, i.e., preserve limits of countable chains. That this inclusion is reflective is well known [17,90,87,26,54], although the constructions in these references only treat the unconditional case. The reflection sends $\leqslant_{\mathcal{R}(X)}$ to $\leqslant^{\omega}_{\mathcal{R}(X)}$, the *free ω-continuous algebra*

on X associated to the set of inequalities defined by the rules in $\mathcal{R}$. Using Corollary 3.16 below and the fact that the adjunction map

$$\leqslant_{\mathcal{R}(X)} \rightarrow \leqslant^{\omega}_{\mathcal{R}(X)}$$

can be described as a completion by limits that leaves intact the order in $\leqslant_{\mathcal{R}(X)}$, it is easy to show that rewriting logic—reinterpreted in this context as a logic of inequalities—remains complete when the models are restricted to the class $\mathcal{R}\text{-}Pos(\omega)$. Note that, by composition of adjoints [83], the inclusion $\overline{\mathcal{R}\text{-}Pos(\omega)} \hookrightarrow \overline{\mathcal{R}\text{-}Sys}$ is also reflective.

Poset models of this kind are particularly appealing for the study of $\mathcal{R}$-systems performing functional computations in which $\mathcal{R}$ is Church–Rosser and acyclic; this case has been studied quite extensively, specially in connection with recursive function definitions (see for example [103,25,114] and references there). However, poset models—with or without a continuity assumption—can also be used in connection with non Church–Rosser and therefore nondeterministic systems; in particular, a variety of algebraic approaches to nondeterminism—many based on poset algebras whose elements are subsets of a given set—can be viewed as models of this kind. The Plotkin, Hoare and Smith powerdomains models belong in this category. In more direct connection with term rewriting and abstract data types, there is work by a number of authors including Boudol [18], Hesselink [57], Hussmann [63], Jayaraman and Plaisted [65], O'Donnell [106] and others. The elegant theory of *unified algebras* developed by Peter Mosses [102] also deals with nondeterminism by means of subsets. In fact, it seems that all these approaches can be mutually related by studying them within the equationally defined subclass of ordered models for an $\mathcal{R}$-system. For an example, consider the following quite interesting "powerset" $\mathcal{R}$-poset which seems to be related to ideas sketched by Michael O'Donnell in [106]. Let $\mathcal{R} = (\Sigma, E, L, R)$ be a rewrite theory. The powerset

$$\mathcal{P}(T_{\Sigma,E}(X))$$

has a natural Σ-algebra structure by pointwise extension of the operations, i.e., by defining

$$f_{\mathcal{P}(T_{\Sigma,E}(X))}(S_1,\ldots,S_n) = \{[f(t_1,\ldots,t_n)] \mid [t_i] \in S_i, 1 \leqslant i \leqslant n\}.$$

We call a subset $S \subseteq \mathcal{P}(T_{\Sigma,E}(X))$ an *$\mathcal{R}$-ideal* if whenever $[t] \in S$ and $[t] \rightarrow_{\mathcal{R}(X)} [t']$, then $[t'] \in S$. We denote by $Idl_{\mathcal{R}}(X)$ the set of all $\mathcal{R}$-ideals $S \subseteq \mathcal{P}(T_{\Sigma,E}(X))$. By assigning to each set S the smallest ideal $\overline{S}$ containing it, we define a monotonic "closure map"

$$\mathcal{P}(T_{\Sigma,E}(X)) \rightarrow Idl_{\mathcal{R}}(X).$$

This map is actually a Σ-homomorphism when we give to $Idl_{\mathcal{R}}(X)$ a Σ-algebra structure defined by

$$f_{Idl_{\mathcal{R}}(X)}(S_1,\ldots,S_n) = \overline{f_{\mathcal{P}(T_{\Sigma,E}(X))}(S_1,\ldots,S_n)}.$$

But in fact, $Idl_{\mathcal{R}}(X)$ is not only a Σ-algebra; it has a natural *$\mathcal{R}$-poset structure* with ordering given by $S \leqslant S'$ iff $S \supseteq S'$. This is so because the operations of $Idl_{\mathcal{R}}(X)$ are monotonic (thus functorial) and for any sequent $r: [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$ provable from R we have

$$t_{Idl_{\mathcal{R}}(X)}(S_1,\ldots,S_n) \supseteq t'_{Idl_{\mathcal{R}}(X)}(S_1,\ldots,S_n).$$

Although poset models can reflect the nondeterminism of a system, they cannot reflect at all the cyclic behavior of a system that returns to a previous state after a series of transitions. This makes the usefulness of poset models rather limited for treating reactive systems in which such cyclic behavior is important.

Since by definition we have $[t] \to_{\mathcal{R}(X)} [t']$ iff $[[t]] \leqslant_{\mathcal{R}(X)} [[t']]$, where $[[t]]$ denotes the equivalence class of $[t]$ in the poset associated to the preorder, we also have the following corollary.

Corollary 3.16 (Completeness relative to posets). *For $\mathcal{R}$ a rewrite theory,*

$$\underline{\mathcal{R}\text{-}Pos} \models [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

iff

$$\mathcal{R} \vdash [t(x_1,\ldots,x_n)] \to [t'(x_1,\ldots,x_n)]$$

iff

$$[[t(x_1,\ldots,x_n)]] \leqslant_{\mathcal{R}(X)} [[t'(x_1,\ldots,x_n)]]$$

for $X = \{x_1,\ldots,x_n\}$.

Intersecting $\underline{\mathcal{R}\text{-}Pos}$ and $\underline{\mathcal{R}\text{-}Preord}$ with the category $\underline{\mathcal{R}\text{-}Grpd}$ we get two subcategories definable by the first equation or by both, but now in the context of $\underline{\mathcal{R}\text{-}Grpd}$. Combining the notions of a groupoid and a preorder we get exactly the notion of an *equivalence relation* and therefore a subcategory $\underline{\mathcal{R}\text{-}Equiv}$ whose initial object is the usual congruence $\equiv_{\mathcal{R}}$ on (E-equivalence classes of) ground terms modulo provable equality generated by the rules in $\mathcal{R}$ when regarded as equations. Similarly, the free object of this class on a set X is the congruence generated by $\mathcal{R}$ on terms with variables in X. A poset that is also a groupoid yields a *discrete category* whose only arrows are identities, i.e., a set. It is therefore easy to see that the subcategory obtained by intersecting $\underline{\mathcal{R}\text{-}Pos}$ with $\underline{\mathcal{R}\text{-}Grpd}$ is just the familiar category $\underline{\mathcal{R}\text{-}Alg}$ of ordinary Σ-algebras that satisfy the equations $E \cup unlabel(R)$, where the

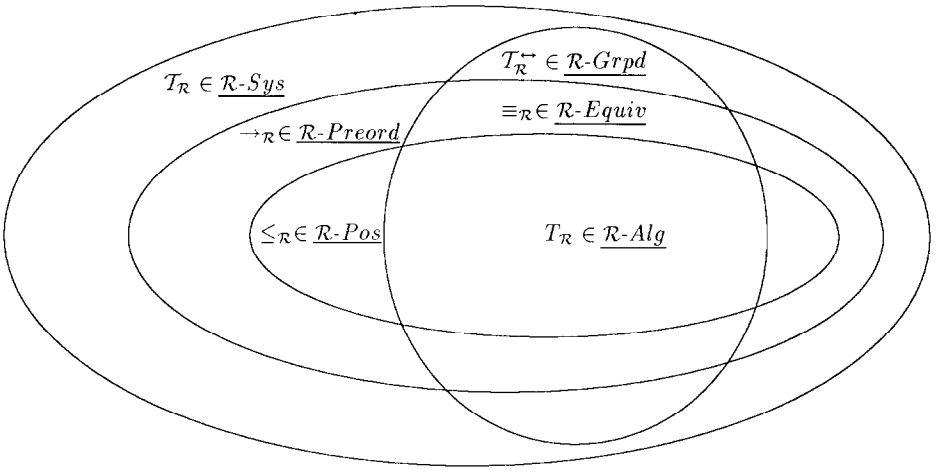


Fig. 5. Subcategories of $\underline{\mathcal{R}\text{-}Sys}$ and their initial objects.

unlabel function removes the labels from the conditional rules and turns the sequent signs "$\rightarrow$" into equality signs. Similarly, the reflection functor into $\underline{\mathcal{R}\text{-}Alg}$ maps $\mathcal{T}_{\mathcal{R}}(X)$ to $T_{\mathcal{R}}(X)$, the free $\mathcal{R}$-algebra on X. The reflection map

$$\rho_{\underline{\mathcal{R}\text{-}Alg}}(\mathcal{T}_{\mathcal{R}}(X)) : \mathcal{T}_{\mathcal{R}}(X) \rightarrow T_{\mathcal{R}}(X)$$

yields $T_{\mathcal{R}}(X)$ as a quotient of $\mathcal{T}_{\mathcal{R}}(X)$. Therefore, the *initial algebra semantics* of abstract data types [53] is recovered as a very special case of the more general semantics presented here. Note that—even for equational logic—the groupoid semantics is another possibility generalizing the classical algebraic semantics. Figure 5 summarizes the relationships among all these categories, except for the category $\underline{\mathcal{R}\text{-}Pos(\omega)}$ which is not included. This figure makes clear that the models of rewriting logic support a wide and flexible spectrum of choices for giving semantics to concurrent systems—plus the possibility of systematically relating those choices by means of adjoint functors. In the most general context of arbitrary $\mathcal{R}$-systems, the models are concurrent systems themselves and provide the most detailed model theoretic semantics, one with a strong operational flavor. At the expense of losing more and more information, models can become more and more abstract and "denotational" in flavor. Restricting ourselves to preorder models leaves intact the system's state structure, but collapses the transitions to a reachability preorder. Poset models further simplify the picture by collapsing all cycles between states. The extreme of information loss is reached with traditional initial algebra semantics, where the forwards and backwards directions of reachability are confused; for non Church–Rosser rewrite theories this is a brutal last step that in general should not be taken, as the NAT-CHOICE example discussed in Section 1.3 illustrates, but for Church–Rosser theories

it provides a very satisfactory and abstract denotational semantics. In fact, as in OBJ, classical initial algebra semantics is the denotational semantics adopted for Maude's functional modules (see Section 4).

Although any of the subcategories of $\underline{\mathcal{R}\text{-}Grpd}$ discussed above provides a suitable class of models for equational logic, the classical notion of model is of course provided by the algebras in $\underline{\mathcal{R}\text{-}Alg}$. The inclusion functor $\underline{\mathcal{R}\text{-}Alg} \hookrightarrow \underline{\mathcal{R}\text{-}Sys}$ is a key ingredient for establishing a systematic connection between rewriting logic and equational logic. Technically, such a connection takes the form of a *map of logics*

$$RewritingLogic \longrightarrow EquationalLogic$$

in the exact sense of [91] (note that, on models, the map goes backwards!). The two other ingredients required to establish such a map are the assignment to each rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ of the equational theory[17] $(\Sigma, E, unlabel(R))$, and the assignment to each sequent $[t] \rightarrow [t']$ of the equation $[t] = [t']$. Without entering into technical details, we remark that these data do indeed determine a map of logics in the sense of [91]. This map of logics is implicitly used to define the different semantics given in Maude to functional, system, and object-oriented modules (see Section 4) and provides in this way a very simple unification of the functional and concurrent programming paradigm. A quite striking fact in this regard is that the extension of functional programming to the context of concurrent systems programming is achieved by moving (backwards along the above map) to rewriting logic, a logic that is proof-theoretically *simpler* than equational logic. However, the very simplicity of the new proof theory allows a more sophisticated model theory which makes the traditional models of equational logic appear rather crude. This can be interpreted as an a posteriori explanation of why the treatment of nondeterminism and concurrency within a functional context has proved so difficult and has generally resulted in rather baroque and unsatisfactory solutions.

4. Rewrite rules as a programming language

The idea of using rewrite rules to specify algorithms and compute with them is not new; it is implicit in algebraic simplification techniques that go back to antiquity. In a formal version it can be traced back to the Herbrand–Gödel–Kleene theory of general recursive functions, and—in a variety of

[17] As for rewrite theories, we allow equational theories in which the equations are specified *modulo* E; this increases the expressiveness of equational theories.

limited forms [18]—it also goes back to Post systems, combinators, the lambda calculus and of course pure Lisp. In the context of equational logic, early proposals to program with rewrite rules include work by O'Donnell [104], and Goguen and Tardo [51], and a vast body of work has followed since that time [19]. However, the semantics of such rewrite rule languages has usually been based on equational logic, i.e., they have been given a *functional* interpretation.

In this paper I have put forward the view that, by generalizing the deduction rules and the model theory of equational logic to those of rewriting logic, a much broader field of applications for rewrite rule programming is possible—based on the idea of programming *concurrent systems* rather than *algebras*—with the same high standards of mathematical rigor for its semantics. Nevertheless, as explained below, the functional interpretation can still be maintained for those programs for which it is natural, i.e., we can at the same time achieve a very simple and rigorous integration of programming language paradigms.

4.1. Maude modules and their semantics

This section presents a specific proposal for a semantics of rewrite rule programs as concurrent systems. This proposal has two advantages. First, the functional case of equational logic is kept as a sublanguage having a more specialized semantics; second, the operational and mathematical semantics of a module are related in a particularly nice way. The proposal is embodied in Maude [92], a language containing OBJ3 [46] as its functional sublanguage. As already mentioned, all the ideas and results in this paper extend without problem [20] to the *order-sorted* case [21]; the unsorted case has only been used for the sake of a simpler exposition. Therefore, all that is said below is understood in the context of order-sorted rewriting logic. In Maude there are three kinds of *modules*:

- *functional modules*—introduced by the keyword `fmod`—such as the `NAT` and `REVERSE` modules in Section 1.2;
- *system modules*—introduced by the keyword `mod`—such as the module `NAT-CHOICE` in Section 1.3 and other modules to be discussed in Section 5; and

[18] See Section 5 for a discussion of how all these models can be regarded as special cases of concurrent rewriting.

[19] See in particular [105,39,46] for further developments of these two early proposals.

[20] Exercising of course the well known precaution of making explicit the universal quantification of rules.

[21] I.e., there is not just one sort, but a partially ordered set of sorts—with the ordering understood as type inclusion—and the function symbols can be overloaded [50].

- *object-oriented modules*—introduced by the keyword `omod`—that will be discussed in Section 5.3.4.

As we shall see later, the semantics of object-oriented modules reduces to that of system modules; therefore, in this section we focus on functional and system modules. Maude's functional and system modules are respectively of the form

- `fmod` $\mathcal{R}$ `endfm`, and
- `mod` $\mathcal{R}'$ `endm`,

for $\mathcal{R}$ and $\mathcal{R}'$ rewrite theories[22]. Their semantics is given in terms of an *initial machine* linking the module's operational semantics with its denotational semantics. The general notion of a machine is as follows.

Definition 4.1. For $\mathcal{R}$ a rewrite theory and $\Theta \hookrightarrow \underline{\mathcal{R}\text{-}Sys}$ a reflective subcategory, an *$\mathcal{R}$-machine over Θ* is an $\mathcal{R}$-homomorphism $[\![\_]\!] : \mathcal{S} \to \mathcal{M}$—called the machine's *abstraction map*—with $\mathcal{S}$ an $\mathcal{R}$-system and $\mathcal{M} \in \Theta$. Given $\mathcal{R}$-machines over Θ, $[\![\_]\!] : \mathcal{S} \to \mathcal{M}$ and $[\![\_]\!]' : \mathcal{S}' \to \mathcal{M}'$ an $\mathcal{R}$-machine *homomorphism* is a pair (F, G), with $F : \mathcal{S} \to \mathcal{S}'$ an $\mathcal{R}$-homomorphism, and $G : \mathcal{M} \to \mathcal{M}'$ in Θ, such that $[\![\_]\!] ; G = F ; [\![\_]\!]'$. This defines a category $\underline{\mathcal{R}\text{-}Mach/\Theta}$; it is easy to check that the initial object in this category is the unique $\mathcal{R}$-homomorphism $\mathcal{T}_{\mathcal{R}} \to R_{\Theta}(\mathcal{T}_{\mathcal{R}})$.

The intuitive idea behind a machine $[\![\_]\!] : \mathcal{S} \to \mathcal{M}$ is that we can use a *system* $\mathcal{S}$ to *compute* a result relevant for a *model* $\mathcal{M}$ of interest in a subcategory Θ of models. What we do is to perform a certain computation in $\mathcal{S}$, and then output the result by means of the abstraction map $[\![\_]\!]$. A very good example is an *arithmetic machine* with $\mathcal{S} = \mathcal{T}_{\mathtt{NAT}}$, for `NAT` the rewrite theory of the Peano natural numbers corresponding to the module `NAT`[23] in Section 1.2, with $\mathcal{M} = \mathbb{N}$, and with $[\![\_]\!]$ the unique homomorphism from the initial `NAT`-system $\mathcal{T}_{\mathtt{NAT}}$; i.e., this is the initial machine in $\underline{\mathtt{NAT}\text{-}Mach/\mathtt{NAT}\text{-}Alg}$. To compute the result of an arithmetic expression t, we perform a terminating rewriting and output the corresponding number, which is an element of $\mathbb{N}$.

Each choice of a reflective subcategory Θ as a category of models yields a different semantics. Possible choices include:

$$\Theta = \underline{\mathcal{R}\text{-}Sys}, \ \underline{\mathcal{R}\text{-}Preord}, \ \underline{\mathcal{R}\text{-}Pos}, \ \underline{\mathcal{R}\text{-}Pos(\omega)}, \ \underline{\mathcal{R}\text{-}Alg}.$$

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

[23]In this case E is the commutativity attribute, and R consists of the two rules for addition.

As already implicit in the arithmetic machine example, the *semantics of a functional module*[24] `fmod` $\mathcal{R}$ `endfm` is the initial machine in $\mathcal{R}$-*Mach*/$\mathcal{R}$-*Alg*, whose abstraction map is the unique $\mathcal{R}$-homomorphism $\mathcal{T}_{\mathcal{R}} \rightarrow \overline{T_{\mathcal{R}}}$ sending each $[t]$ to its equivalence class in $T_{\mathcal{R}}$ and transforming all morphisms into identitites.

For the *semantics of a system module* `mod` $\mathcal{R}$ `endm` not having any functional submodules[25] I propose the initial machine in $\mathcal{R}$-*Mach*/$\mathcal{R}$-*Preord*, whose abstraction map is the unique $\mathcal{R}$-homomorphism $\mathcal{T}_{\mathcal{R}} \rightarrow \overline{(\rightarrow_{\mathcal{R}})}$ which leaves the states unchanged but collapses the category $\mathcal{T}_{\mathcal{R}}$ to its associated reachability relation $\rightarrow_{\mathcal{R}}$. Although this choice of semantics seems natural for system modules, other choices are also possible; on the one hand, we could choose to be as concrete as possible and take $\Theta = \mathcal{R}Sys$ in which case the abstraction map is the identity homomorphism for $\mathcal{T}_{\mathcal{R}}$; on the other hand, we could instead be even more abstract, and choose $\Theta = \underline{\mathcal{R}\text{-}Pos}$; however, this would have the unfortunate effect of collapsing all the states of a cyclic rewriting, which seems undesirable for many reactive systems.

If the machine $\mathcal{T}_{\mathcal{R}} \rightarrow \mathcal{M}$ is the semantics of a functional or system module with rewrite theory $\mathcal{R}$, then we call $\mathcal{T}_{\mathcal{R}}$ the module's *operational semantics*, and $\mathcal{M}$ its *denotational semantics*. Therefore, the operational and denotational semantics of a module can be extracted from its initial machine semantics by projecting to the domain or codomain of the abstraction map. Note that the operational semantics of functional and system modules is defined in the same way, i.e., we compute with them in exactly the same way by concurrent rewriting; where they drastically differ is in their denotational semantics which for a functional module is an algebra, whereas for a system module is the sytem's reachability relation (together with its additional algebraic structure).

4.2. Submodules

In Maude a module can have *submodules*. Functional modules can only have functional submodules, but system modules can have both functional and system submodules; for example, `NAT` was declared a submodule of `NAT-CHOICE`. The meaning of submodule relations in which the submodule and the supermodule are both of the same kind is the obvious one, i.e., we augment the signature, equations, labels, and rules of the submodule by adding to them the corresponding ones in the supermodule; we then give semantics to the module so obtained according to its kind, i.e., functional or

[24]For this semantics to behave well, the rules R in the functional module `fmod` $\mathcal{R}$ `endfm` must be *ground Church–Rosser* modulo E (see Section 5.1.2 for a discussion of this important point).

[25]See below for a discussion of submodule issues.

system. The semantics of a system module having a functional submodule is somewhat more delicate. Suppose that the rewrite theory of the functional submodule[26] is $\mathcal{R} = (\Sigma, E, L, R)$ and that of the system supermodule plus its system submodules is $\mathcal{R}' = (\Sigma', E', L', R')$; as before we can form $\mathcal{R} \cup \mathcal{R}' = (\Sigma \cup \Sigma', E \cup E', L \cup L', R \cup R')$, but the semantics of the module is now given by the initial machine in the category

$$(\mathcal{R} \cup \mathcal{R}')\text{-}Mach/(\Sigma \cup \Sigma', E \cup E' \cup unlabel(R), L', R')\text{-}Preord.$$

Notice that $(\Sigma \cup \Sigma', E \cup E' \cup unlabel(R), L', R')$-*Preord* is an equationally definable full subcategory of $(\mathcal{R} \cup \mathcal{R}')$-*Preord*, namely the one defined by the equations $t(\overline{x}) = t'(\overline{x})$ for each rewrite rule $r : [t(\overline{x})] \to [t'(\overline{x})]$ in R, and therefore it is also reflective.

Given a preorder $\mathcal{M}$ in $(\Sigma \cup \Sigma', E \cup E' \cup unlabel(R), L', R')$-*Preord* we can forget about R' and the labels and view it as an $\mathcal{R}$-algebra $\mathcal{M}|_{\mathcal{R}}$. Given a system module `mod` $\mathcal{R}'$ `endm` having `fmod` $\mathcal{R}$ `endfm` as its functional submodule and $\mathcal{T}_{\mathcal{R} \cup \mathcal{R}'} \to \mathcal{M}$ as its semantics, we say that this submodule relation is *extending* if the unique $\mathcal{R}$-homomorphism $h : T_{\mathcal{R}} \to \mathcal{M}|_{\mathcal{R}}$ is injective; similarly, we say that it is *protecting* if h is an isomorphism. We leave for the reader to check that the extending relation asserted for the importation of `NAT` in `NAT-CHOICE` does in fact hold.

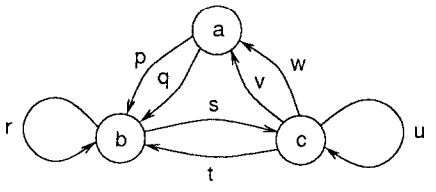
As OBJ, Maude has also *theories* to specify semantic requirements for interfaces and to make high level assertions about modules; they can be functional, system, or object-oriented. Also as OBJ, Maude has *parameterized modules*—again of the three kinds—and *views* that are theory interpretations relating theories to modules or to other theories. Details for all these aspects of the language will appear elsewhere[27]. Finally, note that Maude is a *logic programming language* in the general axiomatic sense made precise in [91].

5. Unifying models of concurrency

This section discusses a variety of models of concurrency that can be obtained as special cases of concurrent rewriting. A natural way of studying specializations of this kind is to impose restrictions on the rewrite theories being used. The most obvious restriction is fixing the set E of structural axioms. We consider three cases:

[26] We assume that, if several functional submodules have been declared, we have already taken their union.

[27] Some basic results about views and paramctcrization for system modules have already been given in [94].




```
mod LTS is
  sort State .
  ops a,b,c : -> State .
  rls p,q : a => b .
  rl    r : b => b .
  rl    s : b => c .
  rls v,w : c => a .
  rl    t : c => b .
  rl    u : c => c .
endm
```

Fig. 6. A labelled transition system and its code in Maude.

(1) *Syntactic rewriting*, i.e., $E = \emptyset$; this case includes labelled transition systems and functional programming.
(2) *String rewriting*, i.e., $E = AI$, associativity and identity; this case includes Post systems and phrase-structure grammars.
(3) *Multiset rewriting*, i.e., $E = ACI$, associativity, commutativity and identity; this case includes Petri nets, the chemical abstract machine, CCS, and a general logical theory of concurrent objects that yields Actors and the UNITY model of computation as special cases.

5.1. Case $E = \emptyset$

This is of course the best known case, and the one which is easiest to implement. We discuss two special cases, namely labelled transition systems and functional programming.

5.1.1. Labelled transition systems

This is the particularly simple case of unconditional rewrite theories obtained by imposing the additional requirements that $\Sigma = \Sigma_0$, i.e., Σ only involves constants, and that all the rules in $\mathcal{R}$ only involve *ground* terms, i.e., they are of the form $r : a \rightarrow b$ for a, b constants. For example, the transition system of Fig. 6 corresponds to the rewrite theory of the system module LTS in the same figure. Since Σ contains only constants and the rules have no variables, the rules (1)–(4) generating the proofs of rewriting logic specialize to very simple rules. The Σ-structure rule can only apply to constants $a \in \Sigma_0$ with no arguments so that for any such constant a we get a deduction

$$\frac{}{a : a \rightarrow a}$$

which becomes a special case of the identity rule thanks to the functoriality equation (2-b).

The rule of replacement has also to be applied with no arguments so that for each $r : a \to b$ in $\mathcal{R}$ we obtain the single deduction

$$\frac{}{r : a \to b}$$

Note that only the category equations (1) are relevant in this case; the equations (2)–(5) defining $\mathcal{T}_{\mathcal{R}}$ become vacuous or redundant due to the lack of arguments in the function symbols and in the rules, and to the vacuity of E. In particular, the decomposition and exchange laws (4)–(5) become trivial instances of the identity law (1-b) due to the lack of arguments in the system's transition rules. Therefore, we just have generation by identities, the rules $r : a \to b$ in $\mathcal{R}$, and composition with the category equations of associativity and identity, i.e., $\mathcal{T}_{\mathcal{R}}$ is just the *free category*—also called the *path category*—generated by the labelled transition system when regarded as a graph. More generally, *any* $\mathcal{R}$-system with $\mathcal{R}$ a labelled transition system is just a *category* $\mathcal{C}$ together with the assignment of an object of $\mathcal{C}$ to each constant in Σ and a morphism in $\mathcal{C}$ for each rule in R in a way consistent with the assignment of objects. In other words, such systems are just *sequential systems*, and their sequentiality is precisely due to the absence of any operations other than constants. In a negative way, they support our identification of distributed structure with algebraic structure, since they lack distributed structure precisely because they lack any algebraic structure beyond constants. In fact, labelled transition systems are intrinsically *sequential* as rewrite theories, in the precise sense of Definition 2.4. However, since several transitions are in general possible from a given state, they can exhibit a form of *nondeterminism*. In other words, labelled transition systems are a very special case of sequential but not necessarily deterministic rewrite theories.

Interleaving approaches to concurrency restrict themselves to labelled transition systems or similar sequential structures. We can always sequentialize a concurrent computation (see Lemmas 2.6 and 3.6) and much valuable work can be and has been done in this context. However, the context as such is intrinsically sequential and forces a form of *indirect reasoning* when considering concurrency aspects; therefore, it seems quite limited. Plato's analogy of the cave[28] may provide an apt metaphor for this situation, with labelled transition systems being the wall of the cave on which the shadows of true concurrency are reflected. The metaphor seems apt because it agrees with the mathematical facts; for $\mathcal{R}$ an arbitrary rewrite theory, the descent into the cave is precisely the forgetful functor $\underline{\mathcal{R}\text{-}Sys} \to \underline{Cat}$.

[28] Republic, Bk. VII, 514–517.

5.1.2. (Parallel) Functional programming

This case is obtained by requiring that the rewrite theory $\mathcal{R}$ is *ground Church–Rosser*. In Maude, it exactly corresponds to the case of functional modules. The Church–Rosser requirement allows us to regard the collection of rules with same top function symbol f on their lefthand sides as a collection of *recursive equations* defining the function f. If the rules are nonterminating, one has to worry about finding the unique normal form when there is one, and use a strategy that will accomplish this goal; however, this is less of a problem when the computation is performed in parallel, because in that case application of the rules is typically done in a fair way. Note that functional programming, as characterized by the ground Church–Rosser property, is not restricted to syntactic rewriting; rather, it extends to rewriting modulo axioms. For example, in OBJ and Maude it is possible to perform functional computations modulo a variety of sets of axioms. Therefore, the reader should keep clearly in mind that our inclusion of functional programming within the case $E = \emptyset$ is just a matter of expediency and a gross oversimplification.

The ground Church–Rosser property is required for Maude's functional modules precisely because only such modules provide a good way of computing the solution of a functional expression by rewriting. This provides further insights on the semantics given in Maude to a functional module `fmod` $\mathcal{R}$ `endfm`, namely the machine with abstraction map the $\mathcal{R}$-homomorphism $\mathcal{T}_{\mathcal{R}} \to T_{\mathcal{R}}$ sending each $[t]$ to its equivalence class in $T_{\mathcal{R}}$. Intuitively, such semantics associates to $[t]$ the result of its functional evaluation. Indeed, if the rules are terminating, $T_{\mathcal{R}}$ can be identified with the set of canonical forms, i.e., with a "canonical term algebra" [44] and we can run the machine by computing a terminating rewriting computation beginning with t and then extracting the corresponding canonical form as its semantics. For nonterminating computations this of course will not work; however, we can in that case view the equivalence class itself as an idealized result, or we can, under certain circumstances, associate to an infinite computation an infinite canonical form [34].

An issue worth considering is the computational interpretation of the adjective "parallel" in the expression "parallel functional programming" applied to Maude's functional modules. The adjective "concurrent" could have been used instead; however, "parallel" seems preferable because it can convey the additional nuance that—as it is the case for Church–Rosser rules—all terminating computations of an expression yield the same result. Note also that the parallelism of a functional module `fmod` $\mathcal{R}$ `endfm` is a property of its operational semantics $\mathcal{T}_{\mathcal{R}}$. From an operational semantics point of view, the module is regarded as a concurrent system whose states are terms built up with the syntax of the module's signature. This exemplifies very nicely

our claim that the distributed structure exhibited by a concurrent system *coincides* with the system's algebraic structure. In this case, states are spatially distributed according to a tree-structured topology determined by the operators in the signature. It is precisely this tree-structured distribution that makes possible the application of many rules at once, i.e., the parallelism. This is the most obvious fact staring at the reader in Fig. 2, the first example of a concurrent computation given in this paper[29]. Of course, the algebraic—and therefore distributed—structure of the system involves not only states but also transitions; this means that certain rewrite rules can be better than others from the point of view of providing greater parallelism in computing a function. Indeed, in trying to program a functional module for optimal parallelism, program transformations that improve both the data structures—i.e., the signature—and the rules while preserving correctness can be of great help.

Herbrand–Gödel–Kleene. The Herbrand–Gödel–Kleene [41,75] theory of general recursive functions, one of the basic models of computation at the beginning of computability theory, is a perfect example of computation by equational rewriting. After Ackerman showed that there are computable functions that cannot be defined by primitive recursion, the question arose of how to characterize the most general notion of a total computable function on the natural numbers. The Herbrand–Gödel-Kleene answer was later proved to be equivalent to that afforded by other models. In concurrent rewriting terms this notion can be defined as follows (we assume $\mathbb{N}$ in Peano notation, i.e., $\mathbb{N} = T_{\{0,s\}}$ for 0 a constant and s a unary function).

Definition 5.1. For Σ a finite signature, a collection of functions $f_{\mathbb{N}} : \mathbb{N}^k \to \mathbb{N}$, $f \in \Sigma_k$, $k \in \mathbb{N}$, is said to be *defined recursively* by a rewrite theory $\mathcal{R} = (\Sigma \uplus \{0, s\}, \emptyset, L, R)$ with R finite iff:

- $\mathcal{R}$ is ground Church–Rosser,
- the set of ground $\mathcal{R}$-normal forms is exactly $\mathbb{N}$, and
- for each $f \in \Sigma_k$, $n_1, \ldots, n_k$, $n \in \mathbb{N}$, $k \in \mathbb{N}$, $f(n_1, \ldots, n_k) \to n$ is an $\mathcal{R}$-rewrite iff $f_{\mathbb{N}}(n_1, \ldots, n_k) = n$.

The lambda calculus and combinatory logic. The lambda calculus is another of the original models of computation in computability theory and permits programming with higher order functions, even with functions that apply to themselves; of course, the Church–Rosser property is historically associated

[29]Note that, in that example, the tree-structured topology is rendered somewhat more flexible for nodes labelled by the + symbol, since the order of subtrees under such nodes is irrelevant thanks to the commutativity axiom.

with this model. Also, ideas about concurrent evaluation of redexes have been used in this context for a long time; a good example is the "parallel moves lemma" (see [29, IV.4, Theorem 5]).

Computation is directly based on rewriting, but in the standard lambda notation one has to deal with the extra fuss of variables and their substitution. Since in its classical notation substitution does not belong to the calculus itself but rather to the meta-level, in that notation the lambda calculus cannot be directly expressed by rewrite rules in our sense. This, however, is just a small accident of syntax and not at all an essential difficulty; the solution consists in making substitution an explicit part of the calculus. Indeed, a variety of solutions have been proposed in this regard. An elegant solution that is entirely faithful to the original lambda calculus is the $\lambda\sigma$-calculus of Abadi, Cardelli, Curien and Lévy [1]. The basic idea is to express substitutions as an explicit part of the term structure using De Bruijn notation, and to equationally axiomatize substitution by a set σ of Church–Rosser and terminating rewrite rules. Those rules, together with *Beta*—a version of the β rule which now also becomes a standard rewrite rule—form a set of Church–Rosser rules that perfectly simulate the original β rule while at the same time making explicit the computations implicit in the treatment of substitutions.

The reason why the $\lambda\sigma$-calculus so faithfully mirrors the original lambda calculus can be best explained by adopting the point of view of rewriting modulo a set of equations. Indeed, $\lambda\sigma$-terms in σ-canonical form are in one-to-one correspondence with lambda calculus terms (modulo α conversion). Therefore, there is an isomorphism between the terms of the lambda calculus (modulo α conversion) and the terms of the $\lambda\sigma$-calculus modulo the substitution equations σ. Under this isomorphism, lambda calculus rewriting using the β rule corresponds to rewriting with the rewrite theory $\lambda\sigma = (\Sigma, \sigma, L, Beta)$, with Σ the $\lambda\sigma$-calculus syntax, σ the substitution equations, L an appropriate set of labels, and *Beta* the rule corresponding to β. Specifically, under this isomorphism one step of β-reduction exactly corresponds to one step of *Beta*-rewriting of the form $[can_\sigma(t)] \rightarrow [t']$, where $[t]$ denotes the σ-equivalence class, $can_\sigma(t)$ is the σ-canonical form, and where $can_\sigma(t) \rightarrow t'$ is actually a syntactic *Beta*-rewriting. There is also the added bonus that one step of *Beta*-rewriting beginning with a term representative that is *not* in σ-canonical form may correspond in this isomorphism not to one but to *several* β-reduction steps. In summary, we can regard the traditional lambda calculus as syntactic sugar for a rewrite theory modulo equations.

Of course, combinators in either the classical Schönfinkel style [29,12,58], or supercombinators [62], or categorical combinators [28], provide another solution to this problem in terms of Church–Rosser equational rules; however, they simulate the lambda calculus less closely than the $\lambda\sigma$-calculus

does, and therefore are less satisfactory in this regard.

Algebraic data types. In this case the rules are both ground Church–Rosser and terminating; in Maude this is illustrated by functional modules such as NAT or REVERSE whose rules are terminating (and of course ground Church–Rosser). The module's denotational semantics is its corresponding initial algebra [53], which coincides with the canonical term algebra of normal forms obtained by rewriting [44,95]. Therefore, a module of this kind describes an algebraic abstract data type equationally axiomatized by the rules.

Of course, any functional expression in such a module is effectively computable by rewriting and evaluates to its canonical form. Therefore, these modules free computability from any numerical encoding and are in this sense a natural generalization of the Herbrand–Gödel–Kleene model. Indeed, any (total) computable function can be defined in this way [15]. This is the best known case for which the semantics of OBJ and of similar equational programming languages has been fully developed; however, the more general nonterminating case can also be handled nicely.

5.2. Case $E = AI$

The acronym AI stands for associativity and identity, i.e., we assume a binary operator which is associative (therefore, we can use empty syntax for the operator and denote the result of applying the operator to arguments x and y by xy) as well as a constant λ which is an identity element for that binary operation. Of course, there can be other function symbols around; however, in the following I will concentrate on the special case in which the only additional function symbols are constants and the rules are unconditional. This leads us to Post systems, Chomsky grammars, and Turing machines.

The flavor of concurrency exhibited by these systems is different from other examples. This is of course a world of strings where the algebraic—and therefore distributed—structure of a state is a linear structure, in which the linear order of the elements is fundamental and cannot be forgotten.

5.2.1. Post systems

This is the case in which $\Sigma_0 = \Delta \uplus \{\lambda\}$, $\Sigma_2 = \{\_\_\}$, and all the other Σ_n are empty. Therefore, $T_{\Sigma,AI} = \Delta^*$, and $T_{\Sigma,AI}(X) = (\Delta \uplus X)^*$. The rules of a rewrite theory for this case must have the form:

$$u_0 X_{k_1} u_1 \dots u_{n-1} X_{k_n} u_n \to v_0 X_{l_1} v_1 \dots v_{m-1} X_{l_m} v_m$$

with $n, m \in \mathbb{N}$, $u_i, v_j \in \Delta^*$, where the variables $X_{k_i}, X_{l_j} \in X$ could actually be repeated, i.e., we could have $X_{k_i} = X_{k_{i'}}$ with $i \neq i'$ and similarly for the

X_l's. The case considered by Post, i.e, what is called a *Post system*, places two very reasonable restrictions—besides requiring Δ and R finite—namely, that all the variables occurring in the lefthand side are different, and that the variables in the righthand side are among those in the lefthand side. Post studied sets L of words definable in terms of

- a subset $\Omega \subseteq \Delta$,
- a finite set $A \subseteq \Delta^*$, called the set of *axioms*, and
- a Post system $\mathcal{R}$,

by requiring that $v \in L$ iff $v \in \Omega^*$ and there is an axiom $u \in A$ with an $\mathcal{R}$-rewrite $u \to v$ (in Post terminology, v is then called a *theorem* deducible from the axioms A). Post then showed that such sets are exactly the recursively enumerable sets.

5.2.2. *Phrase-structure grammars and Turing machines*

Phrase-structure grammars are the particular case of Post systems $\mathcal{R}$ in which the rules only involve ground terms, i.e., they are of the form $u \to v$ with $u, v \in \Delta^*$. The problem of interest is the same, namely to study sets L, called *languages*, such that for a given subset $\Omega \subseteq \Delta$, called the set of *terminal symbols*, and for a single letter axiom $a \in \Delta - \Omega$, L is the set of theorems $\mathcal{R}$-deducible from a that lie in Ω^*. In addition, Ω must be such that all the words in Ω^* are in $\mathcal{R}$-normal form. The entire Chomsky hierarchy, with its associated machines to recognize each kind of language, is then obtained by imposing additional restrictions to the rules in $\mathcal{R}$. *Turing machines*, perhaps the mathematical model of computation that (for better or for worse) has so far influenced computer science and computer technology the most, appear naturally in that hierarchy. As is well known, they can be formalized as a particular type of context-sensitive phrase-structure grammar and therefore are just a very special type of *sequential* rewrite theories.

A natural question to ask is what the restriction of rewriting logic to phrase-structure grammars amounts to from the logical point of view. Actually, it corresponds to the linear conjunction fragment of *noncommutative* propositional linear logic [40]. We postpone discussing this matter until Section 5.3.2.

5.3. *Case $E = ACI$*

The acronym *ACI* stands for associativity, commutativity and identity, i.e., we add to *AI* a commutativity law $xy = yx$. We keep the same juxtaposition notation, but due to commutativity we can represent *ACI*-equivalence classes as monomials $a_1^{n_1} \ldots a_n^{n_k}$, where the order of the factors is immaterial, i.e., as *multisets*; in the presence of other function symbols, the binary *ACI* operator can of course appear at different levels, i.e., the a_i may themselves contain monomial subexpressions. We first consider Petri

nets, then the chemical abstract machine [16] and CCS. We then propose a new abstract theory of concurrent objects—in the sense of concurrent object-oriented programming—which yields Actors [3] and UNITY [21] as special cases. Engelfriet et al.'s POPs higher level Petri nets [36,37] can also be viewed as a special case of our theory of concurrent objects, but a precise account of their formalism would be somewhat lengthy and is therefore omitted.

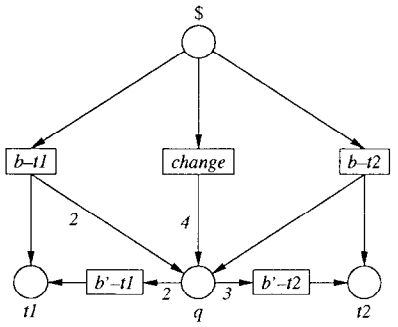
This case is quite important and contains as special subcases a good number of models that have already been studied. In fact, the associativity and commutativity of the axioms appear in some of those models as "fundamental laws of concurrency". However, from the perspective of this work the *ACI* case—while 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 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.

More importantly—and this is a key advantage of Maude's object-oriented modules—an *ACI* operator, even when present, does account only for *some* of the concurrency of a system when other operators not having that property are also present. For example, an object may communicate with other objects in an *ACI* distributed state, but its attributes can also have a distributed structure—typically a tree structure—so that their updating can be performed in parallel. Concurrent rewriting does not discriminate between one level of parallelism (*ACI* communication between objects) and another (parallel attribute updating); instead, it integrates both levels within the same formal framework supporting concurrency *at all levels*.

Of course, the general claim that a system's distributed structure coincides with its algebraic structure applies also here. The *ACI* axioms lead to a state structure that is distributed as a *commutative word*, *multiset*, or *bag*, all these being different expressions for the same idea. This is a very fluid and flexible structure which, in particular, is an ideal abstract structure for *communication*; as already pointed out, this may only account for the *top level structure* of a system, which in the framework of rewriting logic is seamlessly integrated with any other distributed structures at lower levels.

5.3.1. Petri nets

Petri nets are one of the most basic models of concurrency. It has the great advantage of exhibiting concurrency *directly*, not through the indirect mediation of interleavings. Its relationship to concurrent rewriting can be expressed very simply. It is just the particular case of an unconditional




```
mod TICKET is
  subsorts Place < Marking .
  ops $,q,t1,t2 : -> Place .
  op _⊗_ : Marking Marking -> Marking
                [assoc comm id: λ] .
  rl b-t1 : $ => t1 ⊗ q ⊗ q .
  rl b-t2 : $ => t2 ⊗ q .
  rl change : $ => q ⊗ q ⊗ q ⊗ q .
  rl b'-t1 : q ⊗ q => t1 .
  rl b'-t2 : q ⊗ q ⊗ q => t2 .
endm
```

Fig. 7. A Petri net and its code in Maude.

rewrite theory where $E = ACI$, $\Sigma_0 = \Delta \uplus \{\lambda\}$, $\Sigma_2 = \{\otimes\}$, with all the other Σ_n empty, and where the terms in the rules are all ground terms. Consider for example the Petri net in Fig. 7, which represents a machine to buy subway tickets. With a dollar we can buy a ticket $t1$ by pushing the button $b-t1$ and get two quarters back; if we push $b-t2$ instead, we get a longer distance ticket $t2$ and one quarter back. Similar buttons allow purchasing the tickets with quarters. Finally, with one dollar we can get four quarters by pushing *change*. The corresponding order-sorted rewrite theory is that of the TICKET system module in the same figure.

A key point about this module is that the operator $\otimes$—corresponding to *multiset union* of markings on the net—provides the system's commutative monoid algebraic structure (this is the "Petri nets as monoids" view [96,97]) which is at the same time its concurrent structure. That is, concurrent computations are possible in the net precisely due to the distributed nature of a marking as a multiset whose elements are put together by the binary union operator. By contrast, the state of a labelled transition system is *atomic*. Lacking any algebraic structure, concurrency becomes impossible. Of course, the commutative monoid structure is also enjoyed by the computations themselves. In this example, *ACI*-rewriting captures exactly the concurrent computations of the Petri net. Suppose, for example, that we begin in a state with four quarters and two dollars. Then, by first concurrently pushing the buttons $b'-t1$ and $b-t2$, and then concurrently pushing the buttons $b'-t2$ and $b-t2$ we end up with a ticket for the shorter distance, three tickets for the longer distance and a quarter, as shown in the two steps of concurrent *ACI*-rewriting below:

$$\mathtt{q} \otimes \mathtt{q} \otimes \mathtt{q} \otimes \mathtt{q} \otimes \$ \otimes \$ \rightarrow \mathtt{q} \otimes \mathtt{q} \otimes \mathtt{t1} \otimes \mathtt{t2} \otimes \mathtt{q} \otimes \$ \rightarrow \mathtt{t2} \otimes \mathtt{t1} \otimes \mathtt{t2} \otimes \mathtt{t2} \otimes \mathtt{q}.$$

The rules (1)–(4) generating the proofs of rewriting logic specialize as

follows. The Σ-structure rule applied to $\otimes$ yields the rule

$$\frac{\alpha : [t_1] \to [t_1'] \qquad \beta : [t_2] \to [t_2']}{\alpha \otimes \beta : [t_1 \otimes t_2] \to [t_1' \otimes t_2']}.$$

Similarly, the Σ-structure rule applied to each constant $a \in \Sigma_0$ yields a deduction

$$\frac{}{a : [a] \to [a]},$$

which becomes a special case of the identity rule thanks to the functoriality equation (2-b).

Since all the rules involve only ground terms, the rule of replacement has to be applied with no arguments so that for each transition rule $r : [t] \to [t']$ in the Petri net $\mathcal{N}$ we obtain the single deduction

$$\frac{}{r : [t] \to [t']}.$$

Note also that the equations (1)–(5) defining $\mathcal{T}_{\mathcal{N}}$ reduce to the category, functoriality and *ACI*-equations (1)–(3), since the decomposition and exchange laws become trivial instances of the identity law (1-b) due to the lack of arguments in the transition rules.

Thus, we just have generation by identities, the above Σ-structure rule for $\otimes$, the rules $r : [t] \to [t']$ in $\mathcal{N}$, and composition, subject to the category, functoriality and *ACI*-equations. Therefore, the $\mathcal{T}_{\mathcal{N}}$ construction specializes exactly to the monoidal category construction $\mathcal{T}[\mathcal{N}]$ associated to a Petri net $\mathcal{N}$ in joint work with Ugo Montanari [96,97].

5.3.2. Connections with linear logic

The connections with linear logic become now clear. Interpreting $\otimes$ as conjunction in linear logic [40] and denoting sequents with the $\vdash$ symbol, a Petri net $\mathcal{N}$ can be interpreted as a propositional theory whose axioms are the sequents $r : [t] \vdash [t']$ corresponding to the transitions $r : [t] \to [t']$ in $\mathcal{N}$.

Under this interpretation, the specialization of rewriting logic to Petri nets described above yields the basic axioms $r : [t] \vdash [t']$ of the linear theory associated to $\mathcal{N}$ as specializations of the replacement rule, as well as the following rules:

(1) *Identity.*

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

(2) $\otimes$.

$$\frac{[t_1] \vdash [t_1'] \qquad [t_2] \vdash [t_2']}{[t_1 \otimes t_2] \vdash [t_1' \otimes t_2']}.$$

(3) *Cut.*

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

which are sound and complete rules of deduction for the linear conjunction fragment of linear logic [9,55,85].

Regarding categorical models for linear logic, Narciso Martí-Oliet and I [85,84] realized that the category $\mathcal{T}_N$ originally constructed in [96,97] provides a categorical model for the linear conjunction fragment, studied general categorical models for the entire logic building on previous work by Seely [118] and Lafont [77], and studied the resulting triangular correspondence between Petri nets, linear logic and categories, which is a particular instance of the more general triangular correspondence between concurrent systems, rewriting logic and categories developed in this paper.

Note that phrase-structure grammars and Petri nets are very similar. Indeed, Petri nets can be obtained from phrase-structure grammars by replacing the empty syntax of word juxtaposition by the symbol $\otimes$ and making the words commutative. Therefore, all that we have said for Petri nets and linear logic holds *mutatis mutandis* for phrase-structure grammars and the linear conjunction fragment of *noncommutative* propositional linear logic [40]. In this light, grammars can be regarded as logical theories in the special case of rewriting logic provided by noncommutative conjunctive linear logic, and also as concurrent systems whose states are words and therefore have a "linearly distributed" state structure—in the same way as Petri net states have a "multiset-distributed" structure thanks to their *ACI* axioms.

Since we have already pointed out that the rules of rewriting logic should be understood as rules to reason about change in a concurrent system, the above remarks clarify the relationship between rewriting logic and linear logic, which is also a logic of action and becoming [40]. Both logics coincide in their goal of dealing with concurrency, but rewriting logic provides greater generality for achieving this purpose. This generality comes from allowing the logical connectives Σ and the structural axioms E to vary as flexible parameters that can be instantiated to fit the problem in question. This endows rewriting logic with a virtually unlimited capacity for *structuring a distributed state* in many more ways than just as a *multiset* or as a *string*. This capacity is very valuable, because the lack of structuring mechanisms in many concurrency models is one of the main obstacles blocking their application to real problems.

What about the other connectives of linear logic? As Martí-Oliet and I have argued in [85], they do not describe real states, but idealized "Gedanken" states and therefore belong to a logic suitable for specifying properties of a concurrent system rather than to what might be called the *intrinsic* logic of the system. There is in principle no reason why connectives generalizing

those of linear logic could not be added to rewriting logic for specification purposes, and indeed this seems a very interesting area for future research.

5.3.3. The chemical abstract machine and CCS

The *chemical abstract machine*, or *cham* [16], is a model of concurrency recently introduced by Gérard Berry and Gérard Boudol. The basic metaphor giving its name to the model is that of conceiving a distributed state as a "solution" in which many "molecules" float, and thinking of concurrent transitions as "reactions" that can occur simultaneously in many points of the solution.* It is possible to define a variety of chemical abstract machines. Each of them corresponds to a rewrite theory in our sense, and all of them satisfy certain common conditions characterizing them, which I explain below.

There is a common syntax shared by all chemical abstract machines, with each machine possibly extending the basic syntax by additional function symbols. The common syntax is typed, and can be expressed as the following order-sorted signature Ω:

sorts Molecule, Molecules, Solution.
subsorts Solution $<$ *Molecule* $<$ *Molecules.*
op $\lambda : \rightarrow$ *Molecules.*
op $\_,\_ :$ *Molecules Molecules* $\rightarrow$ *Molecules.*
op $\{\!|\_|\!\} :$ *Molecules* $\rightarrow$ *Solution.* `*** membrane operator`
op $\_ \triangleleft \_ :$ *Molecule Solution* $\rightarrow$ *Molecule.* `*** airlock operator`

In our terminology, we can describe a *cham* as a rewrite theory $\mathcal{C} = (\Sigma, ACI, L, R)$, with $\Sigma \supseteq \Omega$, together with a partition

$$R = \mathit{Reaction} \uplus \mathit{Heating} \uplus \mathit{Cooling} \uplus \mathit{AirlockAx}.$$

The *ACI* axioms are asserted of the operator $\_,\_$ with identity λ. The rules in R may involve variables, but are subject to certain syntactic restrictions that guarantee an efficient form of *ACI* matching. *AirlockAx* is the bidirectional sequent[30] $\{\!|m, M|\!\} \leftrightarrow \{\!|m \triangleleft \{\!|M|\!\}|\!\}$, where m is a variable of sort *Molecule* and M a variable of sort *Molecules*. The purpose of this axiom is to choose one of the molecules m in a solution as a candidate for reaction with other molecules outside its membrane. The *Heating* and *Cooling* rules can typically be paired, with each sequent $[t] \rightarrow [t'] \in \mathit{Heating}$ having a symmetric sequent $[t'] \rightarrow [t] \in \mathit{Cooling}$, and viceversa, so that we can view them as a single set of bidirectional sequents *Heating–Cooling*.

**Note added in proof.* The cham's chemical metaphor and its multiset rewriting approach are inspired by the GAMMA language of Banâtre and Le Mètayer (cf. [11]).

[30]Which is of course understood as a pair of sequents, one in each direction.

The paper of Berry and Boudol [16] makes a distinction between *rules*, which are rewrite rules specific to each cham—and consist of the *Reaction*, *Heating*, and *Cooling* rules—and *laws* which are general properties applying to all chams for governing the admissible transitions. The first three laws, the *Reaction*, *Chemical* and *Membrane* laws, correspond in our terms to saying that the machine evolves by *ACI*-rewriting. The fourth law states the axiom *AirlockAx*. The *Reaction* rules are the heart of the cham and properly correspond to state transitions. The rules in *Heating–Cooling* express *structural equivalence*, so that the *Reaction* rules may apply after the appropriate structurally equivalent syntactic form is found. A certain strategy is typically given to address the problem of finding the right structural form, for example to perform "heating" as much as possible. The present framework suggests a way of abstracting the structural equivalence problem in a cham, namely to alternatively view a cham as a rewrite theory $\mathcal{C} = (\Sigma, ACI \cup Heating\text{–}Cooling \cup AirlockAx, L, Reaction)$, in which the *Heating–Cooling* rules and the *AirlockAx* axiom have been made part of the theory's axioms.

This notion has been applied to the semantics of CCS [99], which is described as a particular cham. Berry and Boudol point out that their description is considerably simpler than that afforded using Plotkin's structural operational semantics [108]. The authors have also used this notion to define a concurrent λ-calculus [16]. Stimulated by cham ideas, recent work by Milner has also used ideas of rewriting modulo axioms to provide a compact formulation of his π-calculus [100].

5.3.4. Concurrent object-oriented programming

Concurrent object-oriented programming is a very active area of research. Beyond the fluctuations of fashion, a key reason for this interest goes back to the seminal ideas embodied in the Simula language [30], since this language was designed with the explicit intention of being used for simulating concurrent interactions between objects in the real world. Due to the limitations of a sequential implementation, at the time when Simula was designed this could only be realized in a limited way.

In spite of the recent activity to fully integrate concurrency within the object-oriented programming paradigm (see, e.g., [125,4] and recent OOPSLA proceedings) the field of concurrent object-oriented programming seems at present to lack a clear agreed upon semantic basis. Such a basis is important as a means of communication between different researchers, who can be sure of agreeing on the same concepts when using the same words; besides, a mathematically precise semantics is an absolute prerequisite for formal reasoning about concurrent object-oriented programs.

This section presents a logical theory of concurrent objects that addresses

this conceptual need in a very direct way. The semantics proposed expresses concurrent object-oriented programming in terms of concurrent *ACI*-rewriting. The reader is referred to [92] for a fuller account of this theory. Here we discuss the most basic ideas about objects and the evolution of an object-oriented system by concurrent rewriting of its configuration, which is made up of a collection of objects and messages. In addition, we also discuss how this theory provides an abstract semantics for Actors [3]. In spite of previous formalization efforts [2,23], actors have not, in my opinion, received a treatment at an abstract enough level. Actually, the many details involved in the usual descriptions can become a real obstacle for gaining a clear mathematical understanding of actors. We also discuss how the present theory of concurrent objects relates in a very simple and direct way to the very important work of Chandy and Misra on UNITY [21], specifically to their programming model, which can be obtained as an special case of the theory. Another model that can be obtained as a particular case but whose discussion is omitted is Engelfriet et al.'s POPs higher level Petri nets [36,37].

An *object* can be represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object's name, belonging to a set *OId* of *object identifiers*, C is its class, the a_i's are the names of the object's *attributes*, and the v_i'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: ∅] .
```

where the operator __ is associative and commutative with identity $\emptyset$ and plays a role entirely similar to that played by the operator $\otimes$ for Petri nets. The system evolves by concurrent rewriting (modulo *ACI*) 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.

Intuitively, we can think of messages as "travelling" to come into contact with the objects to which they are sent and then causing "communication events" by application of rewrite rules. In the model, this travelling is accounted for in a very abstract way by the *ACI* 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. Such abstraction from implementation details makes possible high level reasoning about concurrent object-oriented programs and

```
omod ACCOUNT is
  protecting INT .
  class Accnt .
  att bal : Accnt -> 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
```

Fig. 8.

their semantics without having to go down into the specific details of how communication is actually implemented.

In Maude, concurrent object-oriented systems can be defined by means of *object-oriented modules*—introduced by the keyword `omod`—using a syntax more convenient than that of system modules because it assumes acquaintance with basic entities such as objects, messages and configurations, and supports linguistic distinctions appropriate for the object-oriented case. For example, the `ACCOUNT` object-oriented module given in Fig. 8 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 transfering funds betwen two accounts. We assume that a functional module `INT` for integers with a subsort relation `Nat < Int` and an ordering predicate `_>=_` is available.

In this example, the only attribute of an account—introduced by the keyword `att`—is its `bal(ance)`, which 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. The commutative monoid structure of the configuration provides the top level distributed structure of the system and allows concurrent application of the rules. For example, Fig. 9 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, as already mentioned, 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 specified in this way.

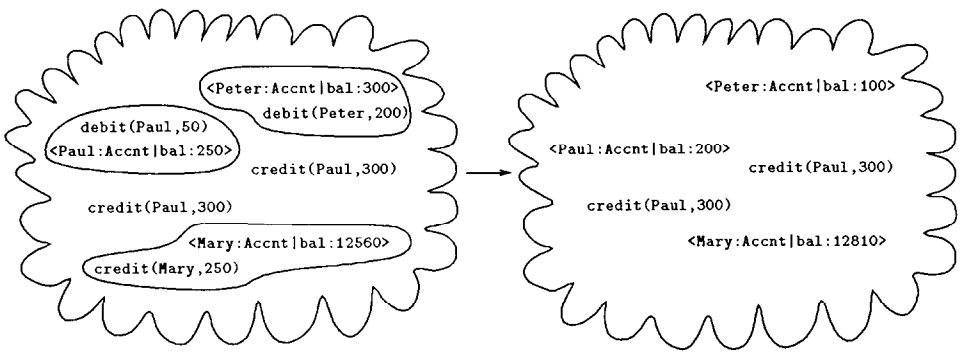


Fig. 9. Concurrent rewriting of bank accounts.

Although Maude's object-oriented modules provide a convenient syntax for programming object-oriented systems, their semantics can be *entirely reduced* to that of system modules, i.e., we can regard the additional syntax as syntactic sugar and nothing else. In fact, each object-oriented module can be translated into a corresponding system module whose semantics *is* by definition that of the original object-oriented module. For example, the system module ACCOUNT# given in Fig. 10 is the translation of the ACCOUNT module above (we assume an already existing functional module ID of identifiers containing a sort OId of object identifiers).

Note the introduction of operators bal._replyto_ and bal._is_to_ cor-

```
mod ACCOUNT# is
  protecting INT ID .
  subsorts Accnt Msg < Configuration .
  ops credit,debit : OId Nat -> Msg .
  op transfer_from_to_ : Nat OId OId -> Msg .
  op bal._replyto_ : OId OId -> Msg .
  op bal._is_to_ : OId Nat OId -> Msg .
  op <_: Accnt | bal:_> : OId Nat -> Accnt .
  op __ : Configuration Configuration ->
       Configuration [assoc comm id: ∅] .
  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 .
  rl (bal. A replyto B) < A : Accnt | bal: N > =>
       < A : Accnt | bal: N > (bal. A is N to B) .
endm
```

Fig. 10.

responding to messages by which an object can request the balance of a given account and receive a reply from the account as specified by the last rewrite rule. This capability is built in for object-oriented modules unless an attribute has been declared `hidden`, and therefore was not mentioned in the original `ACCOUNT` module.

In Maude, the general form required of rewrite rules used to specify the behavior of an object-oriented system is as follows:

$$
\begin{aligned}
& M_1 \ldots M_n \langle O_1 : C_1 \mid attrs_1 \rangle \ldots \langle O_m : C_m \mid attrs_m \rangle \\
& \quad \rightarrow \langle O_{i_1} : C'_{i_1} \mid attrs'_{i_1} \rangle \ldots \langle O_{i_k} : C'_{i_k} \mid attrs'_{i_k} \rangle \\
& \qquad \langle Q_1 : D_1 \mid attrs''_1 \rangle \ldots \langle Q_p : D_p \mid attrs''_p \rangle \\
& \qquad M'_1 \ldots M'_q \\
& \qquad \textit{if } C
\end{aligned}
\tag{†}
$$

where the Ms are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is the rule's condition. A rule of this kind expresses a *communication event* in which n messages and m distinct objects participate. The *outcome* of such an event is as follows:

- the messages $M_1, \ldots, M_n$ disappear;
- the *state* and possibly even the *class* of the objects $O_{i_1}, \ldots, O_{i_k}$ may change;
- all other objects O_j vanish;
- new objects $Q_1, \ldots, Q_p$ are created;
- new messages $M'_1, \ldots, M'_q$ are sent.

There are also some additional requirements to ensure the proper behavior of the rules (†) that are discussed in [92]. Notice that, since some of the attributes of an object—as well as the parameters of messages—can contain object names, very complex and dynamically changing patterns of communication can be achieved by rules of this kind.

We call a communication event (and therefore its corresponding rule) *asynchronous* if only one object appears in the lefthand side. Otherwise, the communication is called *synchronous* and the objects appearing in the lefhand side are said to *synchronize* in the event. For example, the rules for crediting and debiting accounts describe asynchronous communication events, whereas the rule for transfering funds between two accounts forces them to synchronize.

We refer the reader to [92] for a more complete treatment of object-oriented concurrency in Maude, and for more examples. In particular, an important topic treated there is the *creation and deletion* of objects, which can also be treated by concurrent *ACI*-rewriting in a variety of ways and

without any problems. Object creation is typically initiated by means of a "new" message of the form

$$new(C \mid attrs)$$

which specifies the new object's class and initialization values for its attributes and has the effect of creating a new object with those properties and with a fresh new name.

In comparison with the FOOPS language that Joseph Goguen and I developed in [48] and that has provided very valuable experience in the design of Maude, both FOOPS and Maude contain OBJ as their functional sublanguage, and both have declarative style object-oriented modules; also, the idea of transforming objects by rewrite rules goes back to [49], although the use of *ACI* to treat concurrency was not contemplated in that work. However, the semantic frameworks of FOOPS and Maude are quite different, and this leads to different styles of programming with objects and to different approaches to concurrency.

An interesting topic, unfortunately beyond the scope of this paper, is making a precise comparison of the present theory with other very recent proposals to give a formal basis to object-oriented concurrency such as Goguen's sheaf-theoretic semantics [43], and the work of Sernadas et al. [119] among others.

Actors. Actors [3,2] provide a flexible and attractive style of concurrent object-oriented programming. However, their mathematical structure, although already described and studied by previous researchers [23,2], has remained somewhat hard to understand and, as a consequence, the use of formal methods to reason about actor systems has remained limited. The present logical theory of concurrent objects sheds new light on the mathematical structure of actors and provides a new formal basis for the study of this important and interesting approach.

Specifically, the general logical theory of concurrent objects presented in this paper yields directly as a special case an entirely declarative approach to the theory and programming practice of actors. The specialization of our model to that of actors can be obtained by first clarifying terminological issues and then studying their definition by Agha and Hewitt [3].

Actor theory has a terminology of its own which, to make things clearer, I will attempt to relate to the more standard terminology employed in object-oriented programming. To the best of my understanding, the table in Fig. 11 provides a basic terminological correspondence of this kind.

The essential idea about actors is clearly summarized in the words of Agha and Hewitt [3] as follows:

Actors	OOP
Script	Class declaration
Actor	Object
Actor Machine	Object State
Task	Message
Acquaintances	Attributes

Fig. 11. A dictionary for Actors.

"An actor is a computational agent which carries out its actions in response to processing a communication. The actions it may perform are:
- Send communications to itself or to other actors.
- Create more actors.
- Specify the *replacement behavior*."

The "replacement behavior" is yet another term to describe the new "actor machine" produced after processing the communication, i.e., the new state of the actor.

We can now put all this information together and simply conclude that a logical axiomatization in rewriting logic of an actor system—which is of course at the same time an *executable* specification of such a system in Maude—exactly corresponds to the special case of a concurrent object-oriented system in our sense whose rewrite rules instead of being of the general form (†) are of the special asynchronous and unconditional form

$$\begin{aligned} & M\ \langle O : C \mid attrs \rangle \\ & \quad \rightarrow \langle O : C' \mid attrs' \rangle \\ & \qquad \langle Q_1 : D_1 \mid attrs_1'' \rangle \ldots \langle Q_p : D_p \mid attrs_p'' \rangle \\ & \qquad M_1' \ldots M_q'. \end{aligned}$$

Therefore, the present theory is considerably *more general* than that of actors. In comparison with existing accounts about actors [3,2] it seems also fair to say that our theory is *more abstract* so that some of those accounts can now be regarded as *high level architectural descriptions* of ways in which the abstract model can be implemented. In particular, the all-important *mail system* used in those accounts to buffer communication is the implementation counterpart of what in our model is abstractly achieved by the *ACI* axioms. Another nice feature of our approach is that it gives a *truly concurrent* formulation—in terms of concurrent *ACI*-rewriting—of actor computations, which seems most natural given their character. By contrast, Agha [2] presents an interleaving model of sequentialized transitions. Agha is keenly aware of the inadequacy of reducing the essence of true concurrency to nondeterminism and therefore states (on p. 82) that the correspondence between his interleaving model and the truly concurrent computation of actors is "*representationalistic, not metaphysical*".

There is one additional aspect important for actor systems and in general for concurrent systems, namely *fairness*. For actors, this takes the form of requiring *guarantee of mail delivery*. In the concurrent rewriting model it is possible to state precisely a variety of fairness conditions and, in particular, the guarantee of mail delivery for the special case of actors. Details will appear elsewhere.

UNITY. UNITY [21] is an elegant and important theory of concurrent programming with an associated logic to reason about the behavior of concurrent programs that has been developed by Chandy and Misra. We show below how the present logical theory of concurrent objects yields UNITY's model of computation as a special case in a direct way. The discussion is restricted to the programming language and its corresponding model of program execution; UNITY's logic is not discussed.

In essence[31] a UNITY program consists of:

- a declaration of variables with their corresponding type;
- a declaration of initial values for some of those variables;
- a set of multiple assignment statements.

A *multiple assignment statement* is a construct of the form:

$$x_1, \ldots, x_n := exp_1(x_1, \ldots, x_n), \ldots, exp_n(x_1, \ldots, x_n)$$

where the x_i are declared variables and the $\exp_i(x_1, \ldots, x_n)$ are Σ-terms for Σ a fixed many-sorted signature defined on the types of the declared variables. The intuitive meaning of executing such an assignment is that all the variables x_i are *simultaneously* assigned the values that their corresponding expression $exp_i(x_1, \ldots, x_n)$ evaluate to. Beginning with a state that satisfies the declaration of initial values, the *execution* of a UNITY program proceeds by choosing an arbitrary multiple assignment in the set and executing it. This process of choosing and executing assignments is continued *ad infinitum*, but a *fair choice* is assumed, i.e., all assignments are executed infinitely often in such an infinite computation. Therefore, UNITY's model of program execution is an interleaving semantics that assigns to each program the set of all fair execution sequences of the form

$$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \cdots S_n \xrightarrow{a_{n+1}} S_{n+1} \xrightarrow{a_{n+2}} \cdots$$

where s_0 is a state satisfying the declaration of initial values and each $S_n \xrightarrow{a_{n+1}} S_{n+1}$ is a transition from state s_n to the state s_{n+1} reached from s_n

[31] I.e., any UNITY program can be expressed in this form after doing away with conditional assignments (which can be handled as unconditional ones by means of a generalized many-argument if-then-else operator) and expanding out all quantified assignments.

by executing a multiple assignment a_{n+1} chosen among those appearing in the program.

In our model, a program of this kind has a simple formalization as a rewrite theory $\mathcal{R} = (\Sigma, ACI, L, R)$ corresponding to a Maude object-oriented module where the objects have only one attribute and are of the form $\langle x : T \mid val : v \rangle$ with T a type (sort) name and v a value of that type.

The variable declaration just means that we are only considering environments made up of exactly those objects whose identifiers have been declared. In UNITY there is no deletion of objects; also, all existing objects have been created at the very beginning of the computation in a state partially[32] specified by the initialization declaration. No new objects are created afterwards. In the environment there are *no messages*, only objects. Such an environment can therefore be understood as a set of objects—one for each declared variable—and formalizes a possible *state* of the program in question. Each multiple assignment

$$x_1, \ldots, x_n := exp_1(\overline{x}), \ldots, exp_n(\overline{x})$$

yields a rule of R, namely the rule:

$$\langle x_1 : T_1 \mid val : v_1 \rangle \ldots \langle x_n : T_n \mid val : v_n \rangle$$
$$\rightarrow \langle x_1 : T_1 \mid val : exp_1(\overline{v}) \rangle \ldots \langle x_n : T_n \mid val : exp_n(\overline{v}) \rangle$$

which when applied has the effect of *synchronizing* the objects $x_1, \ldots, x_n$. Evolution of the system is of course by concurrent rewriting which, by Lemmas 2.6 and 3.6, can always be expressed by means of interleaving sequences as done in [21], but in our model it has a more appealing form without recourse to such sequences.

5.4. *The Big Picture*

We can pause for a moment and discuss what the concurrent rewriting model does in fact accomplish. Figure 12 provides a good summary of the situation and can help us in this task. At the top of the figure we have the concurrent rewriting model, denoted by the acronym *CR*. The arrows in the tree stand for specializations of the model; first, to the three main cases that we have considered in detail, i.e., to $E = \emptyset, AI, ACI$ respectively, and then to subcases under each of those cases.

One interesting observation is that the two types of unification that we were seeking, i.e., the *internal* unification of concurrency models and the *external* unification of concurrency with other programming language paradigms can both be perceived in the picture.

[32] Since not all declared variables need appear in the initialization declaration.

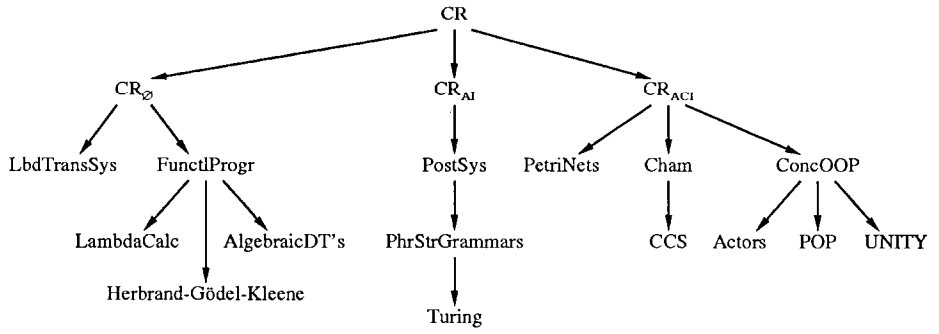


Fig. 12. Specializations of concurrent rewriting.

Regarding the internal unification, we can see how a heterogeneous variety of well-known concurrency models are *specializations* of the concurrent rewriting model by imposing additional restrictions on it. The picture is not at all complete; a notable omission—among several others—is the absence of CSP [59]; also, two of the specializations, to CCS and to POPs, have not really been discussed; for CCS we implicitly rely on the simulation already developed by Berry and Boudol in [16] which can be easily adapted to the present framework, and for Engelfriet et al.'s POPs [36,37] we have already mentioned that the details are omitted. Doing justice to all the relevant models would require a much lengthier exposition, one beyond the scope of the present work. I hope that the models already discussed give a general flavor and offer sufficient enough evidence—even if incomplete—for the possibilities of conceptual unification that the concurrent rewriting model contains.

The external unification deserves some discussion. First, note that functional programming appears as the special case in which the rules are (ground) Church–Rosser. In this way, functional programming is integrated as a special case with additional properties and with a more restricted denotational semantics. Indeed, as already pointed out, this unification is intimately connected with the map of logics

$$\mathit{RewritingLogic} \longrightarrow \mathit{EquationalLogic}$$

discussed in Section 3.6 that relates formally the deduction and model theories of rewriting logic and equational logic. Thanks to this map, we obtain a simple integration of two logic programming languages in the general axiomatic sense of [91], namely a functional programming language based on equational logic, which in Maude corresponds to the sublanguage of functional modules isomorphic to OBJ3 [46], and a concurrent systems programming language directly based on rewriting logic, which in Maude corresponds to the language of system modules.

Concurrent object-oriented programming then appears naturally as a special case of rewriting logic programming involving *ACI*-rewriting. This case is quite common and important and therefore in Maude is given a special syntax—Maude's object-oriented modules. However, this syntax is only syntactic sugar for an equivalent expression as a system module, i.e., as a rewrite theory.

In summary, Maude provides a simple unification of functional programming with a general version of concurrent system's programming that includes concurrent object-oriented programming as a particular instance. Besides, this unification is accomplished entirely within logic so that the entire language is declarative and specifically is a logic programming language in the general axiomatic sense of [91]. Intuitively, this means that programs are theories, computation is deduction, denotational semantics is given by initial models, and there is a nice agreement between facts holding in the initial model and queries answerable by deduction.

This seems to leave out Horn clause logic programming, including concurrent versions [120], which together with functional and object-oriented programming is among the most promising programming language paradigms. However, this omission is only apparent. Actually, Horn clause logic programming appears also very naturally as a special case of rewriting logic programming. By "very naturally" I mean that there is a map of logics

$$\mathit{HornLogic} \longrightarrow \mathit{RewritingLogic}$$

that systematically relates Horn logic to rewriting logic. However, rewriting logic—while keeping the same syntax and models—now has to consider also queries involving existential formulas of the form

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

A detailed discussion of the integration of this additional paradigm within rewriting logic and of the role of unification in the extended operational semantics will appear elsewhere.

This work has emphasized logic and models; it has not addressed questions of implementation. However, a few remarks on this important matter seem appropriate. Although concurrent rewriting is a general and flexible model of concurrency and can certainly be used to reason formally about concurrent systems at a high level of abstraction, it would not be reasonable to implement this model for programming purposes in its fullest generality. This is due to the fact that, in its most general form, rewriting can take place *modulo* an arbitrary equational theory E. As pointed out in Section 2.3 a minimum requirement for E is the existence of an algorithm for finding all the matches modulo E for a given rule and term; however, for some axioms E this process, even if it is available, can be quite inefficient, so that its

implementation should be considered a theorem proving matter and should not be made part of a programming language implementation. However, $CR_{\emptyset}$ can be implemented very efficiently in parallel and is indeed an ideal model of parallel computation on which to base the design of a massively parallel machine architecture; in particular, the Rewrite Rule Machine directly implements this case [47,5]. Other cases can also be very efficient. For example, an important research topic that we are presently investigating is the design of additional architectural mechanisms to support the efficient cases of *ACI*-rewriting which naturally occur in practice, such as concurrent object-oriented programming.

In this regard, it is useful to adopt a *transformational* point of view. For specification purposes we can allow the full generality of the concurrent rewriting model, whereas for programming purposes we should study subcases that can be efficiently implemented. Then, we should develop program transformation techniques that are semantics-preserving and move us from specifications to programs, and from less efficient programs to more efficient ones. This, indeed, seems a practical and important topic for future research. *

6. Related work and concluding remarks

There is a wealth of work on term rewriting, concurrency, Petri nets, linear logic, equational logic and the theory of abstract data types relevant to this paper; a good part of the most closely related work has already been mentioned in the body of the paper. However, now that the main ideas of this work have been explained, I would like to discuss—without attempting in any way to be exhaustive—other work by a variety of authors that bears some relationship to the work presented here. I discuss connections with Plotkin's structured operational semantics, with work in term rewriting, and with work on 2-categories. I also discuss future developments of the present work and relevant literature in this regard. Finally, some concluding remarks summarize some of the main ideas presented.

Structural operational semantics

Specifications using conditional rewrite rules are quite similar to specifications using Plotkin's structural operational semantics (SOS) [108]. Specifically, the case of syntactic rewriting ($E = \emptyset$) closely resembles SOS. However, the flexibility of rewriting modulo structural axioms E probably

**Note added in proof.* A subset of Maude, called Simple Maude, that can be implemented efficiently in a wide variety of parallel architectures and can serve as the target language for program transformations of this kind, is discussed in Meseguer and Winkler [98].

leads to considerably greater expressiveness and more compact axiomatizations. Of course, all this should be further investigated through examples; in particular, systems supporting SOS specification and prototypig such as [22] should be compared with Maude. A key difference between SOS and rewriting logic is that SOS has a proof theory, but—except for the recent proposal of Badouel [10]—has not had a model theory. As a consequence of this situation, operational and denotational approaches to concurrency have lived somewhat separated lives, lacking a common semantics ground on which to be related. By contrast, rewriting logic provides a wide spectrum of classes of models ranging from models quite operational in style such as $\mathcal{T}_{\mathcal{R}}$ to poset and domain-theoretic models, and even to classical algebra models which are still useful for functional concurrent systems; besides, these different semantic choices are systematically related by adjunctions. In this way, the operational and denotational semantics of a concurrent system can be connected within a single model-theoretic framework; in Maude this takes the form of an abstraction homomorphism linking both semantics.

Term rewriting

Within the area of term rewriting, the work of Huet is close in spirit to the approach taken here. In the lecture notes [60], Huet defined a category of rewritings for *regular* rewrite theories extending previous joint work with J.-J. Lévy and earlier ideas by Lévy [82] connected with the "parallel moves lemma" of the lambda calculus [29] (see also [76]). In [60], Huet also briefly discussed rules for a nonconditional version of rewriting logic. Using ideas about residuals originating in the Huet–Lévy work, Stark has developed an elegant categorical model of transition systems exhibiting concurrency, and has applied that model to obtain results on dataflow nets [121]. Another important generalization of the Huet–Lévy work reaching beyond the Church–Rosser case and explicitly addressing nondeterminism is the work already mentioned by Boudol [18].

2-Categories

There is also important work on applications of 2-categories to rewriting, including work by Rydeheard and Stell [116], who for the unconditional case with $E = \emptyset$ constructed a 2-category of term rewritings, and Seely [117], who used 2-categories to treat reduction in lambda calculi. Unpublished work of Pitts [107] brought to my attention by Poigné applies 2-category ideas to the semantics of fixpoints in denotational semantics, both at the level of functions and at the level of solving domain equations; in a sense, his work generalizes to the category-enriched setting previous work on continuous and rational theories. Independently of my own work, Pitts has also developed in [107] a logic of fixpoints, whose fixpoint-free fragment is very closely

related to the unconditional fragment of the rewriting logic presented here. Power [111] has recently studied normalization of what we here call "proof expressions" by means of pasting techniques in 2-categories. Very recent work by Pratt [112] uses n-categories to describe concurrent computations. Several of these approaches have focused on proof-theoretic results, using 2-categories as theories; by contrast, the main focus of this work has been the model theory of rewriting logic and its relationship to concurrency. Although this paper has avoided using the heavier machinery of 2-categories, they are very useful, and Appendix A of [94] develops both the proof theory and the model theory of a 2-categorical approach to rewriting that connects with the 2-categorical work already mentioned, extends already known results to the more complex conditional case, and provides a new perspective from which to view the results and ideas of this paper. An important fruit of the 2-categorical ideas presented in Appendix A of [94] is a theory of morphisms between rewrite theories—developed in Appendix B of the same paper—that generalizes the implementation morphisms of Petri nets introduced in [96,97] and permits treating parameterized rewrite theories.

Further developments

It would be nice to further develop the present theory and its applications in several directions. First, rewriting logic should be embedded within a richer logic to be used for specification purposes; this corresponds to Maude's system and object-oriented *theories*. Parameterization issues, already initiated in [94], should be further studied, and more experience with examples in Maude should be gained. The issue of program transformations to derive more efficient programs, and the related matter of compilation techniques—specially in the context of parallel implementations—are also important topics that need further study.

Maude's multiparadigm capabilities should also be further extended. As already mentioned, the inclusion of Horn clause logic programming is based on a map of logics

$$\mathit{HornLogic} \longrightarrow \mathit{RewritingLogic}.$$

This approach should be compared in detail with work on concurrent logic programming [120] and with other recent proposals to provide a semantic basis for concurrent Horn clause programming such as those of Corradini and Montanari [24], and Andreoli and Pareschi's work on linear logic programming [7]. Yet another possible extension is adding higher order types. Higher order capabilities already exist within the present framework; for example, a lambda calculus with explicit substitution can be defined as a Maude functional module, and Maude's parameterized modules provide important higher order capabilities. However, more explicit higher order

aspects seem worth investigating. In this regard, the 2-categorical work of Seely [117] as well as recent joint work with Narciso Martí-Oliet on a higher order extension of order-sorted equational logic [86] could serve as a basis. Comparisons should also be made with recent higher order approaches to concurrency, including the work of Boudol [19], Berry and Boudol [16], and Milner, Parrow and Walker [101].

Concluding remarks

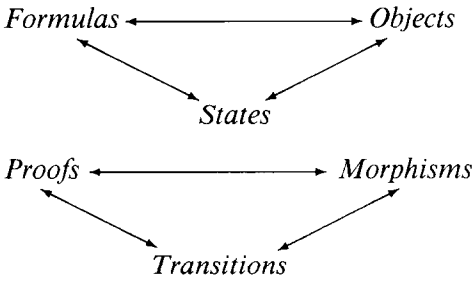
The present work has provided a unification of a wide variety of models of concurrency, which can be obtained by specialization from the concurrent rewriting model. In addition, this model unifies concurrent programming with functional and object-oriented programming in a simple way as shown in Maude, a declarative language for programming concurrent systems that contains OBJ3 as a functional sublanguage and also supports concurrent object-oriented programming. Maude examples have illustrated how rewrite theories can be used directly as a logic programming language to program concurrent systems. Maude's operational and denotational semantics have been defined in terms of the model theory of rewriting logic.

A key characteristic of this model of concurrent computation is that it is directly based on rewriting logic. This provides a conceptual reduction of all the above paradigms and models of concurrency to logic programming in the general sense made precise in [91]. In particular, the traditional dichotomy between procedural and declarative programming is eliminated; this is specially apparent in the new logical theory of concurrent objects presented in this paper.

We have shown that rewriting logic is very well suited for computing with concurrent systems, and have proposed $\mathcal{R}$-systems as a general mathematical notion of concurent system. We have also shown that rewriting logic generalizes the proof theory and the model theory[33] of several quite different and important logics currently used in computer science for quite different purposes. Such logics include: equational logic—which is used for functional programming and for abstract data types among other topics—, conjunctive linear logic—which has been used in connection with Petri nets and in a variety of other applications—, and the logic of inequalities—which is extensively used in denotational semantics and in algebraic approaches to nondeterminism. In fact, the unification of all those logics presented in this paper establishes a variety of suggestive new connections among those different fields, connections that would be worth investigating further.

[33]The only case in which this has not been made explicit is conjunctive linear logic. It is nevertheless true also in that case, for which the appropiate models are *symmetric monoidal categories* [118,77,85], i.e., categories with a commutative monoid structure (perhaps up to coherence).

The model-theoretic semantics developed for rewriting logic gives rise to a general *triangular correspondence* between logic, categories and concurrency that can be summarized as follows:



This correspondence generalizes to arbitrary rewrite theories the triangular correspondence between linear logic, Petri nets and linear categories previously developed in joint work with Narciso Martí-Oliet [85]. In particular, the correspondence between logic and categories is a Lambek–Lawvere correspondence [79,80], a type of correspondence more abstract and general than the Curry–Howard isomorphism.

Acknowledgement

I specially thank Prof. Joseph Goguen for our long term collaboration on the OBJ and FOOPS languages [46,48], concurrent rewriting [47] and its implementation on the RRM architecture [49,5], all of which have directly influenced this work; he has also provided many positive suggestions for improving a previous version of this paper. I specially thank Prof. Ugo Montanari for our collaboration on the semantics of Petri nets [96,97] which has been a source of inspiration for the ideas presented here. Mr. Narciso Martí-Oliet deserves special thanks for our collaboration on the semantics of linear logic and its relationship to Petri nets [85,84], which is another source of inspiration for this work; he also provided very many helpful comments and suggestions for improving the exposition. I also thank all my fellow members of the OBJ and RRM teams, past and present, and in particular Dr. Claude Kirchner, Dr. Sany Leinwand, Mr. Patrick Lincoln and Mr. Timothy Winkler, who deserves special thanks for his many very good comments about the technical content as well as for his kind assistance with the pictures. I also wish to thank Prof. Samson Abramsky, Prof. Gérard Berry, Prof. Pierre-Louis Curien, Prof. Pierpaolo Degano, Prof. Brian Mayoh, Prof. Robin Milner, Dr. Mark Moriconi, Prof. Peter Mosses and Dr. Axel Poigné, all of whom provided helpful comments and encouragement.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien and J.-J. Lévy, Explicit substitution, in: *Proc. POPL '90* (ACM, New York, 1990) 31–46.

[2] G. Agha, *Actors* (MIT Press, Cambridge, MA, 1986).

[3] G. Agha and C. Hewitt, Concurrent programming using actors, in: A. Yonezawa and M. Tokoro, eds., *Object-Oriented Concurrent Programming* (MIT Press, Cambridge, MA, 1988) 37–53.

[4] G. Agha, P. Wegner and A. Yonezawa, eds, *Proc. ACM–SIGPLAN Workshop on Object-Based Concurrent Programming* (ACM, New York, 1989); *Sigplan Notices* (April 1989).

[5] H. Aida, J. Goguen and J. Meseguer, Compiling concurrent rewriting onto the rewrite rule machine, Technical Report SRI-CSL-90-03, SRI International, Computer Science Lab, February 1990, in: S. Kaplan and M. Okada, eds., *Proc. Internat. Workshop on Conditional and Typed Rewriting Systems, Montreal, Canada, June 1990*, Springer Lecture Notes in Computer Science, to appear.

[6] H. Aida, S. Leinwand and J. Meseguer, Architectural design of the rewrite rule machine ensemble, in: J. Delgado-Frias and W.R. More, eds., *Proc. Workshop on VLSI for Artificial Intelligence and Neural Networks, Oxford, September 1990*; also, Tech. Report SRI-CSL-90-17, December 1990.

[7] J.-M. Andreoli and R. Pareschi, LO and behold! concurrent structured processes, in *ECOOP-OOPSLA '90 Conf. on Object-Oriented Programming, Ottawa, Canada, October 1990* (ACM, New York, 1990) 44–56.

[8] A. Appel and D. MacQueen, A standard ML compiler, in: *Proc. Conf. Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 274 (Springer, Berlin, 1987) 301–324.

[9] A. Asperti, A logic for concurrency, Unpublished manuscript, November 1987.

[10] E. Badouel, Conditional rewrite rules as an algebraic semantics of processes, Technical Report 1226, INRIA, May 1990.

[11] J.-P. Banâtre and D. Le Mètayer, The GAMMA model and its discipline of programming, *Sci. Comput. Programming* **15** (1990) 55–77.

[12] H.P. Barendregt, *The Lambda Calculus, its Syntax and Semantics* (North-Holland, Amsterdam, 1984).

[13] M. Barr and C. Wells, *Toposes, Triples and Theories* (Springer, Berlin, 1985).

[14] M. Bauderon and B. Courcelle, Graph expressions and graph rewriting. *Math. Systems Theory* **20** (1987) 83–127.

[15] 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., *Automata, Languages and Programming, 7th Colloquium*, Lecture Notes in Computer Science, Vol. 81 (Springer, Berlin, 1980) 76–90.

[16] G. Berry and G. Boudol, The chemical abstract machine, in: *Proc. POPL '90* (ACM, New York, 1990) 81–94.

[17] S. Bloom, Varieties of ordered algebras, *J. Comput. System Sci.* **13** (1976) 200–212.

[18] G. Boudol, Computational semantics of term rewriting systems, in: M. Nivat and J. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, Cambridge, 1985) 169–236.

[19] G. Boudol, Towards a lambda calculus for concurrent and communicating systems, in: *TAPSOFT '89*, Lecture Notes in Computer Science, Vol. 351 (Springer, Berlin, 1989) 149–161.

[20] G. Boudol and I. Castellani, Permutation of transitions: an event structure semantics for CCS and SCCS, in: J.W. de Bakker, W.-P. de Roever and G. Rozenberg, eds., *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science, Vol. 354 (Springer, Berlin, 1988) 411–427.

[21] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison-Wesley, Reading, MA, 1988).

[22] D. Clément, J. Despeyroux, L. Hascoet and G. Kahn, Natural semantics on the computer, in: K. Fuchi and M. Nivat, eds., *Proc. France–Japan AI and CS Symposium*

(ICOT, 1986) 49–89; also, Information Processing Society of Japan, Technical Memorandum PL-86-6.

[23] W. Clinger, Foundations of actor semantics, AI-TR-633, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1981.

[24] A. Corradini and U. Montanari, An algebraic semantics of logic programs as structured transition systems, in: S. Debray and M. Hermenegildo, eds., *North American Conf. on Logic Programming* (MIT Press, Cambrigde, MA, 1990) 788–812.

[25] B. Courcelle, Infinite trees in normal form and recursive equations having a unique solution, *Math. Systems Theory* **13** (1979) 131–180.

[26] B. Courcelle and M. Nivat, The algebraic semantics of recursive program schemes, in: *Proc. Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, Vol. 64 (Springer, Berlin, 1978) 16–30.

[27] G. Cousineau, P.-L. Curien and M. Mauny, The categorical abstract machine, *Sci. Comput. Programming* **8** (1987) 173–202.

[28] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming* (Pitman, London, 1986).

[29] H.B. Curry and R. Feys, *Combinatory Logic* (North-Holland, Amsterdam, 1968).

[30] O.-J. Dahl, B. Myhrhaug and K. Nygaard, The Simula 67 common base language, Technical report, Norwegian Computing Center, Oslo, 1970, Publication S-22.

[31] P. Degano, J. Meseguer and U. Montanari, Axiomatizing net computations and processes, in: *Proc. LICS '89* (IEEE, 1989) 175–185.

[32] P. Degano, J. Meseguer and U. Montanari, Axiomatizing the algebra of net computations and processes, Technical Report SRI-CSL-90-12, SRI International, Computer Science Laboratory, November 1990, submitted.

[33] N. Dershowitz and J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (Elsevier, Amsterdam, 1990) 243–320.

[34] N. Dershowitz, S. Kaplan and D. Plaisted, Infinite normal forms, in: G. Ausiello, M. Dezani-Ciancaglini and S. Ronchi Della Rocca, eds., *Proc. 16th Internat. Conf. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 372 (Springer, Berlin, 1989) 249–262.

[35] H. Ehrig, Introduction to the algebraic theory of graph grammars, in: V. Claus, H. Ehrig and G. Rozenberg, eds., *Graph-Grammars and their Application to Computer Science and Biology*, Lecture Notes in Computer Science, Vol. 73 (Springer, Berlin, 1979) 1–69.

[36] J. Engelfriet, Net-based description of parallel object-based systems, or POTs and POPs, Technical report, Noordwijkerhout FOOL Workshop, May 1990.

[37] J. Engelfriet, G. Leih and G. Rozenberg, Parallel object-based systems and Petri nets, I and II, Technical Report 90-04-5, Dept. of Computer Science, University of Leiden, February 1990.

[38] P. Freyd, Aspects of topoi, *Bull. Austral. Math. Soc.* **7** (1972) 1–76.

[39] K. Futatsugi, J. Goguen, J.-P. Jouannaud and J. Meseguer, Principles of OBJ2, in: B. Reid, ed., *Proc. 12th ACM Symp. Principles of Programming Languages* (ACM, New York, 1985) 52–66.

[40] J.-Y. Girard, Towards a geometry of interaction, in: J.W. Gray and A. Scedrov, eds, *Proc. AMS Summer Research Conf. on Categories in Computer Science and Logic, Boulder, Colorado, June 1987* (American Mathematical Society, Providence, RI, 1989) 69–108.

[41] K. Gödel, On undecidable propositions of formal mathematical systems, in: M. Davis, ed., *The Undecidable* (Raven Press, Hewlet, NY, 1965) 39–74.

[42] J.A. Goguen, Semantic specifications for the rewrite rule machine, in: A. Yonezawa, W. McColl and T. Ito, eds., *Concurrency: Theory, Language and Architecture*, Lecture Notes in Computer Science, Vol. 491 (Springer, Berlin, 1990) 216–234.

[43] J. Goguen, Sheaf semantics for concurrent interacting objects, in: *Proc. REX School on Foundations of Object-Oriented Programming, Noorwijkerhout, The Netherlands, May 28–June 1, 1990*, to appear.

[44] J. Goguen, How to prove algebraic inductive hypotheses without induction: with applications to the correctness of data type representations, in: W. Bibel and

R. Kowalski, eds., *Proc. 5th Conf. on Automated Deduction*, Lecture Notes in Computer Science, Vol. 87 (Springer, Berlin, 1980) 356–373.

[45] J. Goguen, J.-P. Jouannaud and J. Meseguer, Operational semantics of order-sorted algebra, in: W. Brauer, ed., *Proc. 1985 Internat. Conf. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 194 (Springer, Berlin, 1985) 221–231.

[46] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer and T. Winkler, An introduction to OBJ3, in: J. -P. Jouannaud and S. Kaplan, eds., *Proc. Conf. on Conditional Term Rewriting, Orsay, France, July 8–10, 1987*, Lecture Notes in Computer Science, Vol. 308 (Springer, Berlin, 1988) 258–263.

[47] J. Goguen, C. Kirchner and J. Meseguer, Concurrent term rewriting as a model of computation, in: R. Keller and J. Fasel, eds., *Proc. Workshop on Graph Reduction, Santa Fe, New Mexico*, Lecture Notes in Computer Science, Vol. 279 (Springer, Berlin, 1987) 53–93.

[48] J. Goguen and J. Meseguer, Unifying functional, object-oriented and relational programming with logical semantics, in: B. Shriver and P. Wegner, eds., *Research Directions in Object-Oriented Programming* (MIT Press, Cambridge, MA, 1987) 417–477. Preliminary version in *SIGPLAN Notices* **21**(10) (1986) 153–162; also, Technical Report CSLI-87-93, Center for the Study of Language and Information, Stanford University, March 1987.

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

[50] J. Goguen and J. Meseguer, Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations, Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989. Given as lecture at Seminar on Types, Carnegie-Mellon University, June 1983. Submitted for publication.

[51] J. Goguen and J. Tardo, OBJ-0 preliminary users manual, Semantics and theory of computation report 10, UCLA, 1977.

[52] J. Goguen, J. Thatcher and E. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in: R. Yeh, ed., *Current Trends in Programming Methodology, IV* (Prentice-Hall, Englewood Cliffs, NJ, 1978) 80–149.

[53] J. Goguen, J. Thatcher, E. Wagner and J. Wright, Initial algebra semantics and continuous algebras, *J. ACM* **24**(1) (1977) 68–95.

[54] I. Guessarian, *Algebraic Semantics*, Lecture Notes in Computer Science, Vol. 99 (Springer, Berlin, 1981).

[55] C. Gunter and V. Gehlot, Nets as tensor theories, Technical Report MS-CIS-89-68, Dept. of Computer and Information Science, University of Pennsylvania, 1989.

[56] P.G. Harrison and M.J. Reeve, The parallel graph reduction machine Alice, in: R. Keller and J. Fasel, eds., *Proc. Workshop on Graph Reduction, Santa Fe, New Mexico*, Lecture Notes in Computer Science, Vol. 279 (Springer, Berlin, 1987) 181–202.

[57] W.H. Hesselink, A mathematical approach to nondeterminism in data types, *ACM Trans. Prog. Lang. Sys.* **10** (1988) 87–117.

[58] J.R. Hindley and J.P. Seldin, *Introduction to Combinators and λ-Calculus* (Cambridge University Press, Cambridge, 1986).

[59] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, Englewood Cliffs, NJ, 1985).

[60] G. Huet, Formal Structures for Computation and Deduction, INRIA, 1986.

[61] G. Huet, Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM* **27** (1980) 797–821; preliminary version in 18th Symposium on Mathematical Foundations of Computer Science, 1977.

[62] J. Hughes, Super-combinators: a new implementation method for applicative languages. in: *ACM Symp. on Lisp and Functional Programming* (ACM, New York, 1982) 1–10.

[63] H. Hussmann, Nondeterministic algebraic specifications and nonconfluent term rewriting, Computer Science Dept., TU Munich.

[64] D. Janssens and G. Rozenberg, Actor grammars, *Math. Systems Theory* **22** (1989) 75–107.

[65] B. Jayaraman and D. Plaisted, Functional programming with sets, in: G. Kahn, ed., *Proc. IFIP Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 274 (Springer, Berlin, 1987) 194–210.

[66] Th. Johnsson, Target code generation from *G*-machine code, in: J. Fasel and R. Keller, eds., *Graph Reduction*, Lecture Notes in Computer Science, Vol. 279 (Springer, Berlin, 1987) 119–559.

[67] S.L. Peyton Jones, C. Clack, J. Salkild and M. Hardie, GRIP – a high-performance architecture for parallel graph reduction, in: G. Kahn, ed., *Proc. IFIP Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 274 (Springer, Berlin, 1987) 98–112.

[68] J.-P. Jouannaud and H. Kirchner, Completion of a set of rules modulo a set of equations, *SIAM J. Comput.* **15** (1986) 1155–1194.

[69] R. Keller and J. Fasel, eds., *Proc. Workshop on graph reduction, Santa Fe, New Mexico*, Lecture Notes in Computer Science, Vol. 279 (Springer, Berlin, 1987).

[70] G.M. Kelly and R. Street, Review of the elements of 2-categories, in: G.M. Kelly, ed., *Category Seminar, Sydney 1972/73*, Lecture Notes in Math. Vol. 420 (Springer, Berlin, 1974) 75–103.

[71] R. Kennaway, On "on graph rewritings", *Theoret. Comput. Sci.* **52** (1987) 37–58.

[72] R. Kennaway, Graph rewriting in some categories of partial morphisms, Technical report, School of Information Systems, University of East Anglia, 1990.

[73] R. Kieburtz, The *G*-machine: a fast, graph-reduction evaluator, in: J.-P. Jouannaud, ed., *Proc. Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 201 (Springer, Berlin, 1985) 400–413.

[74] C. Kirchner, H. Kirchner and J. Meseguer, Operational semantics of OBJ3, in T. Lepistö and A. Salomaa, eds., *Proc. 15th Internat. Colloq. on Automata, Languages and Programming, Tampere, Finland, July 11–15, 1988*, Lecture Notes in Computer Science, Vol. 317 (Springer, Berlin, 1988) 287–301.

[75] S.C. Kleene, General recursive functions of natural numbers, *Math. Ann.* **112**(5) (1936) 727–742.

[76] J.W. Klop, Combinatory reduction systems, Mathematisch Centrum, Amsterdam, 1980.

[77] Y. Lafont, The linear abstract machine, *Theoret. Comput. Sci.* **59** (1988) 157–180.

[78] J. Lambek, Subequalizers, *Canad. Math. Bull.* **13** (1979) 337–349.

[79] J. Lambek, Deductive systems and categories II, in: *Category Theory, Homology Theory and their Applications I*, Lecture Notes in Math. Vol. 86 (Springer, Berlin, 1969) 76–122.

[80] F.W. Lawvere, Adjointness in foundations, *Dialectica* **23**(3/4) (1969) 281–296.

[81] S. Leinwand, J.A. Goguen, and T. Winkler, Cell and ensemble architecture for the rewrite rule machine, in: *Proc. of the Internat. Conf. on Fifth Generation Computer Systems, Tokyo, Japan* (ICOT, Tokyo, 1988) 869–878.

[82] J.-J. Lévy, Optimal reductions in the lambda calculus, in: J.P. Seldin and J.R. Hindley, eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* (Academic Press, New York, 1980) 159–191.

[83] S. MacLane, *Categories for the Working Mathematician* (Springer, Berlin, 1971).

[84] N. Martí-Oliet and J. Meseguer, An algebraic axiomatization of linear logic models, Technical Report SRI-CSL-89-11, SRI International, Computer Science Lab, December 1989. to appear in: G.M. Reed, A.W. Roscoe and R. Wachter, eds., *Proc. Oxford Symp. on Topology in Computer Science* (Oxford University Press, Oxford, 1990).

[85] N. Martí-Oliet and J. Meseguer, From Petri nets to linear logic, in: D.H. Pitt et al., eds., *Category Theory and Computer Science*, Lecture Notes in Computer Science, Vol. 389 (Springer, Berlin, 1989) 313–340.

[86] N. Martí-Oliet and J. Meseguer, Inclusions and subtypes, Technical Report SRI-CSL-90-16, SRI International, Computer Science Lab, December 1990, submitted.

[87] J. Meseguer, Varieties of chain-complete algebras, *J. Pure Appl. Algebra* **19** (1980) 347–383.

[88] J. Meseguer, A Birkhoff-like theorem for algebraic classes of interpretations of program schemes, in: J. Díaz and I. Ramos, eds., *Formalization of Programming Concepts*, Lecture Notes in Computer Science, Vol. 107 (Springer, Berlin, 1981) 152–168.

[89] J. Meseguer, The category of commutations of Σ-term rewriting systems, Unpublished manuscript, SRI International, December 1987.

[90] J. Meseguer, On order-complete universal algebra and enriched functorial semantics., in: M. Karpiński, ed., *Proc. Foundations of Computation Theory*, Lecture Notes in Computer Science, Vol. 56 (Springer, Berlin, 1977) 294–301.

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

[92] J. Meseguer, A logical theory of concurrent objects, in: *ECOOP–OOPSLA '90 Conf. on Object-Oriented Programming, Ottawa, Canada, October 1990* (ACM, New York, 1990) 101–115.

[93] J. Meseguer, Rewriting as a unified model of concurrency, in: *Proc. Concur '90 Conf. Amsterdam, August 1990*, Lecture Notes in Computer Science, Vol. 458 (Springer, Berlin, 1990) 384–400.

[94] J. Meseguer, Rewriting as a unified model of concurrency, Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990, revised June 1990.

[95] J. Meseguer and J. Goguen, Initiality, induction and computability, in: M. Nivat and J. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, Cambridge, 1985) 459–541.

[96] J. Meseguer and U. Montanari, Petri nets are monoids: A new algebraic foundation for net theory, in: *Proc. LICS '88* (IEEE, 1988) 155–164.

[97] J. Meseguer and U. Montanari, Petri nets are monoids, *Inform. and Comput.* **88** (1990) 105–155; appeared as SRI Tech Report SRI-CSL-88-3, January 1988.

[98] J. Meseguer and T. Winkler, Parallel programming in Maude, in: J.-P. Banâtre and D. Le Mètayer, eds., *Research Directions in High-level Parallel Programming Languages*, Lecture Notes in Computer Science (Springer, Berlin, 1992), in press; see also SRI Report SRI-CSL-91-08.

[99] R. Milner, *Communication and Concurrency* (Prentice-Hall, Englewood Cliffs, NJ, 1989).

[100] R. Milner, Functions as processes, Technical Report 1154, INRIA, February 1990.

[101] R. Milner, J. Parrow, and D. Walker, A calculus of mobile processes I and II, Technical Report ECS-LFCS-89-85&86, Dept. of Computer Science, University of Edinburgh, 1989.

[102] P. Mosses, Unified algebras and institutions, Technical Report DAIMI PB-274, Computer Science Department, Aarhus University, 1989.

[103] M. Nivat, On the interpretation of recursive program schemes, *Sympos. Math.* **XV** (1975) 225–281.

[104] M.J. O'Donnell, *Computing in Systems Described by Equations*, Lecture Notes in Computer Science, Vol. 58 (Springer, Berlin, 1977).

[105] M.J. O'Donnell, *Equational Logic as a Programming Language* (MIT Press, Cambridge, MA, 1985).

[106] M.J. O'Donnell, Survey of the equational logic programming project, in: M. Nivat and H. Aït-Kaci, eds., *Resolution of Equations in Algebraic Structures* (Macmillan, New York, 1987).

[107] A. Pitts, An elementary calculus of approximations, Unpublished manuscript, University of Sussex, December 1987.

[108] G.D. Plotkin, A structural approach to operational semantics, Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.

[109] D. Plump, Implementing term rewriting by graph reduction: Termination of combined systems, in: S. Kaplan and M. Okada, eds., *Proc. Internat. Workshop on Conditional and Typed Rewriting Systems, Montreal, Canada, June 1990*, Lecture Notes in Computer Science, to appear.

[110] S. Porat and N. Francez, Fairness in term rewriting systems, Manuscript, Technion, May 3, 1990.

[111] A.J. Power, An abstract formulation of rewrite systems, in: D.H. Pitt et al., eds., *Category Theory and Computer Science*, Lecture Notes in Computer Science, Vol. 389 (Springer, Berlin, 1989) 300–312.

[112] V. Pratt, Modeling concurrency with geometry, in: *Proc. POPL '91* (ACM, New York, 1991) 311–322.

[113] J.-C. Raoult, On graph rewritings, *Theoret. Comput. Sci.* **32** (1984) 1–24.

[114] J.-C. Raoult and J. Vuillemin, Operational and semantic equivalence between recursive programs, *J. ACM* **27** (1980) 772–796.

[115] H. Reichel, *Initial Computability, Algebraic Specifications, and Partial Algebras* (Oxford University Press, Oxford, 1987).

[116] D.E. Rydeheard and J.G. Stell, Foundations of equational deduction: A categorical treatment of equational proofs and unification algorithms, in: *Proc. Summer Conf. on Category Theory and Computer Science, Edinburgh, September 1987*, Lecture Notes in Computer Science, Vol. 283 (Springer, Berlin, 1987) 114–139.

[117] R. Seely, Modelling computations: a 2-categorical framework, in: D. Gries, ed., *2nd IEEE Symp. on Logic in Computer Science* (IEEE Computer Soc. Press, Silver Spring, MD, 1987) 65–71.

[118] R.A.G. Seely, Linear logic, $*$-autonomous categories and cofree coalgebras, in J.W. Gray and A. Scedrov, eds., *Proc. AMS Summer Research Conf. on Categories in Computer Science and Logic, Boulder, Colorado, June 1987* (American Mathematical Soc., Providence, RI, 1989) 371–382.

[119] A. Sernadas, J. Fiadeiro, C. Sernadas and H.-D. Ehrich, Abstract object types: A temporal perspective, in: B. Banieqbal, H. Barringer and A. Pnueli, eds., *Temporal Logic in Specification*, Lecture Notes in Computer Science, Vol. 398 (Springer, Berlin, 1989) 324–350.

[120] E. Shapiro, The family of concurrent logic programming languages, *ACM Computing Surveys* **21** (1989) 413–510.

[121] E.W. Stark, Concurrent transition systems, *Theoret. Comput. Sci.* **64** (1989) 221–269.

[122] S. Tison, Fair termination is decidable for ground systems, in: N. Dershowitz, ed., *Rewriting Techniques and Applications, Chapel Hill, NC*, Lecture Notes in Computer Science, Vol. 355 (Springer, Berlin, 1989) 462–476.

[123] Y. Toyama, J.W. Klop and H.P. Barendregt, Termination for the direct sum of term rewriting systems, in: N. Dershowitz, ed., *Rewriting Techniques and Applications, Chapel Hill, NC*, Lecture Notes in Computer Science, Vol. 355 (Springer, Berlin, 1989) 477–491.

[124] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg and J. Sargeant, Flagship: A parallel architecture for declarative programming, in: *Proc. 15th Ann. Internat. Symp. on Computer Architecture* (1988) 124–130.

[125] A. Yonezawa and M. Tokoro, eds., *Object-Oriented Concurrent Programming* (MIT Press, Cambrigde, MA, 1988)