1 Designing Constraint Logic Programming Languages using Computational Systems

Claude Kirchner, Hélène Kirchner and Marian Vittek

1.1 Abstract

This work presents computational systems, a framework for describing computational logics, and ELAN, an interpreter of computational systems whose expressive power is examplified on several examples of constraint solving and operational semantics design. The rewriting logic, proposed by J. Meseguer, provides a logical framework in which other languages can be represented and a semantic framework for the specification of languages and systems. Here we enrich the notion of rewrite theories in rewriting logic with a notion of strategy to build computational systems. This allows expressing for example equational programming, Horn clause programming and constraint solvers as computational systems. We thus get a framework in which rewriting techniques can be used to reason about termination, confluence, combination and enrichment of computational systems.

The ELAN system implements computational systems using a specific language for expressing strategies. The system is presented and its specificities illustrated on examples.

1.2 Introduction

This work stems from the fact that logic programming in a broad sense, theorem proving and constraint solving can be described in a uniform way by giving the syntax of formulas, a set of axioms, a set of deduction rules, a proof calculus making precise how proofs are built, altogether with one or a class of models. These three computational processes can be formulated as instances of a common schema that consists of applying rewrite rules on formulas with some strategy, until getting specific forms. In logic programming the emphasis is on efficiency in reaching a successful solved form characterizing an answer, in theorem proving on completeness in detecting trivially false or true formulas, and in constraint solving on getting some predefined solved form of a constraint.

The rewriting logic, as shown in [MOM93] starting from the notion of general logic [Mes92], provides a logical framework in which other logics can be represented and a semantic framework for the specification of languages and systems. Here we also want to cover specification of constraint solving systems, in the case where the constraint solving process is described with rewrite rules that compute solved forms of constraints. This includes for now constraint languages built from elementary constraints that may be equations [Kir85, JK91], disequations [Com91], inequations on terms compared with simplification orderings [NR92], membership constraints [Com90]. Computational systems allow expressing nicely the proof calculus of a given logic, in order to describe a computational version, if any, of the logic under consideration.

A computational system is given by a signature providing the syntax, a set of conditional rewrite rules describing the deduction mechanism, and a strategy to guide application of rewrite rules. Formally, this is the combination of a rewrite theory in rewriting logic, together with a notion of strategy to efficiently compute with given rewrite rules. The underlying framework is first-order, since the formulas, sets of formulas and proofs are considered as first-order terms. Computation is exactly application of rewrite rules on a term and a strategy describes the intended set of computations, or equivalently in rewriting logic, a subset of proof terms. Computational systems may be generic in the sense that they can be parameterized by a class of signatures and axioms. In this case to each program, i.e. each pair of a signature and a set of axioms, corresponds by instantiation, a computational system.

To validate these ideas, we develop a prototype called ELAN, to describe, experiment and study the combination of different computational systems that provide basis for constraint solving and logic programming paradigms. Computational systems have several advantages: first of all, since the framework is based on first-order tools and the rewriting logic, it is conceptually simple to understand. Second, we benefit from the work on (equational) rewriting and efficient implementation techniques both on sequential models [HL91] and distributed models [KV92, Gog87]. Third, we can define rewriting in the ELAN language itself, and we will see in Section 1.5.5 how easily this can be achieved. Last and even more important, we can reason about computational systems, for instance prove abstract properties of the rewrite rules like termination or confluence with rewrite-based techniques. We can also reason about combination and enrichment of computational systems and build new ones in a modular way, once we know how systems theoretically interact.

So our main contributions are the notion of computational system, the description of the ELAN interpreter of computational systems and several simple but non-trivial and meaningful examples of applications of the introduced concepts. The paper is an elaborated version of ideas first presented in [KKV93]. In the next section, we give the definition of computational systems based on rewriting logic, which is first briefly recalled. We explain what we mean by implementing a specific logic by a computational system. We illustrate the concepts by a simple equational programming language whose operational semantics is narrowing, and by a constraint solver process for unification problems modulo commutativity. The ELAN system is then presented with its specificities, illustrated on the example of a simple Horn clause programming language. Reflexivity of the system is addressed before the conclusion in which we give further perspectives for the development of ELAN.

1.3 General setting

This section first presents our view of the main concepts of general logic [Mes89] and rewriting logic [Mes92] both due to J. Meseguer. The computational systems are then defined as rewrite theories in rewriting logic together with strategies to restrict the computation space. This provides the semantical foundation of the ELAN system, as an environment in which computational systems can be described and then executed.

1.3.1 Rewriting logic

The definitions below are given in the one-sorted case. The many-sorted and order-sorted cases can be handled in a similar, although more technical, manner. Our definitions are consistent with [DJ90, JK91] to which the reader is referred for more detailed considerations on universal algebra, term rewriting systems and unification.

We consider a set $\mathcal{F}$ of ranked function symbols where $\mathcal{F}_n$ is the subset of functions of arity n, a set $\mathcal{X}$ of variables and the set of first-order terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ built on $\mathcal{F}$ using variables in $\mathcal{X}$. The set of variables of a term t is denoted $\mathcal{V}ar(t)$ and this notation is extended in a natural way to any object based on terms. The set of ground terms (i.e. terms without variables) is denoted $\mathcal{T}(\mathcal{F})$. A $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equality is a multiset of terms $\{t, t'\}$ in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, denoted as usual $(t = t')$. For a set of equalities E, $\mathcal{T}(\mathcal{F}, \mathcal{X})/E$ denotes the free algebra of terms modulo E. The equivalence class of a term t modulo E is denoted $\langle t \rangle_E$. To support intuition, the reader can think of E as being the empty set of axioms, or the associativity commutativity axioms for some binary symbol. For details and general results on calculus modulo equational axioms, the reader is invited to consult for example [JK86]. To simplify notation, we denote a sequence of objects $(a_1, \ldots, a_n)$ by $\overline{a}$.

In rewriting logic, proofs are first-order objects and are represented by proof terms. In order to build proof terms and strategies, rules are labeled with a set $\mathcal{L}$ of ranked label symbols, i.e. of symbols with a fixed arity. In order to compose proofs, we introduce the infix binary operator ";". A *proof term* is by definition a term built on equivalence classes of $\mathcal{T}(\mathcal{F}, \mathcal{X})/E$, function symbols in $\mathcal{F}$, label symbols in $\mathcal{L}$ and the concatenation operator ";". In other words, a proof term is an element of the term algebra $\mathcal{PT} = \mathcal{T}(\mathcal{L} \cup \{;\} \cup \mathcal{F} \cup \mathcal{T}(\mathcal{F}, \mathcal{X})/E)$.

A logic is defined by a syntax, an entailment system, a class of models and a satisfaction relation. These four components respectively denoted by $(Synt, \vdash, Model, \models)$ are more precisely defined below, first for a general logic and second in the case of rewriting logic.

Syntax. The syntax needed for defining a logic is provided by a *signature* which allows building *sentences*. Typically a signature for first-order logic consists of two sets of ranked function symbols and predicate symbols. Since we need to be generic, we consider $\mathcal{S}ynt$ a class of pairs (Σ, sen) consisting of a signature Σ together with a mapping sen associating to Σ the set of all legal sentences built on this signature.

▶ In rewriting logic, a signature consists of a 3-tuple $\Sigma = (\mathcal{L}, \mathcal{F}, E)$, where $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols and E is a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities. Equalities in E must be understood as structural axioms forming part of the signature. Typically in a signature allowing a conjunction operator $\wedge$, E consists in the associativity and commutativity axioms for $\wedge$. Sentences built on a given signature are defined as sequents of the form $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$ where $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $\pi \in \mathcal{PT}$. The informal meaning of such sentences is that the proof π allows to derive t' from t.

Entailment systems. For a given class of syntax $\mathcal{S}ynt$ and (Σ, sen) in $\mathcal{S}ynt$, a *theory* is a pair $T = (\Sigma, \Phi)$ where $\Phi \subseteq sen(\Sigma)$.We say that T is *presented* by the set of axioms Φ.

Given a signature Σ, an entailment system is an abstract description of the provability relation of a sentence ϕ starting from a given set of sentences (also called axioms) Φ and using logical rules. Formally, given a class of syntax $\mathcal{S}ynt$, an *entailment system* is a pair $(\mathcal{S}ynt, \vdash)$ such that $\vdash$ is a function associating to each $(\Sigma, sen) \in \mathcal{S}ynt$ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$ such that:

1. $\forall \phi \in sen(\Sigma), \{\phi\} \vdash_\Sigma \phi$, (Reflexivity)
2. If $\Phi \vdash_\Sigma \phi$ and $\Phi \subseteq \Phi'$ then $\Phi' \vdash_\Sigma \phi$, (Monotonicity)
3. If $(\forall i \in I, \Phi \vdash_\Sigma \phi_i)$ and $(\Phi \bigcup_{i \in I}\{\phi_i\} \vdash_\Sigma \phi)$ then $\Phi \vdash_\Sigma \phi$, (Transitivity)
4. For any signature morphism H:
   if $\Phi \vdash_\Sigma \phi$, then $H(\Phi) \vdash_{H(\Sigma)} H(\phi)$. (Translation)

► In rewriting logic, in order to build the entailment system, the notion of rewrite theory is introduced and an appropriate deduction system allows to inductively define (in fact itself using a rewrite theory) the entailment relation.

A *labeled rewrite theory* is a 5-tuple $\mathcal{R} = (\mathcal{X}, \mathcal{F}, E, \mathcal{L}, R)$ where $\mathcal{X}$ is a given countably infinite set of variables, $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols, E a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities, and R a set of labeled rewrite rules of the form $\ell : g \rightarrow d$ where $\ell \in \mathcal{L}$ and $g, d \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ satisfying $\mathcal{V}ar(d) \subseteq \mathcal{V}ar(g)$ and where the arity of ℓ is exactly the number of distinct variables in g. The labels are of crucial use both in the definition of strategies and in the definition of models for the rewriting logic, in which they allow tracing deductions. Note also that the definitions can be extended to conditional rewriting at the cost of more technically complicated definitions. The reader is referred to [Mes92] for a full treatment of this case.

A given labeled rewrite theory $\mathcal{R}$ entails the sequent $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$, which is denoted $\mathcal{R} \vdash \pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$, if $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$ is obtained by some finite application of the deduction rules in Figure 1.1. Note that because of the rules **Reflexivity** and **Replacement**, if the rule $r = (\ell : g \rightarrow d) \in R$, then the sequent $seq(r) = (\ell(\langle x_1 \rangle_E, \ldots, \langle x_n \rangle_E) : \langle g \rangle_E \rightarrow \langle d \rangle_E)$ can be derived. The set of all sequents $\{seq(r) | r \in R\}$ is denoted $seq(R)$ and is used to present the theory.

Reflexivity

$$\Vdash\!\twoheadrightarrow$$
$$\langle t\rangle_E : \langle t\rangle_E \to \langle t\rangle_E$$
if $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$

Congruence

$$\pi_1 : \langle t_1\rangle_E \to \langle t'_1\rangle_E, \ldots, \pi_n : \langle t_n\rangle_E \to \langle t'_n\rangle_E$$
$$\Vdash\!\twoheadrightarrow$$
$$f(\pi_1, \ldots, \pi_n) : \langle f(t_1, \ldots, t_n)\rangle_E \to \langle f(t'_1, \ldots, t'_n)\rangle_E$$
if $f \in \mathcal{F}_n$

Replacement

$$\pi_1 : \langle t_1\rangle_E \to \langle t'_1\rangle_E \ldots \pi_n : \langle t_n\rangle_E \to \langle t'_n\rangle_E$$
$$\Vdash\!\twoheadrightarrow$$
$$\ell(\pi_1, \ldots, \pi_n) : \langle g(t_1, \ldots, t_n)\rangle_E \to \langle d(t'_1, \ldots, t'_n)\rangle_E$$
if $\ell : g(x_1, \ldots, x_n) \to d(x_1, \ldots, x_n) \in R$

Transitivity

$$\pi_1 : \langle t_1\rangle_E \to \langle t_2\rangle_E,\ \pi_2 : \langle t_2\rangle_E \to \langle t_3\rangle_E$$
$$\Vdash\!\twoheadrightarrow$$
$$\pi_1 ; \pi_2 : \langle t_1\rangle_E \to \langle t_3\rangle_E$$

Figure 1.1
REW: The rules of rewrite deduction

The entailment system is then defined for a signature $\Sigma = (\mathcal{L}, \mathcal{F}, \mathcal{X}, E)$ and a labeled rewrite theory $\mathcal{R}$, by $seq(R) \vdash_\Sigma (\pi : u \to v)$ if $\mathcal{R} \vdash (\pi : u \to v)$. The set of variables in the definition of a rewrite signature is often implicit. It is not difficult to verify that the desired properties of an entailment system are fulfilled.

Models. For a given syntax $\mathcal{S}ynt$, to each signature and associated set of sentences $(\Sigma, sen) \in \mathcal{S}ynt$ is associated a class $Mod(\Sigma)$ of objects called Σ-models and a satisfaction relation denoted $\models_\Sigma \subseteq Mod(\Sigma) \times sen(\Sigma)$ which is compatible with signature morphisms (see [Mes89] for more details).

For a given set of sentences Φ, $Mod(\Sigma, \Phi)$ is the class of all Σ-models M such that $\forall \phi \in \Phi, M \models_\Sigma \phi$, called the models satisfying Φ. $Mod(\Sigma, \Phi) \models_\Sigma \phi$ means that $M \models_\Sigma \phi$ for all $M \in Mod(\Sigma, \Phi)$.

► A model of rewriting logic can be chosen as the computation space of the considered rewrite theory $\mathcal{R}$. Such a computation space is determined by the set of proof terms π computed in the sequents $\pi : \langle t\rangle_E \to \langle t'\rangle_E$ modulo the equivalence of computations. This equivalence is given by E and a set $\mathcal{E}_{\mathcal{PT}(R)}$ of equational axioms on the set of proof terms described in Figure 1.2. So the considered model is the

$\forall \pi_1, \pi_2, \pi_3 \in \mathcal{PT}$	$\pi_1; (\pi_2; \pi_3) = (\pi_1; \pi_2); \pi_3$	Associativity
$\forall \pi : \langle t \rangle_E \to \langle t' \rangle_E,$	$\pi; \langle t' \rangle_E = \pi,$ and $\langle t \rangle_E; \pi = \pi$	Local Identities
For all $f \in \mathcal{F}_n, n \in \mathbf{N}$, $\forall \pi_1, \dots, \pi_n, \pi'_1, \dots, \pi'_n$: $f(\pi_1; \pi'_1, \dots, \pi_n; \pi'_n) = f(\pi_1, \dots, \pi_n); f(\pi'_1, \dots, \pi'_n)$		Independence
For all $f \in \mathcal{F}_n, n \in \mathbf{N}$: $f(\langle t_1 \rangle_E, \dots, \langle t_n \rangle_E) = \langle f(t_1, \dots, t_n) \rangle_E$		Preservation of E
$\forall \ell : g \to d \in R, \forall \pi_1 : \langle t_1 \rangle_E \to \langle t'_1 \rangle_E, \dots, \pi_n : \langle t_n \rangle_E \to \langle t'_n \rangle_E$ $\ell(\pi_1, \dots, \pi_n) = \ell(\langle t_1 \rangle_E, \dots, \langle t_n \rangle_E); d(\pi_1, \dots, \pi_n)$ and $\ell(\pi_1, \dots, \pi_n) = g(\pi_1, \dots, \pi_n); \ell(\langle t'_1 \rangle_E, \dots, \langle t'_n \rangle_E)$		Parallel Move Lemma

Figure 1.2
$\mathcal{E}_{\mathcal{PT}(R)}$: Equivalence of proof terms

quotient set $\{\pi \mid \mathcal{R} \vdash \pi : \langle t \rangle_E \to \langle t' \rangle_E\}/(E \cup \mathcal{E}_{\mathcal{PT}(R)})$.

Logic. We have now defined the ingredients of a logic $\mathcal{L}$, which is given by five components $\mathcal{L} = (\mathcal{S}ynt, sen, \mathcal{M}od, \vdash, \models)$ such that $(\mathcal{S}ynt, sen, \vdash)$ is an entailment system and $(\mathcal{S}ynt, sen, \mathcal{M}od, \models)$ a class of models that satisfy furthermore the *soundness* condition:

$$\forall (\Sigma, sen) \in \mathcal{S}ynt, \Phi \subseteq sen(\Sigma), \phi \in sen(\Sigma), \;\; \Phi \vdash_\Sigma \phi \Rightarrow Mod(\Sigma, \Phi) \models_\Sigma \phi.$$

The logic is *complete* when the above implication is also an equivalence.

▶ This specializes trivially to the notion of rewriting logic, using the basic ingredients defined before.

Proof calculus. When a logic is defined this way, it is quite abstract since there is a priori no constructive hint about how the entailment relation is built. Thus this approach only gives a formal description of the relations $\vdash$ and $\models$ and of their relationship. What we need also is to describe how to prove sentences starting from a set of axioms Φ. This description of proofs should allow the flexibility to define subcalculi that are complete for proving theorems in a specific subclass of the set of sentences. For example linear resolution is a (correct and) complete proof method for Horn clause logic and this induces the well-known refutation as computation paradigm. Thus we require that a proof calculus operationally describes the relation $\vdash$; indeed there may exist several proof calculi for a logic, as it is the case for first-order logic.

A proof calculus associates to each theory T an algebraic structure $P(T)$ using axioms of T as hypotheses and using deduction rules as operators to build new proofs. The set $proofs(T)$ consists of all proofs of all theorems of the theory T. To each proof is associated the theorem it is a proof of: τ_T maps each proof $p \in proofs(T)$ to the sentence ϕ such that p is a proof of $\phi = \tau_T(p)$. A *proof subcalculus* formalizes the restriction of an entailment system to a specific syntax and to specific forms of sentences as axioms or conclusions. This allows us to consider only a subset of all proofs and thus to account for efficiency. A proof calculus or subcalculus is formally given by an entailment system and for each theory T, the associated triple $(P(T), proofs(T), \tau_T)$.

▶ In rewriting logic, the set of proofs is defined as the set of proof terms and the proof calculus is thus built by the same set of deduction rules defining the entailment system in Figure 1.1.

1.3.2 Computational systems

In [MOM93], it is shown how various logics can be represented in rewriting logic, as for example equational logic, Horn logic, linear logic and the natural semantics [Kah87]. We will rather focus now on the specification of operational semantics of languages and systems like constraint solving. So, we need to express not only a rewrite theory but also computations in this theory that mirror computations in various logical systems, e.g. how a deduction in Horn logic is mapped to a proof in rewriting logic.

Computations. The first ingredient of a computational system is a labeled rewrite theory from which we define a notion of computation.

Let us first concentrate on a one-step computation. For a given rewrite theory $\mathcal{R}$, and for the sequent $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$ with the proof term $\pi = t[\ell(\sigma\overline{x})]_\omega$, then

$$\langle t \rangle_E \Rightarrow_{\ell,\sigma,\omega} \langle t' \rangle_E$$

is called a *single rewrite step*. Note that this corresponds exactly to the usual definition of one step of rewriting at occurrence ω using the rule labeled ℓ and the match σ.

Indeed, we are interested in a canonical representation of all computations that are equivalent modulo the axioms $\mathcal{E}_{\mathcal{PT}(R)}$ (see Figure 1.2). This can be done thanks to the following result [Mes92] which shows that up to associativity of the ";" symbol, any sequent can be decomposed into a composition of elementary (sequential) ones: for any sequent $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$, either $\pi = \langle t \rangle_E = \langle t' \rangle_E$, or there exists $n \in \mathbf{N}$ such that:

$$\langle t \rangle_E = \langle t_0 \rangle_E \Rightarrow_{\ell_0} \langle t_1 \rangle_E \Rightarrow_{\ell_1} \langle t_2 \rangle_E \cdots \Rightarrow_{\ell_{n-1}} \langle t_n \rangle_E = \langle t' \rangle_E,$$

and $\pi =_{A(;)} (\pi_0; \pi_1; \pi_2; \ldots; \pi_{n-1})$, where $A(;)$ denotes the associativity of ";". When π is the proof term above, $\langle t_n \rangle_E$ is called the *result of* π *applied to* $\langle t_0 \rangle_E$ and this is also denoted $\langle t \rangle_E \stackrel{\pi}{\Longrightarrow} \langle t' \rangle_E$.

Indeed the equivalence relation generated by $(E \cup \mathcal{E}_{\mathcal{PT}(R)})$ on proof terms induces an equivalence on computations: two computations are equivalent (they lead to the same result) if their proof terms are equivalent.

Strategies. In general, we are not interested in all computations, but only in those guided by a strategy, which is in this work a description of the single rewrite steps sequence allowed in the computations. From a formal point of view, a *strategy* is a subset of the set of proof terms which is closed under concatenation.

The result of applying a strategy $\mathcal{S}$ on a term t, denoted functionally $\mathcal{S}(t)$, is the set of all $\langle t' \rangle_E$ for which there exists $\pi \in \mathcal{S}$ such that $\langle t' \rangle_E$ is the result of π applied to $\langle t \rangle_E$:

$$\mathcal{S}(t) = \{\langle t' \rangle_E \mid \exists \pi \in \mathcal{S}, \langle t \rangle_E \stackrel{\pi}{\Longrightarrow} \langle t' \rangle_E\}.$$

It should be noted from this definition that a strategy applied to a term may have several different terms as results.

A first way to describe a strategy is to extensively enumerate the intended subset of the proof terms. But this is of course not satisfactory in practice and the problem indeed is to define a language to describe subsets of the proof terms. We will see later how is implemented in ELAN an instance of this language which is powerful enough to deal with the examples we are interested in.

Computational systems. We are now in position to define the main concept of this paper: A *computational system* is a labeled rewrite theory $\mathcal{R} = (\mathcal{X}, \mathcal{F}, E, \mathcal{L}, R)$ together with a strategy $\mathcal{S}$.

In order to show the expressivity of this concept, we give below examples of languages and systems design in this formalism. Before entering into these examples, we first present the way strategies can be expressed.

1.3.3 Expressing strategies

In this section, we present one possibility, adopted in the current version of the language ELAN, to express a subset of proof terms using a first-order language. We intend to make the strategy description conceptually easy and such that the structure of the denoted proof terms can be straightforwardly guessed from it.

In the strategy language, we first want to include the rule names, an iterator operator that describes the iteration of a computation zero and more times, a concatenation operator and a selecting operator that, given n arguments, provides a possible choice of one of them. To raise the expressive power of expressions built on this vocabulary, we need the possibility to express application of a strategy on subterms. Moreover the strategies can be named from a set of strategy names $\mathcal{N}$, and the names of strategies can be used in the definition of other strategies. With these requirements, a simple language for defining strategies can be designed. A strategy expression in this language is a term built up on a set of rule labels $\mathcal{L}$, a set of strategy names $\mathcal{N}$, a set of function symbols $\mathcal{F}$, the binary and associative concatenation symbol ";", a unary function symbol **iterate** and a function symbol **choose** with a variable arity.

- The strategy expression consisting just in one rule name corresponds to all computations obtained by exactly one application of this rule.
- The concatenation $s_1; s_2$ of two strategy expressions s_1 and s_2 in $\mathcal{S}$ denotes all computations made with respect to the strategy s_1 then to the strategy s_2.
- The expression **choose**$(s_1, \ldots, s_n)$ where $s_1, \ldots, s_n$ are in $\mathcal{S}$, describes a possible choice of one of the strategies given as argument.
- The expression **iterate**(s) where s is in $\mathcal{S}$, describes the iteration of a computation zero, one, two, ... times.

- The expression $f(s_1, \ldots, s_n)$ where $s_1, \ldots, s_n$ are in $\mathcal{S}$ and f is a functional symbol from $\mathcal{F}$ with arity n, expresses the fact that the term to be reduced must be of the form $f(t_1, \ldots, t_n)$ and that each subterm t_i will be reduced following the strategy s_i.
- A strategy name in $\mathcal{N}$ is just an abbreviation for a complex expression s. A name n has to be defined by an equality $n = s$ where $s \in \mathcal{S}$. It is enough for our purpose to have non-recursive equalities here, but the case of recursive definitions of strategies is described in [Vit94].

On the set of all strategy expressions $\mathcal{S} = \mathcal{T}(\mathcal{L} \cup \mathcal{N} \cup \mathcal{F} \cup \{;, \mathbf{iterate}, \mathbf{choose}\})$, we define a function $\mathcal{PF} : \mathcal{S} \to \mathcal{PT}$ giving the set of proof terms corresponding to each expression in $\mathcal{S}$. $\mathcal{PF}$ is recursively defined as follows:

- If ℓ is the name of a rule, with variables $x_1, \ldots, x_n$;
$\mathcal{PF}(\ell) = \{\ell(\langle t_1\rangle_E, \ldots, \langle t_n\rangle_E) \mid \forall i = 1, \ldots, n, t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})\}$.
- If s_1 and s_2 are in $\mathcal{S}$;
$\mathcal{PF}(s_1; s_2) = \{\pi_1; \pi_2 \mid \pi_1 \in \mathcal{PF}(s_1), \pi_2 \in \mathcal{PF}(s_2)\}$
- If $s_1, \ldots, s_n$ are in $\mathcal{S}$;
$\mathcal{PF}(\mathbf{choose}(s_1, \ldots, s_n)) = \{\pi_i \mid \pi_i \in \mathcal{PF}(s_i), 1 \leq i \leq n\} = \bigcup_{1 \leq i \leq n} \mathcal{PF}(s_i)$.
- If s is in $\mathcal{S}$;
$\mathcal{PF}(\mathbf{iterate}(s)) = \{\langle t\rangle_E \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{X})\} \cup \{\pi_1; \ldots; \pi_n \mid \pi_i \in \mathcal{PF}(s), 1 \leq i \leq n, n \in \mathbf{N}\}$, where the first part of this union stands for the zero iteration case.
- If $s_1, \ldots, s_n$ are in $\mathcal{S}$ and $f \in \mathcal{F}_n$;
$\mathcal{PF}(f(s_1, \ldots, s_n)) = \{f(\pi_1, \ldots, \pi_n) \mid \pi_i \in \mathcal{PF}(s_i), 1 \leq i \leq n\}$.
- If n is a name from $\mathcal{N}$ defined by $n = s$;
$\mathcal{PF}(n) = \mathcal{PF}(s)$.

1.3.4 Mapping to rewriting logic

Our goal is now to use computational systems to express and compute with various logics describing the operational semantics of languages and systems. Formally programs in such languages are theories (Σ, Φ) and each entailment $\Phi \vdash \phi$ in the given logic has a proof built by the proof calculus. We need to map this proof to a computation in a computational system that describes the construction of proofs with labeled rewrite rules and a strategy. To do that, we need to map a theory (Σ, Φ) to a rewrite theory (Σ', Φ') in such a way that any proof from (Σ, Φ) is mirrored by a proof from (Σ', Φ').

Given an entailment system $(\mathcal{S}ynt, \vdash)$ for a logic $\mathcal{L}$, a *mapping to rewriting logic* is defined by a theory mapping

$$\mathcal{M} : T = (\Sigma, \Phi) \mapsto \mathcal{M}(T) = (\mathcal{M}(\Sigma), \mathcal{M}(\Phi))$$

and a proof mapping $\alpha : proofs(T) \mapsto proofs(\mathcal{M}(T))$, such that for any proof p of a theorem ϕ in T, $\alpha(p)$ is a proof of $\mathcal{M}(\phi)$ in $\mathcal{M}(T)$. Mappings of interest are those in which moreover proofs in rewriting logic actually correspond to proofs in the original logic.

Since our goal logic is rewriting logic, $(\mathcal{M}(\Sigma), \mathcal{M}(\Phi))$ is a labeled rewrite theory, $\mathcal{M}(\phi)$ is a sequent of the form $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$. So for a given logic, we have to build a computational system that realises this mapping. Since we want to abstract from the theories, we actually need to define a *generic computational system* that realises this mapping for any theory in the original logic. Indeed this is in general a non-trivial problem to prove that a computational system realises the required mapping and we give examples of such correct constructions in the two next sections.

1.4 Examples of computational systems

1.4.1 Mapping equational programs to computational system

The first example we consider is equational programming based on narrowing. Such languages have been considered for instance (among many others) in [GM86, DP88, MNRA92].

Using the framework of the previous section, equational programs are theories built on a first order signature where the axioms are finite sets of equalities denoted $Pg = \{l_i \sim r_i | i \in I\}$, which once oriented from left to right form a convergent term rewrite system $PgR = \{l_i \rightarrowtail r_i | i \in I\}$. In order to avoid confusing the reader, we distinguish between the rewrite system underlying rewriting logic and the rewrite system PgR, systems that are a priori different: we call PgR a set of oriented equalities and we denote the arrow symbol with the special symbol $\rightarrowtail$. This difference between the rewrite relation of the computational system and the one defined by the user in a specification of his choice is further discussed in the later section on reflexivity.

The query to solve is assumed to be of the form $g(\overline{x}) = d(\overline{x})$ and the underlying proof calculus is the basic narrowing process [Hul80] that computes a complete set of Pg-unifiers of the query. In order to better understand how basic narrowing is described in rewriting logic, let us briefly explain this process. The reader interested by deeper details and references on narrowing can refer to [JK91]. The initial query is transformed into an expression of the form $(\exists\emptyset : g = d \mid\mid \top)$, where $\top$ denotes the empty conjunction of equations, and $\exists\emptyset$ is a trivial quantification over an empty set of variables. The basic narrowing process transforms expressions of the form $(\exists X, g = d \mid\mid S)$ where $g = d$ is an equation to be solved modulo the equational theory generated by Pg and S a conjunction of equations to be solved in the empty theory. The process tries to unify a subterm of g at position ω with a left-hand side of an oriented equality $l \rightarrowtail r \in PgR$, taking care, by the use of a $Variant$ function, that its variables are disjoint from those of g. This gives rise to a unification problem $(g_{|\omega} =^?_\emptyset l)$ and a new goal $(g[r]_\omega = d)$ to be further solved. So a current query $(\exists X, g = d \mid\mid S)$, where the existentially quantified variables are the "new" variables introduced by the narrowing process, is transformed into the following one:

$$(\exists X \cup \mathcal{V}ar(l \rightarrowtail r),\ g[r]_\omega = d \mid\mid S \wedge (g_{|\omega} =^?_\emptyset l))$$

such that the two systems of equations $(\exists X, g =^?_{Pg} d \wedge S)$ and $(\exists X \cup \mathcal{V}ar(l \rightarrow r),\ g[r]_\omega =^?_{Pg} d \wedge S \wedge (g_{|\omega} =^?_\emptyset l))$ have the same set of solutions. Furthermore a query $(\exists X, g = d \mid\mid S)$ has an obvious solution when g and d are syntactically unifiable and this is exactly the solution of the system $S \wedge (g =^?_\emptyset d)$. Note that a successful computation transforms a problem in the theory Pg (or equivalently PgR) into unification problems in the empty theory. Indeed there is a lot of undeterminism in the choices of the oriented equality $(l \rightarrowtail r)$ and the position ω where the narrowing step is applied. All possible choices have to be explored when completeness of the returned set of solutions is required (see [NRS90]), which is indeed what we want here.

To define a mapping from equational programming to rewriting logic, we consider a rewrite theory whose signature is taken from the program and enriched by the symbols $\rightarrowtail$ for building program rules, $\exists$, $=$ and $\mid\mid$ for building queries, $=^?_\emptyset$ and $\wedge$ for building equations and equation systems. The rewrite rules of the rewrite theory, that apply to formulas built on this enriched signature, are as follows:

Narrow $(\exists X, g = d \,||\, S)$
$\to$
$(\exists X \cup \mathcal{V}ar(l, r),\ g[r]_\omega = d \,||\, S \wedge (g_{|\omega} =^?_\emptyset l))$
if $l \rightarrowtail r \in Variants(PgR)$ and $(S \wedge (g_{|\omega} =^?_\emptyset l))$ is satisfiable

Block $(\exists X, g = d \,||\, S)$
$\to$
$(\exists X, \top \,||\, S \wedge (g =^?_\emptyset d))$
if $(S \wedge (g =^?_\emptyset d))$ is satisfiable

Since the queries are of the form $g(\overline{x}) = d(\overline{x})$, which is the same as $(\exists\emptyset, g = d \,||\, \top)$, the formula to prove in rewriting logic is:

$((\exists\emptyset, g = d \,||\, \top) \to (\exists X, \top \,||\, S))$,

where S is a system of equations solvable in the empty theory. This last condition is ensured using a unification process itself described by a computational system, as this is done in the next section for commutative unification.

The results on correctness and completeness of basic narrowing state that each computation proving such a formula consists of several applications of rule **Narrow** followed by exactly one application of rule **Block**. So, to complete the definition of the computational system, it remains to define a strategy $\mathcal{S}$ that expresses this completeness result. With the notations defined before, we can take for strategy $\mathcal{S} = \mathbf{iterate}(\mathbf{Narrow}); \mathbf{Block}$, which describes the following application of rewrite rules:

$\{\mathbf{Block},\ \ \mathbf{Narrow};\mathbf{Block},\ \ \mathbf{Narrow};\mathbf{Narrow};\mathbf{Block},\ldots\}.$

The result of the application of this strategy is the set of all expressions $(\exists X, \top \,||\, S)$ such that $(\exists X, \top \,||\, S) \in \mathcal{S}(\exists\emptyset, g = d \,||\, \top)$.

1.4.2 Constraint solving as computational systems

The description of constraint solving using rule-based algorithms as in [Com91, JK91], allows easier correctness and completeness proofs of constraint solvers, partly thanks to the explicit distinction made in this approach between deduction rules and control. At the light of the framework described in this paper, this rule approach to constraint solving can now be seen as the expression of the constraint solving process as a computational system. A constraint solver for symbolic constraints such as equations, inequations or disequations on terms is viewed as a computational system aimed at computing solved forms for a class of considered formulas called constraints.

For instance, unification in commutative theories is a constraint solving process that can be described with a computational system where the signature of underlying rewriting logic consists of first-order terms with an equality symbol denoted $=^?_C$ (C stands for commutative). As rewrite rules, we use rules proposed in [JK91], which are proved correct and complete for the computation of a complete set of C-unifiers. These rules transform conjunctions of equations of the form $(P \wedge s =^?_C t)$ and are given in Figure 1.3. They use an implicit universal quantification on the variables of conjunction of equations (P), on terms (t_i, s_i) and on the operator symbols (f, g). These later symbols are in fact instantiated when the program, i.e. the rewrite theory, is known. The $+$ symbol is assumed to be the only one which is commutative; when several commutative symbols are needed, general results on combining unification algorithms have to be invoked [BS92, Rin93]. The symbol $\mathbf{F}$ denotes a problem without solution (i.e. failure), and for any term t, $|t|$ denotes its size, i.e. the number of its function symbols.

Following [JK91], solving a given equation $s =^?_C t$ consists in finding the set of its (tree) solved forms. A solved form is either $\mathbf{F}$ if the equation has no solution or a system of equations with the same set of solutions as $s =^?_C t$, and of the form $\bigwedge_{i \in I} x_i =^?_C t_i$ where the variables x_i do not appear elsewhere in the system. Thus, in rewriting logic, we are interested by proofs of the formula: $(s =^?_C t) \rightarrow S$, where S is a solved form. This is expressed using an appropriate strategy: let $\mathcal{S}$ be a strategy that causes application of rules in any order until the normal form is obtained. Then solving the constraint $(s =^?_C t)$ amounts to look for each S such that: $S \in \mathcal{S}(s =^?_C t)$.

Describing constraint solving processes as computational systems has several advantages over constraint solving systems where solvers are encapsulated in black boxes.

1. Although completeness may be in some cases, like syntactic unification, proved for any normalizing strategy, the solved forms are in general reached more efficiently with smart choices of rules. Through the design

Delete	$P \wedge s =^?_C s$ $\rightarrow$ P
Decompose	$P \wedge f(s_1, \ldots, s_n) =^?_C f(t_1, \ldots, t_n)$ $\rightarrow$ $P \wedge s_1 =^?_C t_1 \wedge \ldots \wedge s_n =^?_C t_n$
ComMutate	$P \wedge s_1 + s_2 =^?_C t_1 + t_2$ $\rightarrow$ $P \wedge s_1 =^?_C t_2 \wedge s_2 =^?_C t_1$
Conflict	$P \wedge f(s_1, \ldots, s_n) =^?_C g(t_1, \ldots, t_p)$ $\rightarrow$ **F** if $f \neq g$
Coalesce	$P \wedge x =^?_C y$ $\rightarrow$ $\{x \mapsto y\}P \wedge x =^?_C y$ if $x, y \in \mathcal{V}ar(P)$ *and* $x \neq y$
Merge	$P \wedge x =^?_C s \wedge x =^?_C t$ $\rightarrow$ $P \wedge x =^?_C s \wedge s =^?_C t$ if $0 < \lvert s\rvert \leq \lvert t\rvert$
Check	$P \wedge x =^?_C s$ $\rightarrow$ **F** if $x \in \mathcal{V}ar(s)$ *and* $s \notin \mathcal{X}$
Eliminate	$P \wedge x =^?_C s$ $\rightarrow$ $\{x \mapsto s\}P \wedge x =^?_C s$ if $x \notin \mathcal{V}ar(s), s \notin \mathcal{X}, x \in \mathcal{V}ar(P)$

Figure 1.3
ComUnif: The rules for commutative unification

of rules and strategies, the developer has a direct access to the constraint solver and this point is crucial for discovering and experimenting new solvers.

2. To be able to formalize constraint solving by rewrite rules also makes termination proofs easier and possibly partly automated. This also opens the door to non-trivial applications of termination proof techniques.

3. Constraint handling like simplification, entailment or propagation can be described in a very abstract way.

4. Another interest of this point of view is the conceptually easy combination of constraint solving with other computational systems. Indeed a computational system for a language involving now formulas with constraints $(\varphi \parallel c)$, where φ is a formula (a Horn clause for instance) and c is a constraint, is incrementally built by importing a computational system for the constraint language in which c is expressed. Statements are sets of formulas with constraints, like in $CLP(X)$ [JL87]. It is yet possible to define rewrite rules on formulas with constraints, as mentioned later in Section 1.6, but such combinations generates delicate problems concerning completeness.

1.5 The ELAN interpreter

Sketched in a few words, ELAN can be understood as a specification language that allows first to describe the syntactic part of a logic and the transformation of its proofs into proofs in the rewriting logic, and second to perform actual proofs using the computational system. Thus ELAN is a language for designing and executing computational systems. As such, it provides an implementation of the concepts presented in Section 1.3, where some slight adaptations and restrictions with respect to the general case were made and are detailed in this section. A full description of the language and its implementation is given in [Vit94].

ELAN is a modular language, so a computational system description is usually divided into modules. Each module represents a particular computational system and is combined with other ones to build the whole system. Modules can be parametrized. ELAN works with many-sorted rewriting logic, and the definition of a signature permits mixfix notation of symbols. These facilities are largely used in examples presented in the rest of this paper.

1.5.1 Expressing mappings in ELAN

To smartly implement the theory mapping from a given theory to a computational system in the sense of Section 1.3.4, ELAN provides a special kind of parametrization of computational systems by abstracting from the actual signature and axioms of specifications.

A non-standard construction of parametrization in ELAN is a **FOR EACH** construction, which allows the generation of rewrite rules with respect to an actual theory, as illustrated on examples in the following sections. This parametrization tool in ELAN is very powerful and also permits internalization of objects as shown in Section 1.5.5.

1.5.2 Expressing rewrite rules in ELAN

In full generality, the underlying term algebra of a rewrite theory is defined as the quotient term algebra modulo some given set of equations E. All rewrite rules are then working on these equivalence classes rather than on simple terms. This concept is implemented in ELAN, in the restricted case of associativity and commutativity axioms. In practice, this means that symbols can be marked to be associative and commutative.

A labeled rewrite rule in ELAN is obviously defined as a pair of terms built on functional symbols and local variables. In addition, rules can be conditional and a kind of local assignment can be used inside the rules. The purpose of local assignment is mainly to introduce strategy control on subterms. Rules are named by labels that are just identifiers, but variables are also specified. We shall see later in examples that more than one rule can be labeled by the same label. The concrete syntax of a labeled rewrite rule is :

<rwrule> ::= **rule** *<rule name>* **for** *<sort>*
declare { *<local variable declaration>* }$^+$
body *<rbody>*
end of rule

<rbody> ::= *<term>* **=>** *<term>*

<rbody> ::= *<rbody>* **if** *<boolean term>*

<rbody> ::= *<rbody>* **where** *<variable>* **:=** ([*<strategy name>*]) *<term>*

Here and in the rest of the paper, we use a version of BNF-notation to express concrete syntax. In this notation, the expressions written in italics and enclosed into <> parentheses are nonterminal symbols; the reserved words are written in bold, as well as special symbols from the language; an expression enclosed into $\{\ \}^*$ parentheses is iterated zero and more times; an expression enclosed into $\{\ \}^+$ parentheses is iterated one and more times, and finally an expression enclosed into [ ] brackets can be omitted.

A successful application of a rewrite rule on a given term is processed in the expected way: the left-hand side of the rule has to match a subterm of the given term; this causes instantiation of variables occurring in the left-hand side. Then the local assignments and conditions are evaluated, in reverse order, so that the evaluation starts with the last one and stops with the first one. The conditions have to be evaluated to a constant *true* and the local assignments must instantiate all remaining variables. At the end, the original subterm is replaced by the instantiated right-hand side of the rule. In the **where** assignments, the terms <*term*> are reduced with respect to strategies named <*strategy name*> and the results are then assigned to the variables on the left-hand sides of the **:=** assignement signs.

We can consider local assignment as syntactic sugar, because, for example, using a rule of the form:

$$\ell : \quad g \quad => \quad d[x]_\omega \quad \textbf{where} \;\; x := (s) \;\; t$$

in a strategy $S[\ell]_v$ is equivalent to using the rule:

$$\ell' : \quad g \quad => \quad d[t]_\omega$$

with ℓ' a new label, in the strategy $S[(\ell'; d[s]_\omega)]_v$.

1.5.3 Expressing strategies in ELAN

Strategy definitions in ELAN are based on the expressions defined in Section 1.3.3, with a few more specific points. Based on practical experience, rewrite rules are divided into two groups: the labeled rewrite rules, whose application is controlled by user-defined strategies, and the non labeled rules, whose application is controlled by a default normalization strategy, which consists in applying non-labeled rules at any position of a term until the normal form is reached. The normalization strategy is used with rules defining functions and is applied after each reduction produced by a labeled rewrite rule. This corresponds to the intuition that, after each application of a labeled rewrite rule, functions introduced by the right-hand side of rule must be evaluated.

In the current ELAN implementation, strategies built up from functional symbols application and strategies calling other strategies are not yet implemented. But this is not really a limitation since these constructions can be simulated using the **where** local assignment. On the other hand, mainly for efficiency reasons, we have introduced deterministic strategies in our language.

The syntax for describing strategies in ELAN is the following:

<*strategy def*> ::= **strategy** <*strategy name*>
<*strategy*>
end of strategy

<*strategy*> ::= **dont know choose** ({ <*rule name*> }$^+$)
| **dont know choose** ({ <*strategy*> , }* <*strategy*>)
| **dont care choose** ({ <*rule name*> }$^+$)
| **dont care choose** ({ <*strategy*> , }* <*strategy*>)
| **iterate** <*strategy*> **enditerate**
| **repeat** <*strategy*> **endrepeat**
| <*strategy*> <*strategy*>

In this syntax, concatenation is omitted, the constructor **iterate** corresponds to the iterator and the **dont know choose** constructor corresponds to the selecting operator. Note that rule names are always encapsulated in the selecting operator. This is because a rule name represents in general a set of rules, so a selecting operator has always to be specified. The **dont care choose** constructor is what we call a deterministic strategy. It corresponds to the selecting operator, but contrary to **dont know choose**, it selects only one of substrategies giving a non-empty set of results. Another deterministic strategy is the **repeat** strategy, which corresponds to the iterate operator, but only the largest chains of iterations are considered. This means that only proof terms to which no other substrategy can be concatenated, are allowed. These two deterministic strategies are used to significantly restrict the search space and to drop redundant results. For example, in the case of syntactic unification, we know that unification rules can be applied in any order and that all solved forms are equivalent.

1.5.4 Horn clause programming language

To illustrate the ELAN environment, we define in this section a simple programming language based on Horn clause logic and we describe the top level of its implementation.

This allows us to introduce an important point: there are two kinds of users of ELAN. The first ones, let us call them super-users, are those describing in ELAN a computational system in order to design for example a logic programming language. The second ones are those using ELAN *and* a (super-user defined) computational system in order to evaluate a query in the context of a given specification.

In Horn clause logic, formulas are of the form $A \Leftarrow B_1, \cdots, B_n$ with A, B_i being atoms, and a theory is given by a signature and a set of formulas. An example of Horn clause theory to be given as input to ELAN (where a theory is called specification) is:

```
specification member
  signature
    vars        X , Y , Z
    ops         cons:2, nil:0, a:0, b:0, c:0
    predicates  member:2
  clauses
    member(X,cons(X,Y)).
    member(X,cons(Y,Z)) :- member(X,Z).
end of specification
```

The syntax of a specification is fully defined by the ELAN super-user; the heading of a specification and the division on named parts (here *signature* and *clauses*) have to be respected by the user, since they are part of the Horn clause programming language syntax.

An implementation of this programming language must be able to solve queries of type: $\exists \overline{x} P(\overline{x})$ where the quantifier can be omitted under the assumption that all variables occurring in P are implicitly existentially quantified. The expected proof calculus must correspond to SLD resolution and give a complete set of substitutions satisfying the query.

Intuitively, our intention is to specify a computational system with a strategy which, applied on the initial term $(P(\overline{x}) \mid\mid true)$, produces a set of final terms of form $(nil \mid\mid \sigma)$, where σ represents a solution. The rewrite rules of the computational system apply to an execution state that is a formula made of a first component which is a list of goals, initially reduced to one element $P(\overline{x})$ and a second component which is a conjunction of equalities in solved forms, initially reduced to an empty conjunction denoted $true$.

The main rule of the computation system is:

$$(P, x \mid\mid \theta) \Rightarrow (B_1, \cdots, B_n, x \mid\mid \theta \circ \theta') \qquad \textbf{where } \theta' = mgu(\theta P, \theta A)$$

for a variant with fresh variables of any clause $A \Leftarrow B_1, \cdots, B_n$ in the specification. This rule is written below in ELAN syntax. The quantification **FOR EACH** $\cdots$ { <*rule*> } means that <*rule*> will be generated for all clauses in the specification with an atom A as head and a list of atoms B as body. The infix operator @ used inside the rule is an append function on lists; the variable sigma is set to a renaming substitution which is applied on the clause $A \Leftarrow B$ to obtain a variant with fresh variables; the variable RRS denotes a set of equations which is checked to be satisfiable by the strategy p_unifys.

```
FOR EACH C:clause; A:pred_atom; B:pred_atom_list
        SUCH THAT clause(C) & head(C,A) & body(C,B) : {

            rule trans for ex_state
            declare RS,RRS : system;
                    rest : pred_atom_list;
                    h : pred_atom;
                    sigma : substitution;
            body
               (h,rest || RS) => ( sigma(B) @ rest || RRS)
                    where RRS := (p_unifys) sigma(A)=h & RS
                    where sigma := () rename_subst from vars (A,B) w.r.t. h,rest
            end of rule

}
```

If after some applications of this rule trans, the empty list of goals is obtained, then a solution is reached. To test emptiness of the list, we add the rule:

```
rule trans_successfin for ex_state
declare sigma1,sigma2 : system;
body
   ( nil || sigma1) => ( nil || sigma2)
          where sigma2 := (delete_new_vars) sigma1
end of rule
```

which also deletes non interesting variables (i.e. variables added during execution) from the result.

Of course, this is only the top level description of the computational system, since all details of the implementation of e.g. replacement, substitution, unification, have to be given. This is done using appropriate computational systems, and thus the description of this simple Prolog interpreter (as well as other examples) reduces completely to a conditional term rewriting system.

We finish this example by giving a strategy defining the application of these two rules.

```
strategy eval_strat
     iterate
          dont know choose ( trans )
     enditerate
     dont know choose ( trans_successfin )
end of strategy
```

1.5.5 Reflexivity

In order to illustrate two different ways to encode rewriting in the rewriting logic itself, let us assume now that we wish to specify an equational logic language, where formulas are rewrite rules forming a convergent term rewriting system and whose proof calculus is given by normalization of terms. An example of program in this language is:

```
specification rwspec
  signature
    Vars: X, Y, Z
    Ops: nil, cons:2, append:2, a, b, c
  rwrules
    append(cons(Y,Z),X) -> cons(Y,append(Z,X)),
    append(nil,X) -> X
end of specification
```

This means that the user wants to define variables X, Y, Z and an operators nil, a, b, c of arity zero (by default), and binary operators cons and append that satisfy the usual semantics of lists.

There are two ways for encoding term rewriting in ELAN. The first one consists in describing the steps of matching, substituting and replacement involved in an elementary rewrite step and how to iterate them. In this case, the implementor has to specify completely all details of the implementation of rewriting. But ELAN itself contains the notion of rewriting, so the second possibility is to simply transfer the rewrite rules from the specification level to the execution level.

The first situation is for example expressed by the following **rewrite** rewrite rule who describes how to apply the rule $L \rightarrow R$ at occurrence *occ* on the term s, with the substitution $theta$.

```
FOR EACH L,R:term SUCH THAT rwrule(L,R) : {

  rule rewrite for term
  declare ME : system; s : term; theta : substitution; occ : occurrence;
  body
    s => s[theta(R)] at occ
          where theta := () system_to_subst(ME)
          where ME := (matchs) s at occ = L
          where occ := (occ) non-var-occ(s)
  end of rule

}

strategy rewriting
    repeat
        dont care choose  ( rewrite )
    endrepeat
end of strategy
```

The strategy expresses the fact that rewrite rules must be applied until the normal form is reached. In this case, the following call of ELAN which uses the computational system described in the file `rewriting` and the specification described in the file `rwspec` returns, when reducing the term $append(cons(a, cons(b, nil)), cons(c, nil))$:

```
--> elan rewriting rwspec
        ******** ELAN-Nancy v0.08 (10/06/93.22:50) ********
...
enter query term:
append(cons(a,cons(b,nil)),cons(c,nil))
[] start with term : append(cons(a,cons(b,nil)),cons(c,nil))

[] result term : cons(a,cons(b,cons(c,nil)))

 number of labeled rules applied: 225
 number of labeled rules tried: 3032
 number of non labeled rules applied: 364
 time: 5.483333 secs
-->
```

On the contrary, we can make an explicit use of rewriting capabilities of ELAN by describing the rewriting in the following reflexive way:

FOR EACH L,R:term **SUCH THAT** rwrule(L,R) : {

> **rule for** term
> **declare FOR EACH** X:identifier **SUCH THAT** var(X): { X : term ; }
> **body**
> L **=>** R
> **end of rule**

}

which simply consists to internalize any rewrite rule $L \rightarrow R$ of the specification into an ELAN rewrite rule. The line:

declare FOR EACH X:identifier **SUCH THAT** var(X): { X : term ; }

means that each identifier X, Y, Z declared as variable in the specification rwspec is now declared as a variable of sort term in the rewrite rule $L \Rightarrow R$. Then calling ELAN on the same query solved in this new description gives the same answer in a quite different manner!

```
--> elan intRewriting rwspec
        ******** ELAN-Nancy v0.08 (10/06/93.22:50) ********
...
enter query term:
append(cons(a,cons(b,nil)),cons(c,nil))
[] start with term : append(cons(a,cons(b,nil)),cons(c,nil))

[] result term : cons(a,cons(b,cons(c,nil)))

 number of labeled rules applied: 0
 number of labeled rules tried: 0
 number of non labeled rules applied: 3
 time: 0.000000 secs
-->
```

As quoted also in [MOM93], these two translations of rewriting into a computational system correspond to the identity functor in the second case and an explicit description of the rewrite process in the first case.

1.6 Further perspectives and conclusion

Up to now, we have expressed in a uniform framework constraint solving and deduction and achieved a high level of modularity. Our objective of designing and experimenting with new computational systems leads us to consider different combination problems. Let us mention two kinds of theoretical problems that arise in this context:

- Putting together a computational system for equality and constraint solving like unification leads to interesting but non-trivial problems from the point of view of their properties. Performing deduction independently on equality formulas and on constraints is not complete in general. This problem is solved in a general way in [KKR90] by restructuring formulas and constraints. Such restructuration has been shown useless with additional syntactic restrictions on formulas [BGLS92].
- Putting together two constraint solvers in two different first-order structures like for instance booleans and a quotient term algebra cannot be reduced to a blind use of both constraint solvers. Typically interactions between theories may appear and must be solved [BS92, KR92].

In the challenge of specifying safe software and executing them efficiently, it is of great importance to present in the same framework, the operational semantics of a programming language, the prover (as far as possible automated) of desired properties of programs, and the efficient execution of the proved safe programs. The constraint paradigm provides a logical foundation to efficient improvements of the deductive process both in programming languages and theorem provers. We believe that the similarity mentioned in the introduction between specifying, proving and computing can rely on the notion of computational systems.

As already emphasized, we expect the ELAN system to provide a uniform framework for describing computational systems, experimenting new combinations and performing proofs of their properties. Completion-based approaches implemented in theorem provers like

REVE, RRL or OTTER, to prove properties such as confluence or consistency of enrichments, can also be designed in the conceptual setting of rewrite rules and strategies [Bac91]. However, this view is not to obtain efficient theorem provers at once, but rather first to allow for experiments and designs of better proof strategies.

A complementary direction is worth considering to get high efficiency: this is the parallelisation of deduction obtained for free through parallelisation of rewriting (see [KV92, Mes92]). This argument that needs further investigations is an additional point in favor of the choice of the evaluation mechanism of ELAN.

Indeed some aspects of this work can be compared with other logical frameworks like ELF, proof development systems like NuPRL and interactive theorem provers like Isabelle, Clam or 2OBJ. However we think that, because of its emphasized ability to express, execute and combine computational systems and the crucial role of constraint solving, ELAN provides interesting and original features.

Acknowledgments: This work took great benefit from discussions with José Meseguer and we sincerely thank him for illuminating explanations on rewriting logic and stimulating comments on our own approach. We also thank Patrick Viry for his very pertinent comments on the previous version of this paper.

Bibliography

[Bac91] L. Bachmair. *Canonical equational proofs.* Computer Science Logic, Progress in Theoretical Computer Science. Birkhäuser Verlag AG, 1991.

[BGLS92] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In *Proceedings 11th International Conference on Automated Deduction, Saratoga Springs (N.Y., USA)*, pages 462–476, 1992.

[BS92] F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *Proceedings 11th International Conference on Automated Deduction, Saratoga Springs (N.Y., USA)*, pages 50–65, 1992.

[Com90] H. Comon. Equational formulas in order-sorted algebras. In Paterson, editor, *Proceedings ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 674–688. Springer-Verlag, 1990.

[Com91] H. Comon. Disunification: a survey. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9, pages 322–359. MIT Press, Cambridge (MA, USA), 1991.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leuven, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[DP88] N. Dershowitz and D. A. Plaisted. Equational programming. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11: The logic and acquisition of knowledge*, chapter 2, pages 21–56. Oxford Press, Oxford, 1988.

[GKK+87] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987.

[GM86] J. A. Goguen and J. Meseguer. EQLOG: Equality, types and generic modules for logic programming. In D. De Groot and G. Lindstrom, editors, *Logic Programming. Functions, relations and equations*, pages 295–364. Prentice-Hall, 1986.

[Gog87] J. A. Goguen. The rewrite rule machine project. In *Proceedings of the second international conference on supercomputing*, Santa Clara, California, May 1987.

[HL91] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, I. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic*, chapter 11, pages 395–414. The MIT Press, 1991.

[Hul80] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, July 1980.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. MIT Press, Cambridge (MA, USA), 1991.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium pn Principles Of Programming Languages*, pages 111–119, Munich (Germany), 1987.

[Kah87] G. Kahn. Natural semantics. Technical Report 601, INRIA Sophia-Antipolis, February 1987.

[Kir85] C. Kirchner. *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles.* Thèse de Doctorat d'Etat, Université de Nancy I, 1985.

[KKR90] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[KKV93] C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In P. Kanellakis, J.-L. Lassez, and V. Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming, Providence (R.I., USA)*, pages 166–175. Brown University, 1993.

[KR92] H. Kirchner and C. Ringeissen. A constraint solver in finite algebras and its combination with unification algorithms. In K. Apt, editor, *Proc. Joint International Conference and Symposium on Logic Programming*, pages 225–239. MIT Press, 1992.

[KV92] C. Kirchner and P. Viry. Implementing parallel rewriting. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 123–138. Springer-Verlag, 1992.

[LC90] P. Le Chenadec. On the logic of unification. In C. Kirchner, editor, *Unification*, pages 157–216. Academic Press, London, 1990.

[Mes89] J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MNRA92] Moreno-Navarro and Rodriquez-Artalejo. Logic programming with functions and predicates: the language BABEL. *Journal of Logic Programming*, 12(3):191–223, February 1992.

[MOM93] N. Martì-Oliet and J. Meseguer. Rewriting logic as a logical and semantical framework. Technical report, SRI International, May 1993.

[NR92] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In D. Kapur, editor, *Proceedings of CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 477–491. Springer-Verlag, 1992.

[NRS90] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. In C. Kirchner, editor, *Unification*, pages 517–540. Academic Press, London, 1990.

[Rin93] C. Ringeissen. *Combinaison de Résolutions de Contraintes.* Thèse de Doctorat d'Université, Université de Nancy 1, December 1993.

[Vit94] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université de Nancy 1, to appear October 1994.