Implementing Computational Systems with Constraints

Claude Kirchner **Hélène Kirchner** **Marian Vittek**

INRIA-Lorraine & CRIN
615, rue du Jardin Botanique, BP 101
54602 Villers-les-Nancy Cedex FRANCE
E-mail: kirchner@loria.fr, vittek@loria.fr

Abstract

The paper presents a framework to describe, experiment and study the combination of different computational systems including the constraint solving paradigm. Computational systems are interpreted in a first-order setting thanks to an evaluator that rewrites formulas.

1 Introduction

Logic programming in a broad sense, theorem proving and constraint solving have in common their foundation on a computational system, described by the syntax of formulas, a set of axioms, a set of deduction rules whose application is governed by some strategies, altogether with one or a class of interpretations. All these activities may be formulated as instances of a common schema that consists of applying deduction rules on formulas with some strategy, until getting specific forms. In logic programming the emphasis is on efficiency in reaching a successful solved form characterising 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.

We claim that this similarity is a major advantage for the design of safe programming environments where programs in high-level programming languages can be developed and some of their properties can be proved. Moreover we believe that the implementation of such an environment can rely on a very simple theoretical setting: first-order equational logic. This means that we see formulas, sets of formulas and proofs as first-order terms. We see a deduction rule as a rewrite rule or a transition rule, that is a rewrite rule applying at the outermost position of such terms. Application of deduction rules is rewriting, possibly under some conditions. We think that this framework has several advantages:
— it is conceptually simple.
— we benefit from the work on rewriting and efficient implementation techniques.
— we can define rewriting in the language itself, so bootstrap the system.
— more important, we can reason about computational systems, for instance prove abstract properties of the deduction rules like termination or confluence with rewrite-based techniques. We can also reason about combination and enrichment of computational systems and build new paradigms in a modular way, once we know how systems theoretically interact.

To validate these ideas, we develop a prototype[1] called ELAN, to describe, experiment and study the combination of different computational systems that provide basis for logic programming paradigms. Our favorite starting example, developed in this paper, is the combination of constraint solving with computational systems based on rewriting and narrowing.

---

[1]This prototype has been partly designed in collaboration with the DEMONS group in Orsay and supported by GRECO PAOIA of CNRS, and Esprit BRA CCL.

2 Conceptual setting

The conceptual setting is given by the notion of computational system whose evaluation mechanism is based on application of transition and rewrite rules. This provides uniformity of the execution principle which is first-order. The notations and definitions used in this paper are consistent with [DJ90, JK91].

In our approach, a **computational system** is given by a syntactic part and a computational part.

- The syntax is defined by
  – a class $Sign$ of signatures; each signature $\Sigma \in Sign$ defines a set of sort symbols, a set of function symbols and a set of predicate symbols.
  For each signature $\Sigma \in Sign$,
  — $stat(\Sigma)$ denotes the sets of statements that are sentences (well-formed formulas) built on this signature.
  Each pair $P = (\Sigma, \Gamma)$ where $\Sigma \in Sign$ and $\Gamma \in stat(\Sigma)$ defines an axiomatic theory.
  — $quer(\Sigma)$ denotes the queries that are sentences built on the signature Σ.

- The computational part is defined by deduction rules and their strategy of application. These deduction rules are aimed to transform queries into "simpler" queries in the sense that they can easily be solved in the axiomatic theory $P = (\Sigma, \Gamma)$. Indeed it is assumed that the deduction rules and their application strategy have been proved correct and complete w.r.t. an adequate semantics. We do not focuss here on this theoretical point but only consider examples where such correctness and completeness results are available.

  A **computation** is a sequence of terms $t_0 \Rightarrow_{r_0} t_1 \cdots \Rightarrow_{r_{n-1}} t_n$, where each t_{i+1} results from the application of the transition rule r_i on t_i. A sequence of names of transition rules is a strategy. More precisely, **a strategy** is a set of such sequences described by a regular expression built on the names of deduction rules using concatenation, iteration and selecting operators. This regular expression (strategy) describes the set of all admissible computations. A computation $t_0 \Rightarrow_{r_0} t_1 \cdots \Rightarrow_{r_{n-1}} t_n$ is admissible w.r.t. the strategy s iff the word "$r_0 \cdots r_{n-1}$" is in the language $SLan(s)$ described by the regular expression s. t_n is called the result of this computation w.r.t. s. From this definition we see that more than one computation starting from the initial term t_0 can belong to $SLan(s)$; in this case the strategy of computation s is nondeterministic and its result is the multiset of results for all admissible computations. On the other hand, when there is no admissible computation starting from t_0, the result of the strategy s is an empty multiset.

For example, a **Horn clause computational system** is defined by chosing $Sign$ as the class of first-order signatures, $stat$ as sets of formulas of the form $(\forall X : A \Leftarrow B_1, \cdots, B_n)$, $quer$ as formulas of the form $(\exists X : G_1, \cdots, G_n)$ where X is a set of variables, A, B_i, G_k are atoms. SLD resolution can be described using deduction rules that produce, from the negation of the query, the empty clause which is trivially unsatisfiable. A correct and complete strategy must include all computations leading to the empty clause.

Another example is an **equational computational system** defined by choosing $Sign$ as the class of first-order signatures with "$=$" and "$\rightarrow$" being the only predicate symbols, $stat$ as all sets R of universally quantified formulas of the form $(\forall X : l \rightarrow r)$, such that R is a convergent term rewriting system, $quer$ as formulas of the form $(\exists X : g = d \parallel S)$, in which S is a possibly empty conjunction of equations denoted $\bigwedge(t =_\emptyset t')$, which are solved with syntactic unification (also called unification in the empty theory). The basic narrowing process [JK91] can be used to solve such queries, provided that R is a convergent term rewriting system. Let us briefly explain this process. Starting from a query of the form $(\exists\emptyset : g = d \parallel \top)$, where $\top$ denotes the empty conjunction of equations, the basic narrowing process tries to unify a subterm of g at position ω with a left-hand side of a rewrite rule $(l \rightarrow r) \in R$, taking care that its variables are disjoint from those of g. This gives rise to a unification problem $(g_{|\omega} =_\emptyset l)$ and a new equation $(g[r]_\omega = d)$. So the query $(\exists X : g = d \parallel S)$ is transformed into another one $(\exists X \cup \mathcal{V}ar(l \rightarrow r),\ g[r]_\omega = d \parallel S \wedge (g_{|\omega} =_\emptyset l))$ such that the two systems of equations $(\exists X : g = d \wedge S)$ and $(\exists X \cup \mathcal{V}ar(l \rightarrow r),\ g[r]_\omega = d \wedge S \wedge (g_{|\omega} =_\emptyset l))$ have the same set of solutions in the theory defined by R. Indeed a query $(\exists X : g = d \parallel S)$ 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 R into a unification problem in the empty theory. Indeed there is a lot of undeterminism in the choices of the rewrite rule $(l \rightarrow r)$ and the position ω. All possible choices have to be explored when completeness of the returned set of solutions is required (see [NRS89]).

Accordingly, basic narrowing is described with two deduction rules:

Narrow
$(\exists X, g = d \parallel S)$
$\mapsto\!\!\!\twoheadrightarrow$
$(\exists X \cup \mathcal{V}ar(l \to r),\ g[r]_\omega = d \parallel S \wedge (g_{|\omega} =_\emptyset l))$
if $l \to r \in Variants(R)$ and $S \wedge (g_{|\omega} =_\emptyset l)$ satisfiable

Block
$(\exists X, g = d \parallel S)$
$\mapsto\!\!\!\twoheadrightarrow$
$(\exists X, \top \parallel S \wedge (g =_\emptyset d))$
if $(S \wedge g =_\emptyset d)$ satisfiable

In this example, S is a conjunction of equational constraints solved in the empty theory. In general, S could be solved in any equational theory, for instance with associativity and comutativity axioms, or even in a specific algebra, for instance a boolean algebra. The previous deduction rules are actually parameterized by the constraint solving process. The next step is to describe it also as a computational system.

3 Constraint solvers as specific 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. This is also the starting point for incorporating constraint solving in our framework. A constraint solver for symbolic constraints such as equations, inequations or disequations on terms is also viewed as a computational system aimed at computing solved forms for the class of considered formulas called constraints.

For instance, **unification** in commutative theories is a constraint solving process that can be described as the computational system where *Sign* is the set of first-order signatures with equality in equivalence classes, denoted $=_C$, as the only predicate (C stands for commutative), *stat* is empty, *quer* is the set of formulas of the form $(t_1 =_C t_2)$. Deduction rules have been proposed and proved correct and complete w.r.t. the computation of a complete set of C-unifiers (see [JK91] for instance). These rules transform conjunctions of equations of the form $(P \wedge s =_C t)$ and include for instance the following ones for a commutative operator $+$:

ComDecompose
$P \wedge s_1 + s_2 =_C t_1 + t_2$
$\mapsto\!\!\!\twoheadrightarrow$
$P \wedge s_1 =_C t_1 \wedge s_2 =_C t_2$

ComMutate
$P \wedge s_1 + s_2 =_C t_1 + t_2$
$\mapsto\!\!\!\twoheadrightarrow$
$P \wedge s_1 =_C t_2 \wedge s_2 =_C t_1$

In general, a language of constraints is defined by a class of signatures defining the syntax of the constraints, the set of queries which are the constraints to solve in this language and a constraint solver. We are interested here in constraint solving processes that can be described with transition rules that compute solved forms of constraints. This includes for now constraint languages built from elementary constraints that may be equations (as above), disequations [Com91], inequations [Com90] on terms expressed with simplification orderings, membership constraints [CD91].

This view 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 strategy, the solved forms are in general reached more efficiently with smart choices of rules. Through the design 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 eventually partly automated. This also opens the door to non-trivial applications of termination proof techniques.
3. 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 transition rules on formulas with constraints, but we shall see later on in Section 5, that such a combination may also generate delicate problems.

4 The ELAN interpreter

ELAN is an environment for designing computational systems, and evaluating queries written in this system. An executable entity for the ELAN interpreter is given by:

A computational system description that provides the syntax, the deduction rules and the strategy of the computational system which is defined.

Specifications that are axiomatic theories written in the syntax of the defined computational system and defining their own signature and set of axioms.

Queries that express the problem to solve in a given specification and can be seen as input values for the computation.

A specification interpreted in a given computational system produces an executable program, which is able to evaluate a query w.r.t. the given computational system.

To achieve our goal of a uniform yet flexible system, the most delicate points were:

- to express the syntax of any computational system that the user wants to define. In this place, ELAN provides a parser for any context-free grammar. This gives the ability to express grammars in a natural way, and in particular to describe mixfix syntax.
- to allow a high degree of modularity. Computational system descriptions for the ELAN interpreter are built from elementary pieces that are modules. **A module** can import other modules and corresponds to a computational system. It defines its own signature, that is the symbols used to express the syntax of statements and queries, but also function symbols used to express transition rules. It defines its own transition rules and rewrite rules useful to express functions used by transition rules. It defines also strategies for applying these transition rules, that may be deterministic or non-deterministic.
- to express the evaluation mechanism of programs for given inputs, i.e. the mechanism for solving the queries. To evaluate a query, the ELAN interpreter repeatedly applies the transition rules according to the strategy, until the strategy is exhausted. A transition rule is implemented as a rewrite rule that can be applied only at the outermost position of a term and whose application is conditionned by a kind of local assignment. Since our choice is to describe computations by transition rules controlled by strategies, a convenient and powerful language to express strategies is needed. ELAN provides in particular don't-know and don't-care non-determinism, so the possibility to express backward and forward chaining for the deductions.

In the following part of this section, we illustrate the capabilities of ELAN on an example. Let us consider again the case of an equational computational system with narrowing as deduction process. Programs in this simple language are algebraic specifications consisting in the definition of a signature and axioms given as rewrite rules. In Figure 1, a specification of lists is given.

```
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
```

Figure 1:

A query is then a single equation between two terms built on this signature.

A computational system description has to define the syntax of the specification and of the query as well as how the query is solved. In a computational system description, all the notions of terms, equations, systems of equations have to be specified explicitly. Thanks to modularity, it is indeed possible to reuse a library of modules with mostly used constructions.

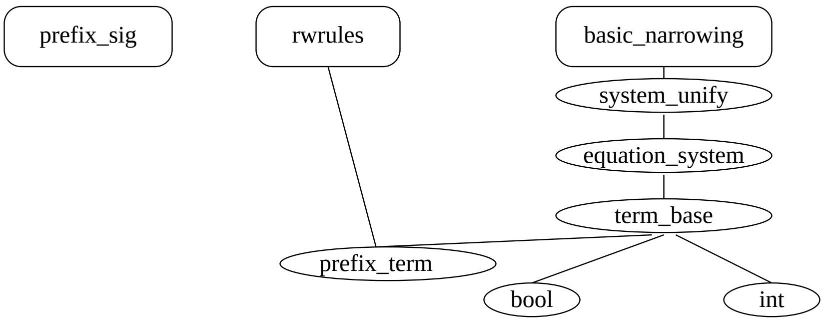

Figure 2:

For instance, defining the signature part of a specification needs to define what are functional symbols, variables, arities, as well as the syntactic form of a function declaration in the specification. This is done in a module *prefix_ sig*. Similarly, defining the rewrite rules part of a specification needs to define a notion of term (here in the module *prefix_ term*) and the notion of rewrite rule (in the module *rwrules*). The module *prefix_ term* is reused by the module *basic_ narrowing* where a query and its execution are defined. Figure 2 shows all modules used in the narrowing example with their import relations.

We do not explain here the construction of the concrete syntax of a specification and how it is processed by the system (this is described in [Vit93]), but rather focus on the processing of queries.

4.1 Transition Rules

As already said, the main ELAN mechanism to process a query is application of transition rules. The syntax for transition rules in ELAN is:

```
<trans rule> ::= transition rule <name of rule> for <type of terms>
                 declare <local variable declarations>
                 body
                   <rule body>
                 end of rule
```

where

```
<rule body> ::=  <term> => <term>
             |    <rule body> where <variable> := ( <strategy> ) <term>
```

The `where` construction is a local assignment which assigns to the variable `<variable>` the result of applying the strategy `<strategy>` to the term `<term>`. If the strategy is nondeterministic, then all results are considered. More precisely, let r be a rule body, x a local variable, s a strategy and t a term. Then the rule

$$r \text{ where } x{:=}\ (s)\ t$$

conceptually represents the set of rules, all with the same name, that are instances of r of the form $(x \mapsto t')r$ for any term t' resulting from the computation of t with respect to s.

A first example of transition rules, without the `where` construction, is given by the implementation of the rules **ComDecompose** and **ComMutate**, for which the translation in ELAN is straightforward:

```
transition rule ComDecompose for system
declare P : system;
        s_1,s_2,t_1,t_2 : term;
body
  P & s_1+s_2 = t_1+t_2        =>       P & s_1 = t_1 & s_2 = t_2
end of rule

transition rule ComMutate for system
declare P : system;
        s_1,s_2,t_1,t_2 : term;
body
  P & s_1+s_2 = t_1+t_2        =>       P & s_1 = t_2 & s_2 = t_1
end of rule
```

In general, functions in ELAN can be written in mixfix notation; in the last rule we have used the operators "`& + =`" in infix form.

The rule **Narrow** from the narrowing process provides a more complex example, that illustrates several features of ELAN. It will be written as:

```
IN rwrules: FOR EACH L,R:term SUCH THAT rwrule(L,R) : {
  transition rule Narrow for b_narr_state
  declare cc,c1   : system;
          g,d,l,r : term;
          sigma   : substitution;
          oc      : occurrence;
  body
    (g = d || cc)       => ( g[r] at oc = d || c1)
      where c1 := (unifys) g at oc = l & cc
      where oc := (occ) non-var-occ(g)
      where l :=  () sigma(L)
      where r :=  () sigma(R)
      where sigma := () rename-subst(L, g = d & cc)
  end of rule
}
```

The mixfix operators used in this rule have the following meanings: applying a substitution *sigma* on a term t is written "$sigma(t)$"; the subterm of t at position *oc* is written "t *at oc*"; the term produced by replacing a subterm of t at position *oc* by the term r is written "$t[r]$ *at oc*". All these functions are imported from other modules.

The quantification `IN rwrules: FOR EACH L,R:term SUCH THAT rwrule(L,R): { <rule> }` means that `<rule>` will be generated for all instances L,R such that `L->R` is a rewrite rule defined in the specification.

All these rules have the name **Narrow**. For instance two such rules will be generated in the list specification given in Figure 1.

During the application of this transition rule, the variables are successively assigned to the following values: *sigma* to a renaming substitution which introduces fresh variables not occurring in the current system; l and r to left and right-hand side of the rewrite rule `L->R` after variable renaming; *oc* is any non-variable position in the term g (**occ** is a implemented by a non-deterministic strategy); *c1* is a solved form of the unification problem $g_{|oc} = l$ & cc. The strategy **unifys** returns a solved form just in the case where the original system is satisfiable. It may be worth emphasizing that the transition rule only applies when every local variable has got a result different from the empty set. Else application of the transition rule fails.

For the sake of completeness, we can write in the same way the rule **Block**.

```
transition rule Block for b_narr_state
declare SF,cc: system;
        g,d: term;
body
  (g=d || cc)    => ( top || SF )
        where SF := (unifys) g=d & cc
end of rule
```

4.2 Strategies

Coming back to the narrowing example, we can observe that after n applications ($n \geq 0$) of the rule **Narrow** followed by one application of rule **Block**, the resulting term is of the form $(\top \parallel S)$ where S provides a solution for the starting equational query. On the other hand, the completeness result for narrowing states that a complete set of R-unifiers is obtained by gathering all solutions that can be computed as a result of n applications of rule **Narrow** for some $n \geq 0$, followed by an application of **Block**. Accordingly, the set of successful computations, that is computations leading to a solution, is described by the following regular expression:

$$(\textbf{Narrow})^* \cdot \textbf{Block}$$

where the $\cdot$ is the concatenation operator, and $()^*$ is an iteration operator. So we consider this expression as describing a correct and complete strategy for narrowing.

A partial concrete syntax for strategies in ELAN is:

```
<strategy> ::= iterate <strategy> enditerate
           |   dont know choose( <list of <strategy>>  )
           |   dont know choose( <list of <rule name>>  )
           |   <strategy> <strategy>
```

In this syntax, concatenation is omitted, the constructor `iterate` corresponds to the iterator $()^*$ and the `dont know choose` constructor corresponds to a selecting operator. Note that names of rules are always encapsulated in the selecting operator. This is because the rule name represents in general a set of rules, so a selecting operator has always to be specified. The full ELAN strategy language is of course richer and includes constructions for dont-care strategies, which is interesting especially from the efficiency point of view and to formalize forward chaining computations.

Since narrowing can be described by a regular expression $(\textbf{Narrow})^* \cdot \textbf{Block}$, the strategy for our example can be implemented as follows in ELAN:

```
strategy b_narrow
  iterate
    dont know choose(Narrow)
  enditerate
  dont know choose(Block)
end of strategy
```

This strategy will generate application of the transition rule `Block`, then of `Narrow` followed by `Block`, then of twice `Narrow` followed by `Block` and so on. This leads to a correct implementation of narrowing in ELAN.

Regular expressions can seem to be very weak to express computational strategies, but their power is largely increased by using the `where` construction inside the rules and by the definition of admissible computations given in Section 2. According to this definition, the `dont know choose` constructor has to try all possible applications of a transition rule in the list given as argument. In our example, this is the way to exhaustively attempt to apply narrowing with all rewrite rules in R and all positions in the query term. The `iterate` constructor then successively tries all possible numbers of iterations. In the ELAN interpreter, backtracking is used to implement a computation following this kind of strategy.

Beyond the case of narrowing, we succeeded for instance to specify commutative unification, rewriting, and a simple PROLOG interpreter.

4.3 Flexibility

ELAN supports computational system descriptions with high level of modularity. Modules are non-parametric and their visibility graph is a directed acyclic graph. Each module can use signatures from modules explicitely introduced in an import part. Aliasing of operators is permitted, so an operator can have more than one name; for example it can be used in mixfix and prefix notation at the same time. Direct importation with aliasing allows providing all operations usual in such modular systems, like renaming, masking and so on.

The flexibility of ELAN can be briefly illustrated on two examples.

— We can straightforwardly replace the syntactic unification in narrowing procedure by unification in another theory. We just need to replace the module *system_unify* by a module providing unification in the desired theory. This can be compared to the $CLP(X)$ framework that could be implemented in ELAN. Each instance would be provided by the choice of a computational system for constraint solving in the computation domain.

— We can also easily change the syntactic form of specification. For example, replacing the module *prefix_sig* by a module *mixfix_sig*, adding a module *mixfix_term* and some minor modifications of *basic_narrowing*, provides mixfix notation at the specification level.

5 Combining computational systems

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 [KKR90]. For instance from two oriented equality with constraints, a new equality can be deduced using the deduction rule

  Deduce
  $$\begin{array}{l} (g \to d \parallel c'), (l \to r \parallel c) \\ \mapsto\!\!\!\twoheadrightarrow \\ (g[r]_m = d \parallel c \wedge c' \wedge (g_{|m} =_\emptyset l)) \\ \text{if } c \wedge c' \wedge (g_{|m} =_\emptyset l) \text{ satisfiable} \end{array}$$

  Unfortunately this deduction rule does not capture all possible equational deductions. Consider for instance the set of function symbols $\mathcal{F} = \{a, c, f, g\}$, and two rewrite rules with equational constraints:

  $$\begin{array}{l} g(a) \to c \\ f(x) \to a \quad \parallel \quad x =_\emptyset g(y) \end{array}$$

  Deduce does not apply although

  $$a \leftarrow f(g(a)) \to f(c)$$

  is a correct deduction performed with the given rewrite rules. This problem is solved in a general way in [KKR90] by restructuring rules and constraints. Such restructuration has been shown useless with additional syntactic restrictions on formulas. For instance it has been proved that Deduce is complete with a limited class of initial constraints [BGLS92, NR92].

- 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].

Such problems are yet an active field of research.

6 Further perspectives and conclusion

As already emphasized, we expect the 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 RRL or OTTER, to prove properties such as confluence or consistency of enrichments, can also be designed in the conceptual setting of transition rules and strategies [Bac91]. Concepts like critical pairs criteria, that appear very useful in the design of efficient provers can be formalised with constraints [KKR90]. This view is not to obtain efficient theorem provers at once, but rather 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.

References

[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 (NY, USA)*, 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 (NY, USA)*, 1992.

[Com90] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4), 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. MIT Press, Cambridge (MA, USA), 1991.

[CD91] H. Comon and C. Delor. Equational formulas with membership constraints. Technical report, Laboratoire de Recherche en informatique, Mar. 1991. To appear in Information and Computation.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers North-Holland, 1990.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th POPL conference*, Munich (Germany), 1987.

[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.

[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.

[KR92] H. Kirchner and C. Ringeissen. A constraint solver in finite algebras and its combination with unification algorithms. In *Proc. Joint International Conference and Symposium on Logic Programming*, 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.

[Mes92] J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming, Volterra (Italy)*, volume 632 of *Lecture Notes in Computer Science*, pages 158–200. Springer-Verlag, September 1992.

[NR92] R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In B. Krieg-Brückner, editor, *Proceedings of ESOP'92*, volume 582 of *Lecture Notes in Computer Science*, pages 371–389. Springer-Verlag, 1992.

[NRS89] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7(3 & 4):295–318, 1989. Special issue on unification. Part one.

[Vit93] M. Vittek. ELAN User's manual. In preparation.