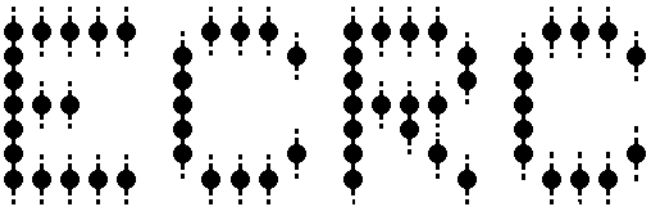
technical report ECRC–92–18

Constraint Simplification Rules

Thom Frühwirth
thom@ecrc.de



ECRC GMBH, ARABELLASTR. 17 D-8000 MÜNCHEN 81, GERMANY - TEL +49 89/926 99 0 - FAX 926 99 170 - TLX 521 6910

For more information about this work, please contact:
Thom Frühwirth
thom@ecrc.de

Abstract

In current constraint logic programming systems, constraint solving is hard-wired in a "black box". We are investigating the use of logic programs to define and implement constraint solvers[1]. The representation of constraint evaluation in the same formalism as the rest of the program greatly facilitates the prototyping, extension, specialization and combination of constraint solvers. In our approach, constraints are specified by definite clauses provided by a *host language*, while constraint evaluation is specified using multi-headed guarded clauses called *constraint simplification rules* (SiRs)[2]. SiRs define determinate conditional rewrite systems that express how conjunctions of constraints simplify. They have been used to encode a range of constraint solvers in our prototype implementation. Additionally, the definite clauses specifying a constraint can be evaluated in the host language, if the constraint is "callable" and no SiR can simplify it further. In this way our approach merges the advantages of constraints (eager simplification by SiRs) and predicates (lazy choices by definite clauses). Consequently our framework provides a tight integration of the logic programming component with user-defined constraint solvers. Furthermore it can make use of any hard-wired solvers already built into the host language. We also present some results obtained using our prototype, a meta-interpreter for Prolog augmented with SiRs.

Keywords: Concurrent Constraint Logic Programming, Constraint Reasoning, Multi-Headed Guarded Clauses, Committed Choice Languages

[1] This is a sligthly revised version of the report of July 1992 which was based on the internal report [Fru91].
[2] Now called constraint handling rules [Fru93, FrHa93].

Contents

1 Introduction 1

1.1 Constraint Logic Programming 1

1.2 Constraint Simplification Rules 2

2 Syntax and Semantics of SiRs 4

2.1 Syntax of SiRs 5

2.2 Declarative Semantics 6

2.3 Operational Semantics 6

3 Reasoning about SiRs 8

3.1 Termination and Confluence 8

4 The Prototype 9

5 An Example Solver for Temporal Time Point Constraints 11

6 Related Work 13

6.1 Constraint Logic Programming Languages 13

6.2 Combined and Extended Languages 13

6.3 Multiple Head Atoms 14

7 Conclusions 15

1 Introduction

The advent of constraints in logic programming is one of the rare cases where both theoretical and practical aspects of a programming language are improved. Programs in constraint logic programming languages [JaLa87, Sar89, Coh90, Fru90, VH91] are both more declarative and more efficient than logic programming languages without constraints. We shortly describe the main ideas behind constraint logic programming and state some problems. Then we introduce constraint simplification rules by example and show how they overcome these problems.

1.1 Constraint Logic Programming

Constraint logic programming (CLP) can be characterized by the interaction of a logic programming system (LP) with a constraint solver (CS). During program execution, the LP incrementally sends constraints and variable substitutions[1] to the CS. The CS tries to solve the constraints. By a solved constraint we mean a constraint that was detected to be unsatisfiable or reduced to a substitution. In the LP the results from the CS cause *a priori* pruning of branches in the search tree spawned by the program. Unsatisfiability means failure of the current branch, and a substitution reduces the number of possible branches, i.e. choices, to explore via backtracking.

In existing CLP systems, the CS usually is a black box (typically encoded in a low-level programming language) not amenable to reasoning, inspection and modification. The lack of reasoning capabilities means that there is no way to establish correctness of a CS. As the behavior of the CS can neither be inspected by the user nor explained by the computer, debugging of real life constraint logic programs is very hard. It has already been demanded elsewhere that "constraint solvers must be completely changeable by users" (p. 276 in [CAL88]). The lack of declarativeness and flexibility becomes a major obstacle if one wants to

- build a new CS,
- extend the CS with new constraints,
- specialize the CS for a particular application,
- combine constraint solvers.

By contrast, we claim that using logic programs to define constraint solvers allows for reasoning about, inspection and modification of CS. We introduce a language in which the user can build any new CS's, as well as extend, specialize and combine them at will. We call it a *user-defined* CS to distinguish it from a built-in CS. The resulting user-defined CS is incremental and determinate by nature of the language, and should be terminating and correct. Constraint solving can be made incremental by repeatedly simplifying constraints until they are solved. By determinate we mean that the CS commits to every constraint simplification it makes. Otherwise we would not gain anything, as the CS would have to backtrack to undo choices like a Prolog program. Note that a determinate computation does not preclude the presence of choices, but the possibility of backtracking. Therefore it must not matter which choice is made. This computation model is referred to as *don't care nondeterminism*, while Prolog employs *don't know nondeterminism* [Sha89][2]. Finally, we would like to be able to prove that the CS program terminates, is confluent, i.e. produces the same result no matter to what choice one commits, and is correct with regard to a specification of the constraints it solves. It is useful to view SiRs as conditional rewrite systems to establish that they are *canonical*, i.e. terminating and confluent. On the other hand, when viewed as logical formulae, the logical *correctness* of SiRs can be established. We can

[1] Of course, substitutions can be regarded equality constraints.

[2] Logic languages that commit to choices are called *committed choice languages* or *concurrent languages*, as they permit concurrent evaluation of potential choices.

prove the former property by adopting well-known techniques such as termination proofs and unfailing completion from (conditional and constrained) rewriting systems. We can prove the latter property by using techniques from automated theorem proving.

1.2 Constraint Simplification Rules

In this subsection we present an amalgamation of Prolog with SiRs. As an example, we define a user-defined constraint $\leq$. In Prolog, the built-in *predicate* `=<` can only be evaluated if the arguments are known, while the user-defined *constraint* will also handle variable arguments.

```
/* 1.  Call Declaration */

(1) callable X≤Y if bound(X),bound(Y).

/* 2.  Constraint Definition */

(2) X≤Y :- X=<Y. % user-defined constraint calls the built-in predicate

/* 3.  Constraint Simplification Rules */

(3a) X≤X ⇔ true.  % reflexivity

(3b) X≤Y,Y≤X ⇔ X=Y. % identity

(3c) X≤Y,Y≤Z ⇒ X≤Z. % transitivity
```

In clause (2), $\leq$ is defined to call the corresponding built-in comparison predicate `=<`. The predicate definition specifies the user-defined constraint, it is thus called a *constraint definition*. The SiRs of (3) specify how $\leq$ simplifies as a constraint. They implement reflexivity, identity and transitivity of less-than-or-equal in a straightforward way. SiR (3a) states that `X≤X` is logically true. Hence, whenever we see the goal `X≤X` we can `replace` it by `true`. Similarly, SiR (3b) means that if we find `X≤Y` as well as `Y≤X` in the current resolvent, we can replace it by the logically equivalent `X=Y`. SiRs (3a) and (3b) are called *replacement* SiR*s*. SiR (3a) detects satisfiability of a constraint, and SiR (3b) solves a conjunction of constraints returning a substitution. SiR (3c) states that the conjunction `X≤Y,Y≤Z` implies `X≤Z`. We can `add` logical consequences as a redundant constraint. This kind of SiR is called *augmentation* SiR[3]. Note that SiRs (3b) and (3c) have multiple head atoms, a feature that is essential in solving conjunctions of constraints. With single-headed SiRs alone, unsatisfiability of a conjunction of constraints (e.g. `X<Y,Y<X`) could never be detected and global constraint satisfaction could not be achieved.

Redundancy produced by Augmentation SiRs is useful, as the following example shows. Given the query `A≤B,C≤A,B≤C`. The first two goals cause SiR (3c) to fire and add `C≤B` to the resolvent. This new goal together with `B≤C` matches the head of SiR (3b). So the two goals are replaced by `B=C`. The substitution is applied to the rest of the resolvent, `A≤B,C≤A`, resulting in `A≤B,B≤A` where `B=C`. SiR (3b) applies, resulting in `A=B`. The resolvent contains no more inequalities, the simplification stops. The constraint solver we built has solved `A≤B,C≤A,B≤C` and produced the answer `A=B,B=C`.

However, our small constraint solver is not complete, because it does not detect unsatisfiability of constraint goals like `4≤3`. A complete solver for $\leq$ would require an infinite number of SiRs, one for each pair of numbers. Instead we utilize the constraint definition (2) to solve the inequality with known arguments. The call declaration (1) states that we can call `X≤Y` as a predicate if

[3] Replacement and augmentation constraint simplification rules are now called simplification and propagation constraint handling rules [Fru93, FrHa93].

both X and Y are bound. Only if no simplification at all is possible, we choose a callable goal and execute its constraint definition. For example, the query $4 \leq A, A \leq 3$ is augmented with $4 \leq 3$ by SiR (3c). Then no more simplification is possible and $4 \leq 3$ is the only callable constraint. Calling its constraint definition produces a failure and so we know that $4 \leq A, A \leq 3$ is unsatisfiable.

We now extend our solver for $\leq$ by a new user-defined constraint over numbers, max(X,Y,Z), which holds if Z is the maximum of X and Y.

```
callable max(X,Y,Z) if bound(X),bound(Y).

max(X,Y,Y):- X≤Y.

max(X,Y,X):- Y≤X.
```

We add the following simplifications.

```
max(X,X,Z) ⇔ X=Z.

max(X,Y,X) ⇔ Y≤X.

max(X,Y,Y) ⇔ X≤Y.

max(X,Y,Z),X≤Y ⇔ Y=Z,X≤Y.

max(X,Y,Z),Y≤X ⇔ X=Z,Y≤X.

max(X,Y,Z) ⇒ X≤Z,Y≤Z. % invariant and approximation

max(X,Y,Z1),max(X,Y,Z2) ⇔ Z1=Z2,max(X,Y,Z1).  % functional dependency
```

In the query max(A,B,C),max(A,C,D), the first constraint goal is augmented with $A \leq C, B \leq C$. The constraints $A \leq C$,max(A,C,D) are simplified into $C=D, A \leq C$. The new resolvent is max(A,B,C), $B \leq C, C=D, A \leq C$. At this point, no more constraint simplification is possible. There is also no callable goal. Therefore the computation deadlocks. We interpret a deadlocked resolvent as *conditional answer*[4] [VE88]. A conditional answer allows for the finite representation of infinitely many solutions. The conditional answer to our query max(A,B,C),max(A,C,D) is max(A,B,C), $B \leq C, C=D, A \leq C$.

So far, SiRs fire as soon as the head atoms match atoms in the resolvent. Thus SiRs wait for bindings so the matching can be performed. Often, one needs to express more sophisticated firing conditions. We extend the firing conditions from bindings to arbitrary built-in constraints and from matching to checking entailment (implication) of these built-in constraints. This is accomplished by introducing *guards*. As soon as the built-in CS of the host language can determine that a guard holds, i.e. is entailed, the corresponding SiR can be fired.

In our example, let =< be a *built-in constraint* from now on, i.e. there is a built-in constraint solver for inequalities (the user-defined constraint $\leq$ is no longer needed). Then we can replace the SiR max(X,Y,Z),X=<Y ⇔ Y=Z,X=<Y by max(X,Y,Z) ⇔ X=<Y | Y=Z, where X=<Y is the guard. As a consequence, the first SiR becomes obsolete, as the built-in constraint X=<Y in the guard naturally covers the case when X=Y. Contrast this with the *user-defined* constraint $\leq$ in the head of the original SiR that clearly cannot match =. Now max can be defined by SiRs as follows.

```
max(X,Y,Z) ⇔ X=<Y | Y=Z.

max(X,Y,Z) ⇔ Y=<X | X=Z.

max(X,Y,X) ⇔ Y=<X.
```

[4] Also called *qualified answer* in [Va86].

```
max(X,Y,Y) ⇔ X=<Y.

max(X,Y,Z) ⇒ X=<Z,Y=<Z.

max(X,Y,Z1),max(X,Y,Z2) ⇔ Z1=Z2,max(X,Y,Z1).
```

However, the CS for `max` is not complete, i.e. there are satisfiable or (worse) unsatisfiable constraint goals which are neither simplifiable nor callable. For example, the query `max(X,7,9)` results in `max(X,7,9),X=<9`, but it is not reduced to `X=9`. In practice, a CS is often not complete for efficiency reasons, For example, the implementation of CLP($\mathcal{R}$) delays non-linear arithmetic expressions [Ja*90], although there is a decision procedure for them. If the application requires it, we can always add SiRs to cover the incomplete cases or modify the call declaration to enable additional choices to trigger the necessary simplifications, while built-in constraint solvers cannot be as easily adopted. In our example, new SiRs of the form

```
max(X,Y,Z) ⇔ Y<Z | X=Z.
```

or an extended call declaration

```
callable max(X,Y,Z) if ground(X),ground(Y).

callable max(X,Y,Z) if ground(X),ground(Z).

callable max(X,Y,Z) if ground(Y),ground(Z).
```

will help.

Potentially, we can view any predicate as a definition of a constraint and add some SiRs for it. Seen this way, SiRs are lemmas that allow us to express the determinate information contained in a predicate. As a result, predicates and constraints are just alternate views on the same information. To see the power of such lemmas consider

```
append(X,[],L) ⇔ X=L,list(L).
```

A recursion on the list `X` in the usual definition of `append` is replaced by a simple unification `X=L` and type check `list(L)`.

With our approach, building a constraint solver amounts to writing a constraint definition and adding SiRs and a suitable call declaration to it. A CS written with SiRs can be easily extended by adding SiRs or relaxing guards, and it can be easily specialized by removing SiRs or strengthening guards. CS's are combined by merging their clauses. However, some care has to be taken if the CS's share constraints. One constraint definition has to be removed and we have to make sure that the resulting set of SiRs is still *canonical*, i.e. terminating and confluent.

In the next section, we introduce the syntax of constraint simplification rules, their declarative and operational semantics. Section 3 introduces our prototype implementation of Prolog with SiRs. Then we discuss related as well as current and future work.

2 Syntax and Semantics of SiRs

SiRs augment a given programming language, the *host language*, in order to define simplification of user-defined constraints. In the following we assume that the host language is a constraint logic programming language, however other languages (e.g. LISP or ML) can be augmented as well. In this context, Prolog can be seen as constraint logic programming language with equality as built-in constraint over the Herbrand Universe, CLP($\mathcal{H}$).

2.1 Syntax of SiRs

A program written in the host language augmented by SiRs is a finite set of clauses from the host language and from the language of SiRs. The clauses of the language of SiRs are call declarations and constraint simplification rules. A predicate definition in the host language extended by one or more call declarations is called a *constraint definition*. A constraint definition together with one or more SiR is called a *user-defined constraint*.

There are two kinds of SiRs. The syntax of a *replacement simplification rule* is

$A_1, \ldots A_i \Leftrightarrow\ C_1, \ldots C_j \mid B_1, \ldots B_k$

and of an *augmentation simplification rule* is

$A_1, \ldots A_i \Rightarrow\ C_1, \ldots C_j \mid B_1, \ldots B_k,$

where the

- head $A_1, \ldots A_i$ is a conjunction of atoms of user-defined constraints (called head atoms),
- guard $C_1, \ldots C_j$ is a conjunction of atoms (called guard atoms) which neither are, nor depend on, user-defined constraints,
- body $B_1, \ldots B_k$ is a conjunction of atoms (called body atoms).

Call declarations are meta-clauses that give conditions when a goal may be considered for calling. The syntax of a *call declaration* for a predicate p is

`callable` A `if` $C_1, \ldots C_j$,

where A is an atom of p and the $C_1, \ldots C_j$ is a guard, a conjunction of atoms which neither are, nor depend on, user-defined constraints. Prolog implementations supporting coroutining use various meta-logical predicates or declarations to specify when a goal has to delay (or suspend). There are *geler* (alias *freeze*) in Prolog II, *wait* declarations in MU-Prolog, *when* declarations in NU-Prolog and *delay* declarations in Sepia-Prolog [M*89]. Call declarations are more expressive than delay declarations. For example, instead of

```
callable max(X,Y,Z) if bound(X), bound(Y).
```

we can be more precise and state that we may call `max(X,Y,Z)` if the order between `X` and `Y` is known,

```
callable max(X,Y,Z) if X=<Y.
```

```
callable max(X,Y,Z) if X>=Y.
```

Furthermore, there is a crucial difference in operational semantics. A goal is eagerly activated as soon as its delay declaration is violated, but its activation is delayed as long as possible even if its call declaration is satisfied.

The syntax of guarded SiRs was chosen to exhibit the relationship with committed choice languages. Like in most of these languages, guards are kept simple, because they are checked many times. Therefore we do not allow user-defined constraints in the guard. As usual, variables in the guard that do not appear in the head of a SiR are considered to be existentially quantified. For the guards, the host language should at least provide built-in predicates or constraints to compare and type-check terms, e.g.

- `true` that always succeeds and `fail` that always fails,

- `bound` and `ground` about the binding of variables[1]
- `number` and `list` about types,
- `=:=` and `=` to check equality of arithmetic expressions and terms respectively,
- `<` and `@<` to provide an order on arithmetic expressions and terms respectively.

2.2 Declarative Semantics

An overview on semantics of logic programming languages with don't know nondeterminism can be found in [Ll87]. There are approaches to give declarative fixpoint semantics to languages with don't care nondeterminism as well, mostly by Levi and his colleagues, e.g. [FL88, Le88]. This is not an easy problem due to the nature of don't care nondeterminism [HA88, Nai89].

By contrast, declarative semantics for programs with SiRs is straightforward because correct Replacement SiRs just define simplifications which preserve meaning and correct Augmentation SiRs just add redundant constraints. In a logical sense, correct SiRs must be redundant when read as an implication from head to body. In other words, a correct SiR must be a logical consequence of the constraint definitions in the program if its guard is true. Note that so far the constraint definition was used to introduce choices, now it is also used as a constraint *specification* to prove correctness. More formally, define a constraint logic program with SiRs to be *completed* if each predicate and constraint definition is completed by Clarks completion [Ll87]. Given a completed constraint logic program P, let P_{Host} be P without the SiRs and call declarations. A replacement simplification rule in P of the form $A_1, \ldots A_i \Leftrightarrow C_1, \ldots C_j \mid B_1, \ldots B_k$ is *correct* with respect to P_{Host} if

$(P_{Host} \wedge C_1 \wedge \ldots C_j) \rightarrow (A_1 \wedge \ldots A_i \leftrightarrow B_1 \wedge \ldots B_k)$.

Analogously, an augmentation simplification rule $A_1, \ldots A_i \Rightarrow C_1, \ldots C_j \mid B_1, \ldots B_k$ in P is *correct* with respect to P_{Host} if

$(P_{Host} \wedge C_1 \wedge \ldots C_j) \rightarrow (A_1 \wedge \ldots A_i \rightarrow B_1 \wedge \ldots B_k)$.

Hence the declarative semantics of the underlying host language are inherited: The declarative semantics of a program with correct SiRs is just the declarative semantics of the program with the SiRs removed. To prove SiRs correct, we have to show that they are indeed logical consequences of the program.

In deterministic ALPS [Ma87] programs the theorems about declarative semantics of don't know nondeterministic logic programs [Ll87] only apply under a strong condition. The guards in the clauses of each predicate have to be mutually exclusive. SiRs can be seen as extending this result to any committed choice language whose clauses are logical consequences of constraint definitions. A quite elaborate soundness theorem with regard to declarative semantics of the Andorra Kernel Language (AKL) is proven in [Fra90]. To achieve the result, the allowed guards of AKL have to be restricted in several nontrivial ways to tame the infamous cut.

2.3 Operational Semantics

The *computation of a constraint program* is a (finite) sequence of *derivation steps* which rewrite the resolvent by adding or removing goals[2]. A *resolvent* is a conjunction of goals, which is considered as a multi-set (or bag) or store of goals[3]. A *goal* is an atom. The initial resolvent

[1] Note that `bound` and `ground` delay until they succeed, they never fail as opposed to `nonvar`.

[2] A more precise operational semantics using a transition system can be found in [Fru93].

[3] A generalization of a constraint store.

is called *query*. The final resolvent is called *(conditional) answer*. It is a conjunction of goals which are neither simplifiable by a SiR nor callable.

Predicates are used in a goal-driven manner to generate new substitutions and constraints when choosing a clause. These choices may lead to a combinatorial explosion in the number of backtrackings. On the other hand, constraints are employed in a data-driven manner. When enough information is available, the constraint simplifies in a determinate way and can eventually be tested for satisfiability. Hence we want to postpone execution of predicates as long as possible and rather do some constraint simplification. We are eager in simplification and lazy in choices. Only if no constraint simplification is possible a *callable* goal can be chosen for execution. This yields to a preference order on possible derivation steps which is reflected in the following *computation strategy*.

- Determinate Derivation Steps by Constraint Simplification
  - solve built-in constraints using the built-in CS
  - simplify user-defined constraints using SiRs
- Introduction of Choices by Predicate Calling
  - call a callable predicate
  - call a callable user-defined constraint using its constraint definition.

A sequence of derivation steps only involving constraint simplification is a single *simplification step*.

In don't know nondeterministic logic programming languages, a determinate derivation step can be performed if a goal succeeds with at most one clause. This idea seems to be first present in [Nai85] and in P-Prolog [YaAi86] and later in ALPS [Ma87], a class of flat committed choice languages with constraints. The Andorra Model of D.H.D. Warren and Guarded Rules of Smolka [Smo91] also give priority to determinate derivation steps, the latter author names this principle residuation. One difference to the Andorra Model is that residuation performs nondeterministic computation steps only on atoms whose predicate is explicitly declared as generating. The problem with these approaches is that they aim at determining at run-time whether at most one clause of a predicate can lead to success, which is undecidable in general. In our approach the determinism in a predicate is made explicit by SiRs, and the condition when a determinate simplification is possible is expressed concisely by the guards. SiRs and many other techniques for speeding up computations do not interfere, they are orthogonal to each other. This holds, for example, for residuation and generalised propagation [LPW92]. In our prototype we have introduced a declaration similar to the call declaration stating when a predicate is deterministic. Determinism is not detected at run-time, but declared[4] and thus the drawback of residuation mentioned before is avoided.

We now explain how user-defined constraints simplify via SiRs. Each user-defined constraint is associated with all SiRs in whose heads it occurs. Every time the constraint is activated, it checks itself the applicability of its associated SiRs. If a SiR has more than one head atom, the resolvent is searched for the other head atoms. Then the guards are evaluated, ideally concurrently. Either a guard succeeds, fails or delays. If the guard succeeds and if the SiR is a Replacement SiR, the matched head atoms in the resolvent are replaced by the body of the SiR. Because the matched head atoms are gone, all the SiRs associated with it are gone as well. If the SiR is an Augmentation SiR, the body of the SiR is added to the resolvent. If a guard fails, the associated SiR cannot fire. It is not necessary to reconsider it again. Any substitution performed or any built-in constraint that is added to the resolvent may cause activation of the constraint and wake up the delayed guard.

[4]The "generating" declaration proposed by Smolka has no conditional part and is therefore too weak for our purposes.

3 Reasoning about SiRs

It is useful to view SiRs as conditional rewrite systems to establish that they are *canonical*, i.e. terminating and confluent. On the other hand, when viewed as logical formulae, the *correctness* of SiRs can be established. We can prove the former property by adopting well-known techniques such as termination proofs and unfailing completion from (conditional and constrained) rewriting systems. We can prove the latter property by using techniques from automated theorem proving.

If we can prove a set of SiRs both canonical and correct we can be sure that the SiRs indeed implement a "well-behaved" constraint solver. As far as we know there is no other logic programming language that relies to such an extent on techniques developed for rewriting systems. An introduction to rewrite systems is [Kir89], to conditional rewriting systems [KR89, DO88]. An introduction to automated theorem proving is [Gal86]. For the relationship of automated theorem proving and logic programming we refer the reader to [WoMc91]. Also note that completion of rewrite systems can serve as a theorem proving procedure [Hs85, JSC91].

Correctness has been discussed in the previous section on declarative semantics.

3.1 Termination and Confluence

Termination is a highly desirable property which has been studied in many different contexts. In particular, termination proofs for sets of SiRs can benefit from work in rewriting systems [Der87] and logic programming [Plu90, VeDe91, Bez89, ApPe90]. If a set of SiRs is terminating, then there is no simplification step from any resolvent consisting of infinitely many derivation steps. Finding a suitable termination order may need user intervention. Termination for a class of SiRs is proved by giving an ordering on atoms showing that the body of a rule is always smaller than the head of the rule. Our experiments with the prototype implementation indicate that the expressive power of terminating SiRs is sufficient to model constraint simplification in a natural way. Such an ordering in addition introduces an intuitive notion of a "simpler" constraint, so that we also prove the intuition that constraints get indeed simplified. A constraint `C1` is simpler than a constraint `C2` if it is "simpler" to define `C1` with `C2` than vice versa. Note that when combining constraint solvers that share constraints, nonterminating simplification steps might arise even if each solver is terminating. E.g. one solver defines less-than in terms of greater-than and the other defines greater-than in terms of less-than.

As correct SiRs are logical consequences of the program, any result of a simplification step will have the same meaning, but it is not guaranteed that the result is syntactically the same. In particular, for a resolvent, unsatisfiability may be detected without making choices or not depending on what SiRs have been used for simplification.

A set of SiRs is *confluent* if each possible order of applications starting from any resolvent leads to the same resulting resolvent. A set of SiRs is *locally confluent* if any two resolvents resulting from one application of a SiR to the intial resolvent can be simplified into the same resolvent. It is well-known from rewrite systems that local confluence and termination imply confluence. Furthermore, in a confluent set of SiRs any resolvent has a *unique* normal form, provided it exists. This means that the answer to a query will always be the most simple one[1]. The notion of confluence is important for combining constraint solvers as well as concurrent applications of SiRs. Concurrent SiRs are not applied in a fixed order. As correct SiRs are logical consequences of the program, any result of a simplification step will have the same meaning, however it is not guaranteed anymore that the result is syntactically the same. In particular, a solver might be

[1] It can, however, contain redundant constraints and introduce new variables.

complete with one order of applications but incomplete with another one. Syntactically different constraint evaluations might also arise if combined solvers share constraints, depending on which solver comes first.

To show that a set of SiRs is locally confluent, we employ a variant of the well-known *completion*[2] procedure originally conceived by Knuth and Bendix [Kir89]. For some contrived examples, any completion procedure may not terminate. For each pair of SiRs whose head atoms overlap, so-called critical pairs taking their bodies are produced. If we cannot show that the resolvents in the critical pair are identical by simplifying them, we orient them into a new SiR that we add. Of course, adding a SiR implies computing new critical pairs. Orientation of the critical pair is based on the termination order and may fail. Such unorientable rules can be oriented using *unfailing completion*[Kir89, KR89, DO88] by adding an appropriate ordering constraint in the guard.

An unfailing completion procedure for SiRs is proposed in a forthcoming report by the same author.

4 The Prototype

A meta-interpreter for Prolog augmented with SiRs has been implemented on top of Sepia-Prolog [M*89] utilizing its delay-mechanism and built-in meta-predicates to create, inspect and manipulate delayed goals. The prototype includes a simple kind of incremental constraints, a preprocessor for SiRs, a tracing tool for user-defined constraints and variable bindings, and a simple partial evaluator based on simplifications. It is completely transparent with regard to the host language Sepia-Prolog. A wide range of constraint solvers have been implemented in the resulting language.

By "incremental constraints" we mean the possibility for the user to add additional constraints during computation. The preprocessor associates each user-defined constraint with the SiRs it can potentially match and partially evaluates their guards. Thus at run-time only single constraint goals need to be meta-interpreted if access to the resolvent is provided. The tracer shows which SiRs are firing and which choices are made or undone. It is based on an extension of the four-port box-model of the standard Prolog debugger. The variable tracer shows when and how a variable is bound and when the binding is undone due to backtracking. The partial evaluator takes a user program and simplifies each clause body, of both SiRs or definite clauses, with the help of the SiRs, but making only a limited number of choices (usually zero or one). There are some additional features not discussed in full in this paper, including run-time loop checking and a declaration for deterministic predicates to support residuation [Smo91]. A combination with generalised propagation [LPW92] is planned.

With the help of the prototype, we designed constraint solvers with SiRs for

- n-queens,
- inequalities,
- booleans,
- finite domains (a la CHIP),
- terminological reasoning [FrHa93],
- temporal reasoning [Fru93],
- real closed fields (a la CLP(R)),
- term manipulation.

[2] Do not confuse with Clarks completion of logic programs.

Typically it took only a few days to produce a reasonable prototype. In the n-queens problem, treating the `no_attack` predicate as a user-defined constraint has reduced the number of backtracks even more than reported with any of the approaches in [VH89], p. 123. However, the additional cost outweighs the savings in backtracking. The solver for inequalities is based on the one described in the introduction. The boolean solver, given the definition of a full-adder, is able to find out by constraint simplification only that adding a number to itself results in a number whose binary representation is shifted by one digit. Finite domains over equalities and inequalities were implemented as introduced in CHIP [VH89]. In terminological reasoning, the constraint simplifier can simplify and prove inconsistency of attributive concept definitions. We plan to extend this application to cover types and subsume finite domains. A constraint solver for temporal time point constraints [DMP91] was developed step by step starting from the solver for inequalities. We give a short presentation of this idea in the next section. A generic constraint solver for qualitative and quantitative temporal constraints over points and intervals based on path consistency is described in [Fru93]. With real closed fields we mean a CLP($\mathcal{R}$)-like solver that runs the examples that are distributed with the CLP($\mathcal{R}$) implementation [Ja*90]. The solver was developed by a straightforward extension of Gaussian elimination with inequalities. Recently we have connected our solver to one for nonlinear polynomials based on Groebner Bases [Mon92]. The term manipulation CS turns the built-in predicates `functor`, `arg` and `=..` into user-defined constraints. SiRs have also been used as a committed choice programming language on their own. Examples from [Sha89] as well as the basics of an Earley parser [Ea70] and a distributed shortest path algorithm have been implemented.

These constraint solvers back up our claim that it is possible to conveniently define constraint solvers with SiRs. This is because one can directly express the essence of constraint evaluation, the simplification of constraints, without worrying about implementation details. Control is predefined but can be customized with the help of high level call declarations.

Regarding speed, for the small problems we have tackled so far, our prototype is on average an order of magnitude slower than the built-in constraint solvers (if available) of CHIP or CLP($\mathcal{R}$). This does not come as a surprise, as the prototype is basically a meta-interpreter and lacks any specialized data structures (e.g. bit vectors) used in built-in constraint solvers. On some examples, we match the speed of the built-in solvers. Depth-first tree search as performed by Prolog has exponential complexity. Constraint programming cuts branches in the search tree. A conjunction of constraints is simplified with an algorithm that often has just polynomial complexity. In other words, once the problem is big enough, using constraints can pay off, even if the CS is slow.

A particular challenge of the implementation is matching of multiple head atoms. In the prototype, we have restricted the number of head atoms to two. For most applications, two heads sufficed. In spite of the restriction, any number of heads can still be matched at the expense of introducing auxiliary constraints and SiRs. In most SiRs, the head atoms are connected through a shared variable. This means that we only have to search for the second goal in the list of those goals which are delayed on the shared variable. The list of these delayed goals comes for free, as it is already maintained by the delay-mechanism of Sepia-Prolog. Of course, the overall complexity of goal search for two headed SiRs is still quadratic in a fraction of the size of the resolvent. If further speed-up is needed, once the CS has been established, proven correct and "tuned" as required, it can be reworked in a low-level language.

The host language has to provide coroutining. There is need to access the resolvent to get hold of the delayed goals (alternatively they could be passed in an additional argument for each predicate). To evaluate guards, there has to be a mechanism to evaluate it locally and to delay the execution of the subsequent body if the guard delays. In Sepia-Prolog, this can be done with a meta-call that returns all delayed goals, so the case the guard delays can be determined.

Another possibility would be an if-then-else that delays until the condition either succeeds or fails. Such a construct was present in a version of CHIP [A*90]. Last but not least, if a user-defined constraint has been tried to match its associated SiRs, but was not replaced, we have to redelay it. In Sepia-Prolog, this is achieved by a special delay declaration. The same effect could be achieved by using a dynamic construct like freeze.

5 An Example Solver for Temporal Time Point Constraints

Recently, as a result of collaborating with CHIC Esprit partners we have started to investigate temporal constraints. So far what we can conclude from this preliminary work is that SiRs enable a clear step by step development of constraint solvers of specific domains.

In order to define a constraint solver on temporal constraints over time points we exploited the natural relationship of these constraints with ordering constraints in general. Therefore, we started from the constraint solver for the less-than-or-equal constraint `'=<'` introduced in the introduction. We extended the inequality to the form `X+N=<Y`, where N is a given positive number, meaning that the distance in time (or space) of the two points `X` and `Y` is at least `N`.

```
callable XN =< Y if ground(XN),bound(Y).

XN=<Y :- call(XN =< Y, sepia).


X+N=<X ⇔ N=0.

X+N=<Y,X+M=<Y ⇔ NM is max(N,M) | X+NM=<Y.

X+N=<Y,Y+M=<X ⇔ N = 0, M = 0, X = Y.

X+N=<Y,Y+M=<Z ⇒ NM is N+M | X+NM=<Z.
```

In the call declaration the extension in syntax is reflected by requiring the first argument to be ground, such that `X+N` can be evaluated. The four SiRs are straightforward extensions of the previous ones. Some auxiliary arithmetic computations with `is` are added in the guards to compute the distances for the resulting inequalities in the body.

If we allow for negative `N` we can express maximal distances as well. The set of SiRs however will be non-terminating, as there is no termination order, because there is no bound on the minimal or maximal distances that could be computed anymore. The termination problem is solved by introducting a new constraint relation `'=<*'` which stands for *derived* inequalities as opposed to the initial ones using `'=<'`.

```
callable XN =< Y if ground(XN),bound(Y).

XN=<Y :- call(XN =< Y, sepia).


callable XN =<* Y if ground(XN),bound(Y).

XN=<*Y :- call(XN =< Y, sepia).


X+N=<Y ⇒ true | X+N=<*Y.


X+N=<*X ⇔ true | N=<0.

X+N=<*Y,Y+M=<*X ⇔ N=0,M=0 | X = Y.
```

```
X+N=<*Y,X+M=<*Y ⇔ NM is max(N,M) | X+NM=<*Y.

X+N=<*Y,Y+M=< Z ⇒ NM is N+M | X+NM=<*Z.
```

The derived inequality constraint of course has the same call declaration and predicate specification as the original inequality. The original SiRs are turned into SiRs for the derived inequality. However, there is one exception, which is the crucial detail causing termination. In the last SiR performing transitive closure, one relation must be not a derived but an original relation. This also elimates redundant inequalities that have been produced by the transitive closure before. To get the simplifications started, we have to give some initial derived relations. This is done by the first SiR which has been added and produces a derived inequality for each initial inequality. We also drop the conditions about inequality of the points that ensured that the most specific SiR is applied first, because that is are handled implicitly by our prototype implementation.

In temporal reasoning applications, usually both minimal and maximal distance of two time points are given. Hence it is a good idea to merge the two constraints `X+N=<Y,Y+M=<X` (`N` positive and `M` negative) into a single constraint `N=<Y-X=<(-M)` (by abuse of the relational notation), where `Y` is the *starting point* and `X` is the *end point* of the interval `Y-X`. This is exactly the notation used in [DMP91].

```
callable X =< Y =< Z if bound(X),ground(Y),bound(Y).

X =< Y =< Z:- call(X =< Y, sepia), call(Y =< Z, sepia).


callable X =<* Y =<* Z if bound(X),ground(Y),bound(Y).

X =<* Y =<* Z:- call(X =< Y, sepia), call(Y =< Z, sepia).


A=<X-Y=<B ⇒ A=<*X-Y=<*B.


A=<*X-X=<*B ⇔ A=<0=<B.

A=<*X-Y=<*B ⇔ A=0,B=0 | X = Y.

A=<*X-Y=<*B,C=<*X-Y=<*D ⇔ AC is max(A,C), BD is min(B,D) | AC=<*X-Y=<*BD.

A=<*X-Y=<*B,C=< Y-Z=< D ⇒ AC is A+C, BD is B+D | AC=<*X-Z=<*BD.

A=<*X-Y=<*B,C=< Z-Y=< D ⇒ AC is A-D, BD is B-C | AC=<*X-Z=<*BD.
```

Above, the SiRs have been extended correspondingly. The only interesting thing to note is that the last SiR about transitivity had to be split into two cases. The reason is that from `X+N=<Y, Y+M=<X` we only produced `N=<Y-X=<(-M)`, but not `M=<X-Y=<(-N)`, as it causes redundant computations for all other SiRs.

The above SiRs will produce derived inequality constraints for every pair of time points (provided they are connected). Again this means redundant information and hence redundant computation, as we can compute all relations when knowing the distances from one given reference point to all other time points. We will specify the reference point `X` with a dummy constraint `start(X)`. For this optimization only the first SiR has to be restricted from

```
A=<X-Y=<B ⇒ A=<*X-Y=<*B.
```

to

```
A=<X-Y=<B,start(X) ⇒ A=<*X-Y=<*B.
```

The resulting set of SiRs defines (and implements) a specialized constraint solver for temporal constraints on time points. The correctness of the solver can be shown and its behaviour has been tailored to temporal constraints starting from inequality constraints. Thus SiRs can support the prototyping of "built-in" constraint solvers. Further optimisations are possible, for example using a dynamic shortest-path algorithm. If further speed-up is needed, once the prototype has been established and "tuned" as required, it can be reworked in a low-level language.

6 Related Work

6.1 Constraint Logic Programming Languages

Current constraint logic programming languages [VH91, Coh90, Fru90] are not extensible, they do not allow for user-defined constraints. The exception to the rule is the constraint logic programming CHIP [VH89]. The general technique of *propagation* is employed over finite domains. The idea is to prune large search trees by enforcing local consistency of built-in and user-defined constraints. There is work at ECRC on extending propagation over finite domains to arbitrary constraint domains [LPW92], and on compiling propagation into demons [Kue91]. These techniques are orthogonal to our approach and thus can be integrated. *Demons* [A*90] are essentially single-headed Replacement SiRs without guards. However, demons must define a constraint completely, no associated constraint definitions are allowed. One version of CHIP also included *forward rules* [Gr89], which correspond to SiRs without guards. [Gr89] also gives a detailed account of the semantics of forward rules and therefore SiRs without guards. In this sense, SiRs can be seen as an extension of the work on demons and forward rules in CHIP. In practice, demons and forward rules have been proven useful in CHIP applications in the boolean domain for circuit design and verification [Si91]. Their potential to define constraint solvers in general was not realised, maybe because of their limitations. To the best of our knowledge, the notion of Augmentation rules is not present in any other logic programming language than CHIP and SiRs.

6.2 Combined and Extended Languages

In the following we relate our approach to other work on combining deterministic and nondeterministic computations into one logic programming language.

Amalgamating pure Prolog with single headed Replacement SiRs only results in a language of the family cc$(\downarrow, \rightarrow, \Rightarrow)$[1] of the cc framework proposed by Saraswat [Sar89, Sar90]. A close study of [Sar89] reveals that he proposes a special *Tell* operation called "inform" that could be used to simulate Augmentation SiRs. SiRs naturally fit the ask-and-tell interpretation of constraint logic programming introduced by Saraswat and applied by [VH91]. The resolvent is viewed as constraint store for user-defined constraints. They are matched by the heads of SiRs and the guards ask if certain constraints hold in the built-in constraint store and on the arguments of the matched user-defined constraints.

Guarded Rules [Smo91] correspond exactly to single headed Replacement SiRs. Like correct SiRs, *admissible* guarded rules are logical consequences of a program to be amalgamated. However, Smolka does not consider predicates with associated Guarded Rules as definitions for user-defined constraints. There are only built-in constraints. Interestingly, Smolka defines the built-

[1] $\downarrow$ means *Ask* in addition *Tell* is supported, $\rightarrow$ is the commit operator for don't care nondeterminism used and $\Rightarrow$ is the commit operator for don't know nondeterminism able to describe pure Prolog.

in constraint system as a terminating and determinate reduction system. Hence it could be implemented by Replacement SiRs. [Smo91] can be read as an excellent introduction to some of the basic ideas also underlying our approach.

The Andorra Model of D.H.D. Warren for parallel computation has inspired a rapid development of numerous languages and language schemes. The Andorra Kernel Language (AKL) [JaHa91] is a guarded language with built-in constraints based on an instance of the Kernel Andorra Prolog control framework [HaJa90]. AKL combines don't care nondeterminism and don't know nondeterministism with the help of different guard operators. There are three kinds of guard operators, namely cut, commit and wait. A language amalgamated with SiRs inherits the the commit operator of the SiRs as well as the guard operators of the host language (e.g. cut in the case of Prolog). Although single-headed Replacement SiRs can be written in AKL using the commit operator, it is not possible to add a constraint definition for the user-defined constraint, as AKL restricts all clauses for a predicate to have the same guard operator.

We think that AKL might be a good implementation language for SiRs and a good host language, because AKL already combines a variant of Prolog with a committed choice language. Like most logic programming languages, AKL itself does not support two of the essential features for defining simplification of user-defined constraints: Augmentation rules and multiple head atoms.

6.3 Multiple Head Atoms

According to [Coh88] at the very beginning of the development of Prolog in the early 70's by Colmerauer and Kowalski, experiments were performed with clauses having multiple head atoms. More recently, clauses with multiple head atoms were proposed to model parallelism and distributed processing, e.g. [Br90, AnPa91], or objects [Con88, AnPa90]. The similarity of the object oriented approaches with SiRs is merely syntactical. Rules about objects cannot be regarded as specifying constraint simplification. Object rules are supposed to model the change of objects, while SiRs model equivalence and implication of constraints. Unlike SiRs, object rules do not support both kinds of nondeterminism.

In committed choice languages, multiple head atoms have been considered only rarely. In his thesis, Saraswat remarks on multiple head atoms that "the notion seems to be very powerful" and that "extensive further investigations seems warranted" ([Sar89], p. 314). He motivates so-called *joint reductions* of multiple atoms as analogous to production rules of expert system languages like OPS5. The examples given suggest the use of joint reductions to model objects in a spirit similar to what is worked out in [AnPa90].

Multi-headed Replacement SiRs are sufficient to simulate the parallel machine for multiset transformation proposed in [BCL88]. This machine is based on the chemical reaction metaphor as means to describe highly parallel computations for a wide spectrum of applications. The proposed implementation on a vector architecture may also be useful for implementing SiRs. Following [BCL88], we implemented the sieve of Eratosthenes to compute primes simply as:

```
primes(1) ⇔ true.

primes(N) ⇔ N>1  |  M is N-1, prime(N),primes(M).

prime(I),prime(J) ⇔ 0 is J mod I  |  prime(I). % J is a multiple of I
```

The conditional answer to the query `primes(`n`)` will be a conjunction of *prime(*p_i*)* where each p_i is a prime ($2 \leq p_i \leq n$).

7 Conclusions

We proposed *constraint simplification rules* (SiRs) to define constraint solvers in logic programming languages. SiRs are multi-headed guarded clauses. By amalgamating a logic programming language with SiRs, a flexible, extensible constraint logic programming language results. Logic programs extended by correct SiRs have a straightforward declarative semantics as SiRs are logically redundant. In this way, a logical reconstruction for constraint solving in logic programming is achieved. As opposed to built-in constraint solvers written in low-level languages, a CS implemented by SiRs can be proven correct with regard to a constraint definition. Critics might argue that real-life constraint solvers defined by SiRs are hard to write and hard to prove correct. However, constraint solvers in a low-level language are harder to write and much harder to prove correct. Although intended as a language for constraint simplification, SiRs could also serve as a powerful programming language on their own.

Completion of SiRs and implementing various constraint solvers and meta-constraints (that take other constraints as arguments) are the topics of current research.

SiRs support rapid prototyping of built-in constraint solvers by providing executable specifications (if not implementations). They support specialization, modification and combination of constraint solvers. Our approach merges the advantages of constraints (simplification via SiRs) and predicates (choices via definite clauses). The result is a tight integration of the logic programming component and user-defined constraint solvers.

We believe that our approach has the potential to provide a comprehensive framework for constraints in logic programming, because SiRs will make it possible

- to add constraint solvers for any required domain of computation. Thus constraint solvers can be specially built for particular applications.
- to generate constraint solvers semi-automatically from constraint definitions.
- to enable debugging of CLPs.
- to integrate closely the logic program and the constraint solver, enabling reasoning about constraint logic programs.

Acknowledgements

Thanks to Alex, Jesper, Mark, Thierry and Volker, my colleagues at ECRC, who discussed these ideas with me, as well as G. Smolka. F. Rossi, and anonymous referees, who commented on this paper.

Bibliography

[A*90] A. Aggoun et al, CHIP Compiler Version 2 Reference Manual, and CHIP Interpreter Version 2.1 Reference Manual, ECRC, Munich, Germany, May 1990.

[AnPa90] Andreoli J.-M. and Pareschi R., Linear Objects: Logical Processes with Built-In Inheritance, Seventh Intl Conf on Logic Programming MIT Press 1990, pp. 495-510.

[AnPa91] Andreoli J.-M. and Pareschi R., Communication as Fair Distribution of Knowledge, Proceedings of OOPSLA '91.

[ApPe90] K. R. Apt and D. Pedreschi, Studies in Pure Prolog: Termination, ESPRIT Computational Logic Symposium, Springer 1990, pp. 150-176.

[BCL88] Banatre J.-P., Coutant A. and Le Metayer D., A Parallel Machine for Multiset Transformation and its Programming Style, Future Generation Computer Systems 4:133-144, 1988.

[Bez89] M. Bezem, Characterizing Termination of Logic Programs with Level Mappings, North American Conference on Logic Programming, MIT Press 1989, pp. 69-80.

[Br90] Brogi A., AND-Parallelism without Shared Variables, Proc of the Seventh Intl Conf on Logic Programming MIT Press 1990, pp. 306-321.

[CAL88] Aiba A. et al, Constraint Logic Programming Language CAL, Int Conf on Fifth Generation Computer Systems, 1988, Ohmsha Publishers, Tokyo, pp. 263-276.

[Coh88] J. Cohen, A View of the Origins and Development of Prolog, CACM 31(1):26-36, Jan. 1988.

[Coh90] J. Cohen, Constraint Logic Programming Languages, CACM 33(7):52-68, July 1990.

[Con88] Conery J. S., Logical Objects, Proc of the Fifth Intl Conf and Symp on Logic Programming MIT Press 1988, pp. 420-434.

[DO88] Dershowitz N. and Okada M., Conditional Equational Programming and the Theory of Conditional Term Rewriting, Int Conf on Fifth Generation Computer Systems, Ohmsha 1988, Tokyo, pp. 337-346

[Der87] N. Dershowitz, Termination of Rewriting, Journal of Symbolic Computation, 3(1+2):69-116, 1987.

[DMP91] R. Dechter, I. Meiri and J. Pearl, Temporal Constraint Networks, Journal of Artificial Intelligence 49:61-95, 1991.

[Ea70] J. Earley, An Efficient Context-Free Parsing Algorithm, CACM, 13(2), 1970.

[FL88] Falaschi M. and Levi G., Finite Failure and Partial Computations in Concurrent Logic Languages, Int Conf on Fifth Generation Computer Systems, Ohmsha 1988, Tokyo, pp. 364-381.

[Fra90] T. Franzen, Formal Aspects of Kernel Andorra: I, SICS Research Report R90008, May 1990, Swedish Institute of Computer Science, Kista, Sweden.

[Fru90] Frühwirth T., Constraint Logic Programming - An Overview, Technical Report E181-2, Christian Doppler Laboratory For Expert Systems, Vienna Austria, August 1990.

[Fru91] Frühwirth T., Constraint Simplification Rules, Internal Report LP-63, ECRC Munich, Germany, October 1991.

[Fru93] Frühwirth T., Temporal Reasoning with Constraint Handling Rules, Technical Report Core-93-8, ECRC Munich, Germany, January 1993.

[FrHa93] Frühwirth T. and Hanschke P., Terminological Reasoning with Constraint Handling Rules, Technical Report, in preparation, ECRC Munich, Germany, January 1993.

[Gal86] J. H. Gallier, Logic for Computer Science: Foundations of Automated Theorem Proving, Harper and Row, New York, 1986.

[Gr89] T. Graf, Raisonnement sur les contraintes en programmation en logique, Ph.D. Thesis, Version of June 1989 Universite de Nice, France, September 1989 (in French).

[HA88] Hewitt C. and Agha G., Guarded Horn Clause Languages: Are they Deductive and Logical?, Int Conf on Fifth Generation Computer Systems, Ohmsha 1988, Tokyo, pp. 650-657.

[HaJa90] Haridi S. and Janson S., Kernel Andorra Prolog and its Computation Model, Seventh Int Conference on Logic Programming, MIT Press 1990, pp. 31-46.

[Hs85] Hsiang J., Refutational Theorem Proving Using Term-Rewriting Systems, Artificial Intelligence 25(3), Elsevier, March 1985, pp. 255-300.

[JSC91] Special Issue on Rewriting Techniques in Theorem Proving, Bachmair L. and Hsiang J. (eds), Journal of Symbolic Computation, 11(1-2), Academic Press, Jan-Feb 1991.

[Ja*90] J. Jaffar et al The CLP($\mathcal{R}$) Language and System, Research Report RC 16292, November 1990, IBM T.J. Watson Research Center.

[JaHa91] S. Janson and S. Haradi, Programming Paradigms of the Andorra Kernel Language, Draft of March 13, 1991, accepted at ILPS 91 in San Diego, Swedish Institute of Computer Science, Kista, Sweden.

[JaLa87] J. Jaffar and J.-L. Lassez, Constraint Logic Programming, ACM 14th POPL 87, Munich, Germany, January 1987, pp. 111-119.

[KR89] Kaplan S. and Remy J.-L., Completion Algorithms for Conditional Rewriting Systems, Chapter 5 in Resolution of Equations in Algebraic Structures, Volume 2 Rewriting Techniques, Ait-Kaci H. and Nivat M. (eds), Academic Press 1989.

[Kir89] C. Kirchner and H. Kirchner, Rewriting: Theory and Applications, Working paper for a D.E.A. lecture at the University of Nancy I, France, 1989.

[Kue91] Küchenhoff V., Compiling Constraint Reduction, Technical Report Draft, ECRC Munich, Germany, September 1991.

[Le88] Levi G., Models, Unfolding Rules and Fixpoint Semantics, Invited Talk, Fifth Intl Conf and Symp on Logic Programming MIT Press 1988, pp. 1649-1665.

[Ll87] Lloyd J. W., Foundations of Logic Programming, 2nd ed., Springer, New York, 1987.

[LPW92] Le Provost T. and Wallace M., Domain Independent Propagation, International Conference on Fifth Generation Computer Systems 1992, Tokyo, Japan, June 1992, p. 1004-1012.

[M*89] Micha Meier et al., SEPIA - An Extendible Prolog System, 11th World Computer Congress IFIP'89, San Francisco, USA, August 1989.

[Ma87] Maher M. J., Logic Semantics for a Class of Committed Choice Programs, Proc of the Fourth Intl Conf on Logic Programming MIT Press 1987, pp. 858-876.

[Mon92] Monfroy E., Non-linear Constraints: A Language and a Solver, Technical Report ECRC-92, ECRC, Munich, Germany, 1992, to appear.

[Nai85] Naish L., Prolog control rules, Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California, September 1985, pp. 720-722.

[Nai89] Naish L., Proving Properties of Committed Choice Logic Programs, Journal of Logic Programming, 7(1), July 1989, pp. 63-84.

[Plu90] L. Pluemer, Termination Proofs for Logic Programs based on Predicate Inequalities, Seventh Int Conference on Logic Programming, MIT Press 1990, pp. 634-648.

[Sar89] V. A. Saraswat, Concurrent Constraint Programming Languages, Ph.D. Dissertation, Carnegie Mellon Univ., Draft of Jan. 1989.

[Sar90] V. A. Saraswat, Concurrent Constraint Programming, ACM Seventeenth Symp on Principles of Programming Languages, POPL 1990, pp. 232-245.

[Sha89] Shapiro E., The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.

[Si91] H. Simonis, Constraint Logic Programming as a Digital Circuit Design Tool, Ph.D. Thesis, Draft Version, February 1991.

[Smo91] Smolka G., Residuation and Guarded Rules for Constraint Logic Programming, Digital Equipment Paris Research Laboratory Research Report, France, June 1991.

[VE88] Van Emden M. H., Conditional Answers for Polymorphic Type Inference, Proc of the Fifth Intl Conf and Symp on Logic Programming MIT Press 1988, pp. 590-603.

[VH89] P. Van Hentenryck, Constraint Satisfaction in Logic Programming, MIT Press, Cambridge, Massachusetts, 1989.

[VH91] P. Van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, 6(3), pp. 151-194, September 1991.

[Va86] Vasey P., Qualified Answers and their Application to Transformation, Proc of the Third Intl Conf on Logic Programming 1986, pp. 425-432.

[VeDe91] K. Verschaetse and D. De Schreye, Deriving Termination Proofs for Logic Programs using Abstract Procedures, 8th Int Conf on Logic Programming, MIT Press 1991, pp. 301-315.

[WoMc91] Wos L. and McCune W., Automated Theorem Proving and Logic Programming: A Natural Symbiosis, Journal of Logic Programming, 11(1), July 1991, pp. 1-53.

[YaAi86] Yang R. and Aiso H., A Parallel Logic Language Based on the Exclusive Relation, Third Int Conference on Logic Programming, MIT Press, 1986.

Other Reports Available from ECRC

[ECRC–TR–LP-60] Mireille Ducasse and Anna-Maria Emde. *Opium 3.1 - User Manual A High-level Debugging Environment for Prolog.* 1991.

[ECRC–TR–LP-61] E. Yardeni, T. Frühwirth, and E. Shapiro. *Polymorphically Typed Logic Programs.* 1991.

[ECRC–TR–DPS-81] U. Baron, S. Bescos, and S. Delgado. *The ElipSys Logic Programming Language.* 17. 01. 1991.

[ECRC–TR–DPS-82] Sergio Delgado, Michel Dorochevsky, and Kees Schuerman. *A Shared Environment Parallel Logic Programming System On Distributed Memory Architectures.* 18. 01. 1991.

[ECRC–TR–DPS-83] Andre Veron, Jiyang Xu, and Kees Schuerman. *Virtual Memory Support for OR-Parallel Logic Programming Systems.* 05. 03. 1991.

[ECRC–TR–DPS-85] Michel Dorochevsky. *Garbage Collection in the OR-Parallel Logic Programming.* 15. 03. 1991.

[ECRC–TR–DPS-100] Alan Sexton. *KCM Kernel Implementation Report.* 22. 05. 1991.

[ECRC–TR–DPS-103] Michel Dorochevsky. *Key Features of a Prolog Module System.* 08. 03. 1991.

[ECRC–TR–DPS-104] Michel Dorochevsky, Kees Schuerman, and Andre Veron. *ElipSys: An Integrated Platform for Building Large Decision Support Systems.* 29. 01. 1991.

[ECRC–TR–DPS-105] Jiyang Xu and Andre Veron. *Types and Constraints in the Parallel Logic Programming System ElipSys.* 15. 03. 1991.

[ECRC–TR–DPS-107] Olivier Thibault. *Design and Evaluation of a Symbolic Processor.* 13. 06. 1991.

[ECRC–TR–DPS-112] Michel Dorochevsky, Jacques Noyé, and Olivier Thibault. *Has Dedicated Hardware for Prolog a Future ?* 14. 09. 1991.

[ECRC–91–1] Norbert Eisinger and Hans Jürgen Ohlbach. *Deduction Systems Based on Resolution.* 29. 10. 1991.

[ECRC–91–2] Michel Kuntz. *The Gist of GIUKU: Graphical Interactive Intelligent Utilities for Knowledgeable Users of Data Base Systems.* 4. 11. 1991.

[ECRC–91–3] Michel Kuntz. *An Introduction to GIUKU: Graphical Interactive Intelligent Utilities for Knowledgeable Users of Data Base Systems.* 4. 11. 1991.

[ECRC–91–4] Michel Kuntz. *Enhanced Graphical Browsing Techniques for Collections of Structured Data.* 4. 11. 1991.

[ECRC–91–5] Michel Kuntz. *A Graphical Syntax Facility for Knowledge Base Languages.* 4. 11. 1991.

[ECRC–91–6] Michel Kuntz. *A Versatile Browser-Editor for NF2 Relations.* 4. 11. 1991.

[ECRC–91–7] Norbert Eisinger, Nabiel Elshiewy, and Remo Pareschi. *Distributed Artificial Intelligence - An Overview.* 4. 11. 1991.

[ECRC–91–8] Norbert Eisinger. *An Approach to Multi-Agent Problem-Solving.* 11. 11. 1991.

[ECRC–91–9] Klaus H. Ahlers, Michael Fendt, Marc Herrmann, Isabelle Hounieu, and Philippe Marchal. *TUBE Implementor's Manual.* 21. 11. 1991.

[ECRC–91–10] Klaus H. Ahlers, Michael Fendt, Marc Herrmann, Isabelle Hounieu, and Philippe Marchal. *TUBE Programmer's Manual.* 21. 11. 1991.

[ECRC–91–11] Michael Dahmen. *A Debugger for Constraints in Prolog.* 26. 11. 1991.

[ECRC–91–12] Jean-Marc Andreoli and Remo Pareschi. *Communication as Fair Distribution of Knowledge.* 26. 11. 1991.

[ECRC–91–13] Jean-Marc Andreoli, Remo Pareschi, and Marc Bourgois. *Dynamic Programming as Multiagent Programming.* 26. 11. 1991.

[ECRC–91–14] Volker Küchenhoff. *On the Efficient Computation of the Difference Between Consecutive Database States.* 5. 12. 1991.

[ECRC–91–15] Sylvie Bescos and Michael Ratcliffe. *Secondary Structure Prediction of rRNA Molecules Using ElipSys.* 16. 12. 1991.

[ECRC–91–16] Michael Dahmen. *Abstract Debugging of Coroutines and Constraints in Prolog.* 30. 12. 1991.

[ECRC–92–1] Thierry Le Provost and Mark Wallace. *Constraint Satisfaction Over the CLP Scheme.* 30. 1. 1992.

[ECRC–92–2] Gérard Comyn, M. Jarke, and Suryanarayana M. Sripada. *Proceedings of the 1st Compulog Net meeting on Knowledge Bases (CNKBS'92).* 30. 1. 1992.

[ECRC–92–3] Jesper Larsson Traeff and Steven David Prestwich. *Meta-programming for reordering Literals in Deductive Databases.* 30. 1. 1992.

[ECRC–92–4] Beat Wüthrich. *Update Realizations Drawn from Knowledge Base Schemas and Executed by Dialog.* 4. 2. 1992.

[ECRC–92–5] Lone Leth. *A New Direction in Functions as Processes.* 25. 2. 1992.

[ECRC–92–6] Steven David Prestwich. *The PADDY Partial Deduction System.* 23. 3. 1992.

[ECRC–92–7] Andrei Voronkov. *Extracting Higher Order Functions from First Order Proofs.* 23. 3. 1992.

[ECRC–92–8] Andrei Voronkov. *On Computability by Logic Programs.* 23. 3. 1992.

[ECRC–92–9] Beat Wüthrich. *Towards Probabilistic Knowledge Bases.* 02. 4. 1992.

[ECRC–92–10] Petra Bayer. *Update Propagation for Integrity Checking, Materialized View Maintenance and Production Rule Triggering.* 08. 4. 1992.

[ECRC–92–11] Mireille Ducassé. *Abstract views of Prolog executions in Opium.* 15. 4. 1992.

[ECRC–92–12] Alexandre Lefebvre. *Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases.* 30. 4. 1992.

[ECRC–92–13] Udo W. Lipeck and Rainer Manthey (Hrsg.). *Kurzfassungen des 4. GI-Workshops "Grundlagen von Datenbanken", Barsinghausen, 9.-12.6.1992.* 12. 05. 1992.

[ECRC–92–14] Lone Leth and Bent Thomsen. *Some Facile Chemistry.* 26. 05. 1992.

[ECRC–92–15] Jacques Noyé (Ed.). *Proceedings of the International KCM User Group Meeting,Munich, 7 and 8 October 1991.* 03. 06. 1992.

[ECRC–92–16] Frederick Knabe. *A Distributed Protocol for Channel-Based Communication with Choice.* 10. 06. 1992.

[ECRC–92-17] Benoit Baurens, Petra Bayer, Luis Hermosilla, and Andrea Sikeler. *Publication Management: A Requirements Analysis.* 03. 07. 1992.

[ECRC–92-18] Thom Frühwirth. *Constraint Simplification Rules.* 28. 07. 1992.

[ECRC–92-19] Mark Wallace. *Compiling Integrity Checking into Update Procedures.* 29. 07. 1992.

[ECRC–92-20] Petra Bayer. *Data and Knowledge for Medical Applications: A Case Study.* 30. 07. 1992.

[ECRC–92-21] Michel Dorochevsky and André Véron. *Binding Techniques and Garbage Collection for OR-Parallel CLP Systems.* 11. 08. 1992.

[ECRC–92-22] Shan-Wen Yan. *Efficiently Estimating Relative Grain Size for Logic Programs on Basis of Abstract Interpretation.* 25. 08. 1992.

[ECRC–92-23] Jean-Marc Andreoli, Paolo Ciancarini, and Remo Pareschi. *Interaction Abstract Machines.* 25. 08. 1992.

[ECRC–92-24] Jean-Marc Andreoli and Remo Pareschi. *Associative Communication and its Optimization via Abstract Interpretation.* 25. 08. 1992.

[ECRC–92-25] Jean-Marc Andreoli, Lone Leth, Remo Pareschi, and Bent Thomsen. *On the Chemistry of Broadcasting.* 25. 08. 1992.

[ECRC–92-26] Marc Bourgois, Jean-Marc Andreoli, and Remo Pareschi. *Extending Objects with Rules, Composition and Concurrency : the LO Experience.* 25. 08. 1992.

[ECRC–92-27] Benoit Dageville and Kam-Fai Wong. *SIM: A C-based SIMulation Package.* 28. 09. 1992.

[ECRC–92-28] Beat Wüthrich. *On the Efficient Distribution-free Learning of Rule Uncertainties and their Integration into Probabilistic Knowledge Bases.* 29. 09. 1992.

[ECRC–92-29] Andrei Voronkov. *Logic Programming with Bounded Quantifiers.* 29. 09. 1992.

[ECRC–92-30] Eric Monfroy. *Gröbner Bases: Strategies and Applications.* 30. 09. 1992.

[ECRC–92-31] Eric Monfroy. *Specification of Geometrical Constraints.* 30. 09. 1992.

[ECRC–92-32] Bent Thomsen, Lone Leth, and Alessandro Giacalone. *Some Issues in the Semantics of Facile Distributed Programming.* 22. 10. 1992.

[ECRC–92-33] Mireille Ducassé. *An Extendable Trace Analyser to Support Automated Debugging.* 04. 12. 1992.

[ECRC–92-34] Jorge Bocca and Luis Hermosilla. *A Preliminary Study of the Performance of MegaLog.* 20. 12. 1992.

[ECRC–93–1] Benoit Dageville and Kam-Fai Wong. *Supporting Thousands of Threads Using a Hybrid Stack Sharing Scheme.* 18. 01. 1993.

[ECRC–93–2] Steven Prestwich. *ElipSys Programming Tutorial.* 18. 01. 1993.

[ECRC–93–3] Beat Wüthrich. *Learning Probabilistic Rules.* 28. 01. 1993.

[ECRC–93–4] Eric Monfroy. *A Survey of Non-Linear Solvers.* 02. 02. 1993.

[ECRC–93–5] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. *Constraint Logic Programming - An Informal Introduction.* 02. 02. 1993.