Constraint Logic Programming: A Survey

Joxan Jaffar and Michael J. Maher

▷ Constraint Logic Programming (CLP) is a merger of two declarative paradigms: constraint solving and logic programming. Though a relatively new field, CLP has progressed in several quite different directions. In particular, the early fundamental concepts have been adapted to better serve in different areas of applications. In this survey of CLP, a primary goal is to give a systematic description of the major trends in terms of common fundamental concepts. The three main parts cover the theory, implementation issues and programming for applications. ◁

1. Introduction

Constraint Logic Programming (CLP) began as a natural merger of two declarative paradigms: constraint solving and logic programming. This combination helps make CLP programs both expressive and flexible, and in some cases, more efficient than other kinds of programs. Though a relatively new field, CLP has progressed in several and quite different directions. In particular, the early fundamental concepts have been adapted to better serve in different areas of applications. In this survey of CLP, a primary goal is to give a systematic description of the major trends in terms of common fundamental concepts.

Consider first an example program in order to identify some crucial CLP concepts. The program below defines the relation $sumto(n, 1+2+\cdots+n)$ for natural numbers n.

```
sumto(0, 0).
sumto(N, S) :- N >= 1, N <= S, sumto(N - 1, S - N).
```

The query `S <= 3, sumto(N, S)` gives rise to three answers $(N = 0, S = 0)$,

Address correspondence to Joxan Jaffar and Michael Maher, IBM Thomas J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, USA. Email: {joxan,mjm}@watson.ibm.com

FINAL DRAFT: Comments, especially about errors, are solicited.

$(N = 1, S = 1)$, and $(N = 2, S = 3)$, and terminates. The computation sequence of states for the third answer, for example, is

$$S \leq 3, sumto(N, S).$$

$$S \leq 3, N = N_1, S = S_1, N_1 \geq 1, N_1 \leq S_1,$$
$$sumto(N_1 - 1, S_1 - N_1).$$

$$S \leq 3, N = N_1, S = S_1, N_1 \geq 1, N_1 \leq S_1,$$
$$N_1 - 1 = N_2, S_1 - N_1 = S_2, N_2 \geq 1, N_2 \leq S_2,$$
$$sumto(N_2 - 1, S_2 - N_2).$$

$$S \leq 3, N = N_1, S = S_1, N_1 \geq 1, N_1 \leq S_1,$$
$$N_1 - 1 = N_2, S_1 - N_1 = S_2, N_2 \geq 1, N_2 \leq S_2,$$
$$N_2 - 1 = 0, S_2 - N_2 = 0.$$

The constraints in the final state imply the answer $N = 2, S = 3$. Termination is reasoned as follows. Any infinite computation must use only the second program rule for state transitions. This means that its first three states must be as shown above, and its fourth state must be

$$S \leq 3, N = N_1, S = S_1, N_1 \geq 2, N_1 \leq S_1,$$
$$N_1 - 1 = N_2, S_1 - N_1 = S_2, N_2 \geq 1, N_2 \leq S_2,$$
$$N_2 - 1 = N_3, S_2 - N_2 = S_3, N_3 \geq 1, N_3 \leq S_3,$$
$$sumto(...)$$

We note now that this contains an unsatisfiable set of constraints, and in CLP, no further reductions are allowed.

This example shows the following key features in CLP:

- Constraints are used to specify the query as well as the answers.
- During execution, new variables and constraints are created.
- The collection of constraints in every state is tested as a whole for satisfiability before execution proceeds further.

In summary, constraints are: used for input/output, dynamically generated, and globally tested in order to control execution.

1.1. Constraint Languages

Considerable work on constraint programming languages preceded logic programming and constraint logic programming. We now briefly survey some important works, with a view toward the following features. Are constraints used for input/output? Can new variables and/or constraints be dynamically generated? Are constraints used for control? What is the constraint solving algorithm, and to what extent is it complete? What follows is adapted from the survey in [188].

SKETCHPAD [246] was perhaps the earliest work that one could classify as a constraint language. It was in fact an interactive drawing system, allowing the user to build geometric objects from language primitives and certain constraints. The constraints are static, and were solved by local propagation and relaxation techniques. (See chapter 2 in [165] for an introduction to these and related techniques.) Subsequent related work was THINGLAB [32] whose language took an object-oriented flavor. While local propagation and relaxation were also used to deal with the essentially static constraints, the system considered constraint solving in two different phases. When a graphical object is manipulated, a plan was generated for quickly re-solving the appropriate constraints for changed part of the object. This plan was then repeatedly executed while the manipulation continued. Works following the THINGLAB tradition included the Filters Project [81] and Animus [80]. Another graphical system, this one focusing on geometrical layout, was JUNO [196]. The constraints were constructed, as in THINGLAB, by text or graphical primitives, and the geometric object could be manipulated. A difference from the abovementioned works is that constraint solving was performed numerically using a Newton-Raphson solver.

Another collection of early works arose from MIT, motivated by applications in electrical circuit analysis and synthesis, gave rise to languages for general problem solving. In the CONSTRAINTS language [240], variables and constraints are static, and constraint solving was limited to using local propagation. An extension of this work [241] provided a more sophisticated environment for constraint programming, including explanation facilities. Some other related systems, EL/ARS [238] and SYN [142], used the constraint solver MACSYMA [186] to avoid the restrictions of local propagation. It was noted at this period [241] that there was a conceptual correspondence between the constraint techniques and logic programming.

The REF-ARF system [89] was also designed for problem solving. One component, REF, was essentially a procedural language, but with nondeterminism because of constraints used in conditional statements. The constraints are static. They are in fact linear integer constraints, and all variables are bounded above and below. The constraint solver ARF used backtracking.

The Bertrand system [165] was designed as a meta-language for the building of constraint solvers. It is itself a constraint language, based on term rewriting. Constraints are dynamic here, and are used in control. All constructs of the language are based on augmented rewrite rules, and the programmer adds rules for the specific constraint solving algorithm to be implemented.

Post-CLP, there have been a number of works which are able to deal with dynamic constraints. The language 2LP [170] is described to be a CLP language with a C-like syntax for representing and solving combinatorial problems. Obtaining parallel execution is one of the main objectives of this work. The commercial language CHARME, also based on a procedural framework, arose from the work on CHIP (by essentially omitting the logic programming part of CHIP). The subsequent work ILOG, which is also commercial, is a library of constraint algorithms designed to work with C++ programs. Using a procedural language as a basis, [92] introduced Constraint Imperative Programming which has explicit constraints in the usual way, and also a new kind of constraints obtained by considering variable assignments such as $x = x + 1$ as time-stamped. Such assignments are treatable as constraints of the form $x_i = x_{i+1} + 1$. Finally, we mention Constraint Functional Programming [70] whose goal is the amalgamation of the ideas of functional programming found in the HOPE language with constraints.

There is work on languages and systems which are not generally regarded as constraint languages, but are nevertheless related to CLP languages. The development of symbolic algebra systems such as MACSYMA [186] concentrated on the solving of difficult algebraic problems. The programming language aspects are less developed. Languages for linear programming [152] provide little more than a primitive documentation facility for the array of coefficients which is input into a linear programming module.

In parallel with the development of these constraint languages, much work was done on the modelling of combinatorial problems as Constraint Satisfaction Problems (CSPs) and the development of techniques for solving such problems. The work was generally independent of any host language. (A possible exception is ALICE [164] which provided a wide variety of primitives to implement different search techniques.) One important development was the definition and study of several notions of consistency. This work had a significant influence on the later development of the CLP language CHIP. We refer the reader to [253] for an introduction to the basic techniques and results concerning CSPs. Finally, we mention the survey [41] which deals not just with constraint programming languages, but constraint-based programming techniques.

1.2. Logic Programming

Next we consider conventional logic programming (LP), and argue by example that the power of CLP cannot be obtained by making simple changes to LP systems. The question at hand is whether predicates in a logic program can be meaningfully regarded as constraints. That is, is a predicate with the same declarative semantics as a constraint a sufficient implementation of the constraint as per CLP? Consider, for example, the logic program

```
add(0, N, N).
add(s(N), M, s(K)) :- add(N, M, K).
```

where natural numbers n are represented by $s(s(\cdots(0)\cdots))$ with n occurrences of s. Clearly, the meaning of the predicate $add(n, m, k)$ coincides with the relation $n+m = k$. However, the query `add(N, M, K), add(N, M, s(K))`, which is clearly unsatisfiable, runs forever in a conventional LP system. The important point here is that a global test for the satisfiability of the two *add* constraints is not done by the underlying LP machinery.

In this example, the problem is that the *add* predicate is not invoked with a representation of the *add* constraints collected so far, and neither does it return such a representation (after having dealt with one more constraint). More concretely, the second subgoal of the query above is not given a representation of the fact that $N + M = K$.

A partial solution to this problem is the use of a delay mechanism. Roughly, the idea is that invocation of the predicate is delayed until its arguments are sufficiently instantiated. For example, if invocation of `add` is systematically delayed until its first argument is instantiated, then `add` behaves as in CLP when the first argument is ground. Thus the query `N = s(s(··· s(0)···)), add(N, M, K), add(N, M, s(K))` fails as desired. However, the original query `add(N, M, K), add(N, M, s(K))` will be delayed forever.

A total solution could, in principle, be obtained by simply adding two extra arguments to the predicate. One would be used for the input representation, and one for the output. This would mean that each time a constraint is dealt with, a representation of the entire set of constraints accumulated must be manipulated and a new representation constructed. But this is tantamount to a meta-level implementation of CLP in LP. Furthermore, this approach raises new challenges to efficient implementation.

Since LP is an instance of CLP, in which constraints are equations over terms, its solver also requires a representation of accumulated constraints. It happens, however, that there is no need for an explicit representation, such as the extra arguments discussed above. This is because the accumulated constraints can be represented by an mgu, and this, of course, is globally available via a simple binding mechanism.

1.3. CLP Languages

Viewing the subject rather broadly, constraint logic programming can be said to involve the incorporation of constraints and constraint "solving" methods in a logic-based language. This characterization suggests the possibility of many interesting languages, based on different constraints and different logics. However, to this point, work on CLP has almost exclusively been devoted to languages based on Horn clauses[1]. We now briefly describe these languages, concentrating on those that have received substantial development effort.

Prolog can be said to be a CLP language where the constraints are equations over the algebra of terms (also called the algebra of finite trees, or the Herbrand domain). The equations are implicit in the use of unification[2]. Almost every language we discuss incorporates Prolog-like terms in addition to other terms and constraints, so we will not discuss this aspect further. Prolog II [59] employs equations and disequations ($\neq$) over rational trees (an extension of the finite trees of Prolog to cyclic structures). It was the first logic language explicitly described as using constraints [61].

CLP($\mathcal{R}$) [133] has linear arithmetic constraints and computes over the real numbers. Nonlinear constraints are ignored (delayed) until they become effectively linear. CHIP [76] and Prolog III [64] compute over several domains. Both compute over Boolean domains: Prolog III over the well-known 2-valued Boolean algebra and CHIP over a larger Boolean algebra that contains symbolic values. Both CHIP and Prolog III perform linear arithmetic over the rational numbers. Separately (domains cannot be mixed) CHIP also performs linear arithmetic over bounded subsets of the integers (known as "finite domains"). Prolog III also computes over a domain of strings. There are now several languages which compute over finite domains in the manner of CHIP, including `clp(FD)`[75], Echidna [103] and Flang [181]. cc($\mathcal{FD}$) [116] is essentially a second-generation CHIP system.

LOGIN [7], and LIFE [9] compute over an order-sorted domain of feature trees. This domain provides a limited notion of object (in the object-oriented sense). The languages support a term syntax which is not first-order, although every term can be interpreted through first-order constraints. Unlike other CLP languages/domains,

[1] We note, however, some work combining constraints and resolution in first-order automated theorem-proving [242, 44].

[2] The language Absys [82], which was very similar to Prolog, used equations explicitly, making it more obviously a CLP language.

Prolog-like trees are essentially part of this domain, instead of being built on top of the domain. CIL [192] computes over a domain similar to feature trees.

BNR-Prolog [198] computes over three domains: the 2-valued Boolean algebra, finite domains, and arithmetic over the real numbers. In contrast to other CLP languages over arithmetic domains it computes solutions numerically, instead of symbolically. Trilogy [256, 257] computes over strings, integers and real numbers. Although its syntax is closer to that of C, 2LP [170] can be considered to be a CLP language permitting only a subset of Horn clauses. It computes with linear constraints over integers and real numbers.

CAL [4] computes over two domains: the real numbers, where constraints are equations between polynomials and a Boolean algebra with symbolic values, where equality between Boolean formulas expresses equivalence in the algebra. Instead of delaying non-linear constraints, CAL makes partial use of these constraints during computation. In the experimental system RISC-CLP(Real) [120] non-linear constraints are fully involved in the computation.

L_λ [189] and Elf [200] are derived from λ-Prolog [190] and compute over the values of closed typed lambda expressions. These languages are not based on Horn clauses (they include a universal quantifier) and were not originally described as CLP languages. However it is argued in [187] that their operational behavior is best understood as the behavior of a CLP language. An earlier language, Le Fun [8], also computed over this domain, and can be viewed as a CLP language with a weak constraint solver.

1.4. Synopsis

The remainder of this paper is organized into three main parts. In part one we provide a formal framework for CLP. Particular attention will be paid to operational semantics and operational models. As we have seen in examples, it is the operational interpretation of constraints, rather than the declarative interpretation, which distinguishes CLP from LP. In part two algorithm and data structure considerations are discussed. A crucial property of any CLP implementation is that its constraint handling algorithms are incremental. In this light we review several important solvers and their algorithms for the satisfiability, entailment, and delaying of constraints. We will also discuss the requirements of an inference engine for CLP. In part three we consider CLP applications. In particular, we discuss two rather different programming paradigms, one suited for the modelling of complex problems, and one for the solution of combinatorial problems.

In this survey we concentrate on the issues raised by the introduction of constraints to LP. Consequently we will ignore, or pass over quickly, those issues inherent in LP. We assume the reader is somewhat familiar with LP and basic first-order logic. Appropriate background can be obtained from [168] for LP and [223] for logic. For introductory papers on constraint logic programming and CLP languages we refer the reader to [63, 65, 156, 94]. For further reading on CLP we suggest other surveys [58, 109, 110], some collections of papers [20, 143, 111], and some books [107, 214]. More generally, papers on CLP appear in various journals and conference proceedings devoted to computational logic, constraint processing or symbolic computation.

1.5. Notation and Terminology

This paper will (hopefully) keep to the following conventions. Upper case letters generally denote collections of objects, while lower case letters generally denote individual objects. u, v, w, x, y, z will denote variables, s, t will denote terms, p, q will denote predicate symbols, f, g will denote function symbols, a will denote a constant, a, b, h will denote atoms, A will denote a collection of atoms, θ, ψ will denote substitutions, c will denote a constraint, C, S will denote collections of constraints, r will denote a rule, P, Q will denote programs, G will denote a goal, $\mathcal{D}$ will denote a structure, D will denote its set of elements, and d will denote an element of D. These symbols may be subscripted or have an over-tilde. $\tilde{x}$ denotes a sequence of distinct variables $x_1, x_2, \ldots, x_n$ for an appropriate n. $\tilde{s}$ denotes a sequence of (not necessarily distinct) terms $s_1, s_2, \ldots, s_n$ for an appropriate n. $\tilde{s} = \tilde{t}$ abbreviates $s_1 = t_1 \wedge s_2 = t_2 \wedge \cdots \wedge s_n = t_n$. $\exists_{-\tilde{x}}\ \phi$ denotes the existential closure of the formula ϕ *except* for the variables $\tilde{x}$, which remain unquantified. $\tilde{\exists}\ \phi$ denotes the full existential closure of the formula ϕ.

A *signature* defines a set of function and predicate symbols and associates an arity with each symbol[3]. If Σ is a signature, a Σ-*structure* $\mathcal{D}$ consists of a set D and an assignment of functions and relations on D to the symbols of Σ which respects the arities of the symbols. A first-order Σ-formula is built from variables, function and predicate symbols of Σ, the logical connectives $\wedge, \vee, \neg, \leftarrow, \rightarrow, \leftrightarrow$ and quantifiers over variables $\exists, \forall$ in the usual way [223]. A formula is closed if all variable occurrences in the formula are within the scope of a quantifier over the variable. A Σ-*theory* is a collection of closed Σ-formulas. A *model* of a Σ-theory T is a Σ-structure $\mathcal{D}$ such that all formulas of T evaluate to *true* under the interpretation provided by $\mathcal{D}$. A $\mathcal{D}$-*model* of a theory T is a model of T extending $\mathcal{D}$ (this requires that the signature of $\mathcal{D}$ be contained in the signature of T). We write $T, \mathcal{D} \models \phi$ to denote that the formula ϕ is valid in all $\mathcal{D}$-models of T.

In this paper, the set of function and predicate symbols defined in the constraint domain is denoted by Σ and the set of predicate symbols definable by a program is denoted by Π. A *primitive constraint* has the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and $p \in \Sigma$ is a predicate symbol. Every *constraint* is a (first-order) formula built from primitive constraints. The class of constraints will vary, but we will generally consider only a subset of formulas to be constraints. An *atom* has the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and $p \in \Pi$. A *CLP program* is a collection of *rules* of the form $a \leftarrow b_1, \ldots, b_n$ where a is an atom and the b_i's are atoms or constraints. a is called the *head* of the rule and $b_1, \ldots, b_n$ is called the *body*. Sometimes we represent the rule by $a \leftarrow c, B$, where c is the conjunction of constraints in the body and B is the collection of atoms in the body, and sometimes we represent the rule by $a \leftarrow B$, where B is the collection of atoms and constraints in the body. In one subsection we will also consider programs with negated atoms in the body. A *goal* (or *query*) G is a conjunction of constraints and atoms. A *fact* is a rule $a \leftarrow c$ where c is a constraint. Finally, we will identify conjunction and multiset union.

To simplify the exposition we assume that the rules are in a standard form, where all arguments in atoms are variables and each variable occurs in at most one

[3] In a many-sorted language this would include associating a sort with each argument and the result of each symbol. However, we will not discuss such details in this survey.

atom. This involves no loss of generality since a rule such as $p(t_1, t_2) \leftarrow C, q(s_1, s_2)$ can be replaced by the equivalent rule $p(x_1, x_2) \leftarrow x_1 = t_1, x_2 = t_2, y_1 = s_1, y_2 = s_2, C, q(y_1, y_2)$. We also assume that all rules defining the same predicate have the same head and that no two rules have any other variables in common (this is simply a matter of renaming variables). However in examples we relax these restrictions.

Programs will be presented in `teletype` font, and will generally follow the Edinburgh syntax. In particular, program variables begin with an upper-case letter, $[Head|Tail]$ denotes a list with head $Head$ and tail $Tail$, and [] denotes an empty list. In one variation from this standard we allow subscripts on program variables, to improve readability.

Part I
The Semantics of CLP Languages

Many languages based on definite clauses have quite similar semantics. The crucial insight of the CLP Scheme [129, 128] and the earlier scheme of [130, 131] was that a logic-based programming language, its operational semantics, its declarative semantics and the relationships between these semantics could all be parameterized by a choice of domain of computation and constraints. The resulting scheme defines the class of languages $CLP(\mathcal{X})$ obtained by instantiating the parameter $\mathcal{X}$.

We take the view that the parameter $\mathcal{X}$ stands for a 4-tuple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$. Here Σ is a signature, $\mathcal{D}$ is a Σ-structure, $\mathcal{L}$ is a class of Σ-formulas, and $\mathcal{T}$ is a first-order Σ-theory. Intuitively, Σ determines the predefined predicate and function symbols and their arities, $\mathcal{D}$ is the structure over which computation is to be performed, $\mathcal{L}$ is the class of constraints which can be expressed, and $\mathcal{T}$ is an axiomatization of (some) properties of $\mathcal{D}$. In the following section we define some important relationships between the elements of the 4-tuple, and give some examples of constraint domains.

We then give declarative and operational semantics for CLP programs, parameterized by $\mathcal{X}$. The declarative semantics are quite similar to the corresponding semantics of logic programs, and we cover them quickly. There are many variations of the resolution-based operational semantics, and we present the main ones. We also present the main soundness and completeness results that relate the two styles of semantics. Finally, we discuss some linguistic features that have been proposed as extensions to the basic CLP language.

2. Constraint Domains

For any signature Σ, let $\mathcal{D}$ be a Σ-structure (the domain of computation) and $\mathcal{L}$ be a class of Σ-formulas (the *constraints*). We call the pair $(\mathcal{D}, \mathcal{L})$ a *constraint domain.* In a slight abuse of notation we will sometimes denote the constraint domain by $\mathcal{D}$. We will make several assumptions, none of which is strictly necessary, to simplify the exposition. We assume

- The terms and constraints in $\mathcal{L}$ come from a first-order language[4].

[4] Without this assumption, some of the results we cite are not applicable, since there can be no appropriate first-order theory $\mathcal{T}$. The remaining assumptions can be omitted, at the expense

- The binary predicate symbol $=$ is contained in Σ and is interpreted as identity in $\mathcal{D}$.[5]
- There are constraints in $\mathcal{L}$ which are, respectively, identically true and identically false in $\mathcal{D}$.
- The class of constraints $\mathcal{L}$ is closed under variable renaming, conjunction and existential quantification.

We will denote the smallest set of constraints which satisfies these assumptions and contains all primitive constraints – the constraints *generated* by the primitive constraints – by $\mathcal{L}_\Sigma$. In general, $\mathcal{L}$ may be strictly larger than $\mathcal{L}_\Sigma$ since, for example, universal quantifiers or disjunction are permitted in $\mathcal{L}$; it also may be smaller, as in example 7 below. However we will usually take $\mathcal{L} = \mathcal{L}_\Sigma$. On occasion we will consider an extension of Σ and $\mathcal{L}$, to Σ^* and $\mathcal{L}^*$ respectively, so that there is a constant in Σ^* for every element of D.

We now present some example constraint domains. In practice, these are not always fully implemented, but we leave discussion of that until later. Most general purpose CLP languages incorporate some arithmetic domain, including BNR-Prolog [198], CAL [4], CHIP [76], CLP($\mathcal{R}$) [133], Prolog III [64], RISC-CLP(Real) [120].

Example 2.1. Let Σ contain the constants 0 and 1, the binary function symbols $+$ and $*$, and the binary predicate symbols $=$, $<$ and $\leq$. Let D be the set of real numbers and let $\mathcal{D}$ interpret the symbols of Σ as usual (i.e. $+$ is interpreted as addition, etc). Let $\mathcal{L}$ be the constraints generated by the primitive constraints. Then $\Re = (\mathcal{D}, \mathcal{L})$ is the constraint domain of arithmetic over the real numbers. If we omit from Σ the symbol $*$ then the corresponding constraint domain $\Re_{Lin} = (\mathcal{D}', \mathcal{L}')$ is the constraint domain of linear arithmetic over the real numbers. If the domain is further restricted to the rational numbers then we have a further constraint domain $\mathbf{Q}_{Lin}$. In constraints in $\Re_{Lin}$ and $\mathbf{Q}_{Lin}$ we will write terms such as 3 and $5x$ as abbreviations for $1+1+1$ and $x+x+x+x+x$ respectively[6]. Thus $\exists y\ 5x + y \leq 3 \wedge z \leq y - 1$ is a constraint in $\Re$, $\Re_{Lin}$ and $\mathbf{Q}_{Lin}$, whereas $x * x \leq y$ is a constraint only in $\Re$. If we extend $\mathcal{L}'$ to allow negated equations[7] (we will use the symbol $\neq$) then the resulting constraint domains $\Re^{\neq}_{Lin}$ and $\mathbf{Q}^{\neq}_{Lin}$ permit constraints such as $2x + y \leq 0 \wedge x \neq y$. Finally, if we restrict Σ to $\{0, 1, +, =\}$ we obtain the constraint domain $\Re_{LinEqn}$, where the only constraints are linear equations.

$\Re_{Lin}$ and $\mathbf{Q}_{Lin}$ (and $\Re^{\neq}_{Lin}$ and $\mathbf{Q}^{\neq}_{Lin}$) are essentially the same constraint domain: they have the same language of constraints and the two structures are elementarily equivalent [223]. In particular, a constraint solver for one is also a constraint solver for the other.

of a messier reformulation of definitions and results.

[5] This assumption is unnecessary when terms have a most general unifier in $\mathcal{D}$, as occurs in Prolog. Otherwise $=$ is needed to express parameter passing.

[6] Other syntactic sugar, such as the unary and binary minus symbol $-$, are allowed. Rational number coefficients can be used: all terms in the sugared constraint need only be multiplied by an appropriate number to reduce the coefficients to integers.

[7] Sometimes called *disequations*.

Prolog and standard logic programming can be viewed as constraint logic programming over the constraint domain of finite trees.

Example 2.2. Let Σ contain a collection of constant and function symbols and the binary predicate symbol $=$. Let D be the set of finite trees where: each node of each tree is labelled by a constant or function symbol, the number of children of each node is the arity of the label of the node, and the children are ordered. Let $\mathcal{D}$ interpret the function symbols of Σ as tree constructors, where each $f \in \Sigma$ of arity n maps n trees to a tree whose root is labelled by f and whose subtrees are the arguments of the mapping. The primitive constraints are equations between terms, and let $\mathcal{L}$ be the constraints generated by these primitive constraints. Then $\mathcal{FT} = (\mathcal{D}, \mathcal{L})$ is the Herbrand constraint domain, as used in Prolog. Typical constraints are $x = g(y)$ and $\exists z\ x = f(z,z) \wedge y = g(z)$. (It is unnecessary to write a quantifier in Prolog programs because all variables that appear only in constraints are implicitly existentially quantified.)

It was pointed out in [60] that complete (i.e. always terminating) unification which omits the occurs check solves equations over the rational trees.

Example 2.3. We take Σ and $\mathcal{L}$ as in example 2. D is the set of rational trees (see [69] for a definition) and the function symbols are interpreted as tree constructors, as before. Then $\mathcal{RT} = (\mathcal{D}, \mathcal{L})$ is the constraint domain of rational trees.

If we take the set of infinite trees instead of rational trees then we obtain a constraint domain that is essentially the same as $\mathcal{RT}$, in the same way that $\Re_{Lin}$ and $\mathbf{Q}_{Lin}$ are essentially the same: they have the same language of constraints and the two structures are elementarily equivalent [174].

The next domain contains objects similar to the previous domains, but has a different signature and constraint language [12][8], which results in slightly different expressive power. It can be viewed as the restriction of domains over which LOGIN [7] and LIFE [9] compute when all sorts are disjoint. The close relationship between the constraints and ψ-terms [5] is emphasized by a syntactic sugaring of the constraints.

Example 2.4. Let $\Sigma = \{=\} \cup S \cup F$ where S is a set of unary predicate symbols (sorts) and F is a set of binary predicate symbols (features). Let D be the set of (finite or infinite) trees where: each node of each tree is labelled by a sort, each edge of each tree is labelled by a feature and no node has two outbound edges with the same label. Such trees are called *feature trees*. Let $\mathcal{D}$ interpret each sort s as the set of feature trees whose root is labelled by s, and interpret each feature f as the set of pairs (t_1, t_2) of feature trees such that t_2 is the subtree of t_1 that is reached by the edge labelled by f. (If t_1 has no edge labelled by f then there is no pair (t_1, t_2) in the set.) Thus features are essentially partial functions. The domain of feature trees $\mathcal{FEAT} = (\mathcal{D}, \mathcal{L})$. A typical constraint is $wine(x) \wedge \exists y\ region(x,y) \wedge rutherglen(y) \wedge \exists y\ color(x,y) \wedge red(y)$, but there is

[8] A variant of this domain, with a slightly different signature, is used in [235].

also a sugared syntax which would represent this constraint as $x : wine[region \Rightarrow rutherglen, color \Rightarrow red]$.

The next constraint domain takes strings as the basic objects. It is used in Prolog III [64].

Example 2.5. Let Σ contain the binary predicate symbol $=$, the binary function symbol ., a constant λ, and a number of other constants. D is the set of finite strings of the constants. The symbol . is interpreted in $\mathcal{D}$ as string concatenation and λ is interpreted as the empty string. $\mathcal{L}$ is the set of constraints generated by equations between terms. Then $\mathcal{WE} = (\mathcal{D}, \mathcal{L})$ is the constraint domain of equations on strings, sometimes called the domain of word equations. An example constraint is $x.a = b.x$.

The constraint domain of Boolean values and functions is used in BNR-Prolog [198], CAL [4], CHIP [76] and Prolog III [64]. CAL and CHIP employ a more general constraint domain, which includes symbolic Boolean values.

Example 2.6. Let Σ contain the constants 0 and 1, the unary function symbol $\neg$, the binary function symbols $\wedge, \vee, \oplus, \Rightarrow$, and the binary predicate symbol $=$. Let D be the set $\{true, false\}$ and let $\mathcal{D}$ interpret the symbols of Σ as the usual Boolean functions (i.e. $\wedge$ is interpreted as conjunction, $\oplus$ is exclusive or, etc). Let $\mathcal{L}$ be the constraints generated by the primitive constraints. Then $\mathcal{BOOL} = (\mathcal{D}, \mathcal{L})$ is the (two-valued) Boolean constraint domain. An example constraint is $\neg(x \wedge y) = y$. In a slight abuse of notation, we allow a constraint $t = 1$ to be written simply as t so that, for example, $\neg(x \wedge y) \oplus y$ denotes the constraint $\neg(x \wedge y) \oplus y = 1$. For the more general constraint domain, let $\Sigma' = \Sigma \cup \{a_1, \ldots, a_i, \ldots\}$, where the a_i are constants. Let $\mathcal{L}'$ be the constraints generated by the Σ' primitive constraints and let $\mathcal{D}'$ be the free Boolean algebra generated by $\{a_1, \ldots, a_i, \ldots\}$. Then $\mathcal{BOOL}_\infty = (\mathcal{D}', \mathcal{L},)$ is the Boolean constraint domain with infinitely many symbolic values[9]. A constraint $c(\tilde{x}, \tilde{a})$ is satisfiable in $\mathcal{BOOL}_\infty$ iff $\mathcal{D} \models \exists \tilde{x} \forall \tilde{y}\ c(\tilde{x}, \tilde{y})$.

The finite domains of CHIP are best viewed as having the integers as the underlying structure, with a limitation on the language of constraints.

Example 2.7. Let $D = \mathbf{Z}$ and $\Sigma = \{\{\in [m,n]\}_{m \le n}, +, =, \neq, \le\}$. For every pair of integers m and n, the interval constraint $x \in [m,n]$ denotes that $m \le x \le n$. The other symbols in Σ have their usual meaning. Let $\mathcal{L}$ be the constraints c generated by the primitive constraints, restricted so that every variable in c is subject to an interval constraint. Then $\mathcal{FD} = (\mathcal{D}, \mathcal{L})$ is the constraint domain referred to as finite domains. (The domain of a variable x is the finite set of values which satisfy all unary constraints involving x.) A typical constraint in

[9] Only finitely many constants are used in any one program, so it can be argued that a finite Boolean algebra is a more appropriate domain of computation. However the two alternatives agree on satisfiability and constraint entailment (although not if an expanded language of constraints is permitted) and it is preferable to view the constraint domain as independent of the program. Currently it is not clear whether the alternatives agree on other constraint operations.

$\mathcal{FD}$ is $x \in [1,5] \wedge y \in [0,7] \wedge x \neq 3 \wedge x + 2y \leq 5 \wedge x + y \leq 9$. The domain of x is $\{1, 2, 4, 5\}$.

There are several other constraint domains of interest that we cannot exemplify here, for lack of space. They include pseudo-Boolean constraints (for example, [26]), which are intermediate between Boolean and integer constraints, order-sorted feature algebras [10], domains consisting of regular sets of strings [258], domains of finite sets [79], domains of CLP(Fun(D)) which employ a function variable [117], domains of functions expressed by λ-expressions [190, 8, 189, 200, 187], etc.

It is also possible to form a constraint domain directly from objects and operations in an application, instead of more general-purpose domains such as those above. This possibility has only been pursued in a limited form, where a general-purpose domain is extended by the ad hoc addition of primitive constraints. For example, in some uses of CHIP the finite domain is extended with a predicate symbol *element* [77]. The relation $element(x, l, t)$ expresses that t is the x'th element in the list l. We discuss such extensions further in Section 9.2

These constraint domains are expected to support (perhaps in a weakened form) the following tests and operations on constraints, which have a major importance in CLP languages. The first operation is the most important (it is almost obligatory), while the others might not be used in some CLP languages.

- The first is a test for *consistency* or *satisfiability*: $\mathcal{D} \models \tilde{\exists}\, c$.
- The second is the *implication* (or *entailment*) of one constraint by another: $\mathcal{D} \models c_0 \rightarrow c_1$. More generally, we may ask whether a disjunction of constraints is implied by a constraint: $\mathcal{D} \models c_0 \rightarrow \bigvee_{i=1}^{n} c_i$.
- The third is the *projection* of a constraint c_0 onto variables $\tilde{x}$ to obtain a constraint c_1 such that $\mathcal{D} \models c_1 \leftrightarrow \exists_{-\tilde{x}}\, c_0$. It is always possible to take c_1 to be $\exists_{-\tilde{x}}\, c_0$, but the aim is to compute the simplest c_1 with fewest quantifiers. In general it is not possible to eliminate all uses of the existential quantifier.
- The fourth is the detection that, given a constraint c, there is only one value that a variable x can take that is consistent with c. That is, $\mathcal{D} \models c(x, \tilde{z}) \wedge c(y, \tilde{w}) \rightarrow x = y$ or, equivalently, $\mathcal{D} \models \exists z \forall x, \tilde{y}\; c(x, \tilde{y}) \rightarrow x = z$. We say x is *determined* (or *grounded*) by c.

In Section 10 we will discuss problems and techniques which arise when implementing these operations in a CLP system. However, we point out here that some implementations of these operators – in particular, the test for satisfiability – are incomplete. In some cases it has been argued [67, 66, 22] that although an algorithm is incomplete with respect to the desired constraint domain, it is complete with respect to another (artificially constructed) constraint domain.

We now turn to some properties of constraint domains which will be used later. The first two – solution compactness and satisfaction completeness – were introduced as part of the CLP Scheme.

Definition 2.1. Let d range over elements of D and c, c_i range over constraints in $\mathcal{L}$, and let I be a possibly infinite index set. A constraint domain $(\mathcal{D}, \mathcal{L})$ is *solution compact* [128, 129] if it satisfies the following conditions:

$$(SC_1) \qquad \forall d\ \exists \{c_i\}_{i \in I}\ s.t.\ \mathcal{D} \models \forall x\ x = d \leftrightarrow \bigwedge_{i \in I} c_i(x)$$

$$(SC_2) \qquad \forall c\ \exists \{c_i\}_{i \in I}\ s.t.\ \mathcal{D} \models \forall \tilde{x}\ \neg c(\tilde{x}) \leftrightarrow \bigvee_{i \in I} c_i(\tilde{x})$$

Roughly speaking, SC_1 is satisfied iff every element d of D can be defined by a (possibly infinite) conjunction of constraints, and SC_2 is satisfied iff the complement of each constraint c in $\mathcal{L}$ can be described by a (possibly infinite) disjunction of constraints.

The definition of SC_2 in [128] is not quite equivalent to the definition in [129] which we paraphrase above; see [175]. It turns out that SC_1 is not necessary for the results we present; we include it only for historical accuracy. There is no known natural constraint domain for which SC_2 does not hold. There are, however, some artificial constraint domains for which it fails.

Example 2.8. Let $\Re^+_{Lin}$ denote the constraint domain obtained from $\Re_{Lin}$ by adding the unary primitive constraint $x \neq \pi$. The negation of this constraint (i.e. $x = \pi$) cannot be represented as a disjunction of constraints in $\Re^+_{Lin}$. Thus $\Re^+_{Lin}$ is not solution compact.

The theory $\mathcal{T}$ in the parameter of the CLP scheme is intended to axiomatize some of the properties of $\mathcal{D}$. We place some conditions on $\mathcal{D}$ and $\mathcal{T}$ to ensure that $\mathcal{T}$ reflects $\mathcal{D}$ sufficiently. The first two conditions ensure that $\mathcal{D}$ and $\mathcal{T}$ agree on satisfiability of constraints, while the addition of the third condition guarantees that every unsatisfiability in $\mathcal{D}$ is also detected by $\mathcal{T}$. The theory $\mathcal{T}$ and these conditions mainly play a role in the completeness results of Section 6.

Definition 2.2. For a given signature Σ, let $(\mathcal{D}, \mathcal{L})$ be a constraint domain with signature Σ, and $\mathcal{T}$ be a Σ-theory. We say that $\mathcal{D}$ and $\mathcal{T}$ *correspond* on $\mathcal{L}$ if
- $\mathcal{D}$ is a model of $\mathcal{T}$, and
- for every constraint $c \in \mathcal{L}$, $\mathcal{D} \models \tilde{\exists}\ c$ iff $\mathcal{T} \models \tilde{\exists}\ c$.

We say $\mathcal{T}$ is *satisfaction complete* with respect to $\mathcal{L}$ if for every constraint $c \in \mathcal{L}$, either $\mathcal{T} \models \tilde{\exists}\ c$ or $\mathcal{T} \models \neg\tilde{\exists}\ c$.

Satisfaction completeness is a weakening of the notion of a complete theory [223]. Thus, for example, the theory of the real closed fields [247] corresponds and is satisfaction complete with respect to $\Re$ since the domain is a model of this theory and the theory is complete. Clark's axiomatization of unification [55] defines a satisfaction complete theory with respect to $\mathcal{FT}$ which is not complete when there are only finitely many function symbols [174].

The notion of independence of negative constraints plays a significant role in constraint logic programming[10]. In [62], Colmerauer used independence of inequations to simplify the test for satisfiability of equations and inequations on the rational

[10] It is also closely related to the model-theoretic properties that led to an interest in Horn formulas [172, 121].

trees. (The independence of inequations states: if a conjunction of positive and negative equational constraints are inconsistent then one of the negative constraints is inconsistent with the positive constraints.) Independence of negative constraints has been investigated in greater generality in [163]. The property has been shown to hold for several classes of constraints including equations on finite, rational and infinite trees [161, 160, 174], linear real arithmetic constraints (where only equations may be negated) [162], sort and feature constraints on feature trees [12], and infinite Boolean algebras with positive constraints [106], among others [163]. We consider a restricted form of independence of negative constraints [177].

Definition 2.3. A constraint domain $(\mathcal{D}, \mathcal{L})$ has the Independence of Negated Constraints property if, for all constraints $c, c_1, \ldots, c_n \in \mathcal{L}$,

$$\mathcal{D} \models \tilde{\exists}\ c \wedge \neg c_1 \wedge \cdots \wedge \neg c_n \text{ iff } \mathcal{D} \models \tilde{\exists}\ c \wedge \neg c_i \text{ for } i = 1, \ldots, n.$$

The fact that $\mathcal{L}$ is assumed to be closed under conjunction and existential quantification is an important restriction in the above definition. For example, Colmerauer's work is not applicable in this setting, since that dealt only with primitive constraints. Neither are many of the other results cited above, at least not in their full generality. However there are still several useful constraint domains known to have this property, including the algebras of finite, rational and infinite trees with equational constraints, when there are infinitely many function symbols [161, 174], feature trees with infinitely many sorts and features [12], linear arithmetic equations over the rational or real numbers, and infinite Boolean algebras with positive constraints [106].

Example 2.9. In the Herbrand constraint domain $\mathcal{FT}$ with only two function symbols, a constant a and a unary function f, it is easily seen that the following statements are true: $\mathcal{FT} \models \exists x, y, z\ x = f(y) \wedge \neg y = a \wedge \neg y = f(z)$; $\mathcal{FT} \models \exists x, y\ x = f(y) \wedge \neg y = a$; $\mathcal{FT} \models \exists x, y, z\ x = f(y) \wedge \neg y = f(z)$. This is an example of the independence of inequations for $\mathcal{FT}$. However, when we consider the full class of constraints of $\mathcal{FT}$ we have the following facts. The statement $\mathcal{FT} \models \exists x, y\ x = f(y) \wedge \neg y = a \wedge \neg \exists z\ y = f(z)$ is not true, since every finite tree y is either the constant a or has the form $f(z)$ for some finite tree z. On the other hand, both $\mathcal{FT} \models \exists x, y\ x = f(y) \wedge \neg y = a$ and $\mathcal{FT} \models \exists x, y, z\ x = f(y) \wedge \neg \exists z\ y = f(z)$ are true. Thus, for these function symbols – and it is easy to see how to extend this example to any finite set of function symbols – the independence of negated constraints does not hold.

As is clear from [220], constraint domains (and constraints) are closely related to the *information systems* (and their elements) used by Scott to present his domain theory. Information systems codify notions of consistency and entailment among elements, which can be interpreted as satisfiability and implication of constraints on a single variable. Saraswat [217, 215] extended the notion of information system to *constraint systems*[11] (which allow many variables) and showed that some of the motivating properties of information systems continue to hold.

[11] although [215] does not treat consistency, only entailment.

Constraint systems (we will not give a formal definition here) can be viewed as abstractions of constraint domains which eliminate consideration of a particular structure $\mathcal{D}$; the relation $\mathcal{D} \models c_1 \wedge \cdots \wedge c_n \rightarrow c$ among constraints $c, c_1, \ldots, c_n$ is abstracted to the relation $c_1, \ldots, c_n \vdash c$ (and the satisfiability relation $\mathcal{D} \models \tilde{\exists}\, c_1 \wedge \cdots \wedge c_n$ among constraints can be abstracted to a set Con of all consistent finite sets of constraints $\{c_1, \ldots, c_n\}$ [220, 215]). Many of the essential semantic details of a constraint domain are still present in the corresponding constraint system, although properties such as solution compactness and independence of negated constraints cannot be expressed without more detail than a constraint system provides.

3. Logical Semantics

There are two common logical semantics of CLP programs over a constraint domain $(\mathcal{D}, \mathcal{L})$. The first interprets a rule

$$p(\tilde{x}) \leftarrow b_1, \ldots, b_n$$

as the logic formula

$$\forall \tilde{x}, \tilde{y} \;\; p(\tilde{x}) \vee \neg b_1 \vee \cdots \vee \neg b_n$$

where $\tilde{x} \cup \tilde{y}$ is the set of all free variables in the rule. The collection of all such formulas corresponding to rules of P gives a theory also denoted by P.

The second logical semantics associates a logic formula to each predicate in Π. If the set of all rules of P with p in the head is

$$\begin{array}{ll} p(\tilde{x}) & \leftarrow B_1 \\ p(\tilde{x}) & \leftarrow B_2 \\ & \cdots \\ p(\tilde{x}) & \leftarrow B_n \end{array}$$

then the formula associated with p is

$$\begin{array}{lll} \forall \tilde{x}\; p(\tilde{x}) \leftrightarrow & & \exists \tilde{y}_1 \;\, B_1 \\ & \vee & \exists \tilde{y}_2 \;\, B_2 \\ & & \cdots \\ & \vee & \exists \tilde{y}_n \;\, B_n \end{array}$$

where $\tilde{y}_i$ is the set of variables in B_i except for variables in $\tilde{x}$. If p does not occur in the head of a rule of P then the formula is

$$\forall \tilde{x} \;\neg p(\tilde{x})$$

The collection of all such formulas is called the *Clark completion* of P, and is denoted by P^*.

A *valuation* is a mapping from variables to D, and the natural extension which maps terms to D and formulas to closed $\mathcal{L}^*$-formulas. If X is a set of facts then $[X]_{\mathcal{D}} = \{v(a) \mid (a \leftarrow c) \in X, \mathcal{D} \models v(c)\}$. A $\mathcal{D}$-interpretation of a formula is an interpretation of the formula with the same domain as $\mathcal{D}$ and the same interpretation for the symbols in Σ as $\mathcal{D}$. It can be represented as a subset of $\mathcal{B}_{\mathcal{D}}$ where $\mathcal{B}_{\mathcal{D}} = \{p(\tilde{d}) \mid p \in \Pi, \tilde{d} \in D^k\}$. A $\mathcal{D}$-model of a closed formula is a $\mathcal{D}$-interpretation which is a model of the formula.

Let $\mathcal{T}$ denote a satisfaction complete theory for $(\mathcal{D}, \mathcal{L})$. The usual logical semantics are based on the $\mathcal{D}$-models of P and the models of $P^*, \mathcal{T}$. The least $\mathcal{D}$-model of a formula Q under the subset ordering is denoted by $lm(Q, \mathcal{D})$, and the greatest is denoted by $gm(Q, \mathcal{D})$. A *solution* to a query G is a valuation v such that $v(G) \subseteq lm(P, \mathcal{D})$.

4. Fixedpoint Semantics

The fixedpoint semantics we present are based on one-step consequence functions $T_P^{\mathcal{D}}$ and $S_P^{\mathcal{D}}$, and the closure operator $[\![P]\!]$ generated by $T_P^{\mathcal{D}}$. The functions $T_P^{\mathcal{D}}$ and $[\![P]\!]$ map over $\mathcal{D}$-interpretations. The set of $\mathcal{D}$-interpretations forms a complete lattice under the subset ordering, and these functions are continuous on $\mathcal{B}_{\mathcal{D}}$.

$$
\begin{aligned}
T_P^{\mathcal{D}}(I) = \{p(\tilde{d}) \mid \quad & p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n \text{ is a rule of } P, a_i \in I,\ i = 1, \ldots, n, \\
& v \text{ is a valuation on } \mathcal{D} \text{ such that} \\
& \mathcal{D} \models v(c),\ v(\tilde{x}) = \tilde{d}, \text{ and } v(b_i) = a_i,\ i = 1, \ldots, n\}
\end{aligned}
$$

$[\![P]\!]$ is the closure operator generated by $T_P^{\mathcal{D}}$. It represents a deductive closure based on the rules of P. Let Id be the identity function, and define $(f+g)(x) = f(x) \cup g(x)$. Then $[\![P]\!](I)$ is the least fixedpoint of $T_P^{\mathcal{D}} + Id$ greater than I, and the least fixedpoint of $T_{P \cup I}^{\mathcal{D}}$.

The function $S_P^{\mathcal{D}}$ is defined on sets of facts, which form a complete lattice under the subset ordering. We denote the closure operator generated from $S_P^{\mathcal{D}}$ by $\langle\!\langle P \rangle\!\rangle$. Both these functions are continuous.

$$
\begin{aligned}
S_P^{\mathcal{D}}(I) = \{p(\tilde{x}) \leftarrow c \mid \quad & p(\tilde{x}) \leftarrow c', b_1, \ldots, b_n \text{ is a rule of } P, \\
& a_i \leftarrow c_i \in I,\ i = 1, \ldots, n, \text{ the rule and facts renamed apart,} \\
& \mathcal{D} \models c \leftrightarrow c' \wedge \textstyle\bigwedge_{i=1}^{n} c_i \wedge a_i = b_i\}
\end{aligned}
$$

We denote the least fixedpoint of a function f by $lfp(f)$ and the greatest fixedpoint by $gfp(f)$. These fixedpoints exists for the functions of interest, since they are monotonic functions on complete lattices. For a function f mapping $\mathcal{D}$-interpretations to $\mathcal{D}$-interpretations, we define the upward and downward iteration of f as follows.

$$
\begin{aligned}
f \uparrow 0 &= \emptyset \\
f \uparrow (\alpha + 1) &= f(f \uparrow \alpha) \\
f \uparrow \beta &= \textstyle\bigcup_{\alpha < \beta} f \uparrow \alpha \quad \text{if } \beta \text{ is a limit ordinal}
\end{aligned}
$$

$$
\begin{aligned}
f \downarrow 0 &= \mathcal{B}_{\mathcal{D}} \\
f \downarrow (\alpha + 1) &= f(f \downarrow \alpha) \\
f \downarrow \beta &= \textstyle\bigcap_{\alpha < \beta} f \downarrow \alpha \quad \text{if } \beta \text{ is a limit ordinal}
\end{aligned}
$$

We can take as semantics $lfp(S_P^{\mathcal{D}})$ or $lfp(T_P^{\mathcal{D}})$. The two functions involved are related in the following way: $[S_P^{\mathcal{D}}(I)]_{\mathcal{D}} = T_P^{\mathcal{D}}([I]_{\mathcal{D}})$. Consequently $[lfp(S_P^{\mathcal{D}})]_{\mathcal{D}} = lfp(T_P^{\mathcal{D}})$. $lfp(S_P^{\mathcal{D}})$ corresponds to the s-semantics [87] for languages with constraints [95]. Fixedpoint semantics based on sets of clauses [34] also extend easily to CLP languages.

Based largely on the facts that the $\mathcal{D}$-models of P are the fixedpoints of $[\![P]\!]$ and the $\mathcal{D}$-models of P^* are the fixedpoints of $T_P^{\mathcal{D}}$, we have the following connections between the logical and fixedpoint semantics, just as in standard logic programming.

Proposition 4.1. Let P, P_1, P_2 be CLP programs and Q a set of facts over a constraint domain $\mathcal{D}$ with corresponding theory $\mathcal{T}$. Then:

- $T_P^{\mathcal{D}} \uparrow \omega = lfp(T_P^{\mathcal{D}}) = [lfp(S_P^{\mathcal{D}})]_{\mathcal{D}} = [\![P]\!](\emptyset)$
- $lm(P, \mathcal{D}) = [\{h \leftarrow c \mid P^*, \mathcal{D} \models (h \leftarrow c)\}]_{\mathcal{D}} = [\{h \leftarrow c \mid P^*, \mathcal{T} \models (h \leftarrow c)\}]_{\mathcal{D}}$
- $lm(P^*, \mathcal{D}) = lm(P, \mathcal{D}) = lfp(T_P^{\mathcal{D}})$
- $gm(P^*, \mathcal{D}) = gfp(T_P^{\mathcal{D}})$
- $[\![P]\!]([Q]_{\mathcal{D}}) = [\![P \cup Q]\!](\emptyset) = lm(P \cup Q, \mathcal{D})$
- $\langle\!\langle P \rangle\!\rangle(Q) = \langle\!\langle P \cup Q \rangle\!\rangle(\emptyset) = lfp(S_{P \cup Q}^{\mathcal{D}})$
- $\mathcal{D} \models P_1 \leftrightarrow P_2$ iff $[\![P_1]\!] = [\![P_2]\!]$

We will need the following terminology later. P is said to be $(\mathcal{D}, \mathcal{L})$-*canonical* iff $gfp(T_P^{\mathcal{D}}) = T_P^{\mathcal{D}} \downarrow \omega$. Canonical logic programs, but not constraint logic programs, were first studied by [136] who showed that every logic program is equivalent (wrt the success and finite failure sets) to a canonical logic program. The proof here was not constructive, but subsequently, [259] provided an algorithm to generate the canonical logic program[12]. Like many other kinds of results in traditional logic programming, these results are likely to extend to CLP in a straightforward way.

5. Top-down Execution

The phrase "top-down execution" covers a multitude of operational models. We will present a fairly general framework for operational semantics in which we can describe the operational semantics of some major CLP systems.

We will present the operational semantics as a transition system on *states*: tuples $\langle A, C, S \rangle$ where A is a multiset of atoms and constraints, and C and S are multisets of constraints. The constraints C and S are referred to as the *constraint store* and, in implementations, are acted upon by a constraint solver. Intuitively, A is a collection of as-yet-unseen atoms and constraints, C is the collection of constraints which are playing an *active* role (or are *awake*), and S is a collection of constraints playing a *passive* role (or are *asleep*). There is one other state, denoted by *fail*. To express more details of an operational semantics, it can be necessary to represent the collections of atoms and constraints more precisely. For example, to express the left-to-right Prolog execution order we might use a sequence of atoms rather than a multiset. However we will not be concerned with such details here.

[12] This proof was performed in the more general class of logic programs with negation.

We will assume as given a *computation rule* which selects a transition type and an appropriate element of A (if necessary) for each state[13]. The transition system is also parameterized by a predicate *consistent* and a function *infer*, which we will discuss later. An initial goal G for execution is represented as a state by $\langle G, \emptyset, \emptyset\rangle$.

The transitions in the transition system are:

$$\langle A \cup a, C, S\rangle \rightarrow_r \langle A \cup B, C, S \cup (a = h)\rangle$$

if a is selected by the computation rule, a is an atom, $h \leftarrow B$ is a rule of P, renamed to new variables, and h and a have the same predicate symbol. The expression $a = h$ is an abbreviation for the conjunction of equations between corresponding arguments of a and h. We say a is *rewritten* in this transition.

$$\langle A \cup a, C, S\rangle \rightarrow_r fail$$

if a is selected by the computation rule, a is an atom and, for every rule $h \leftarrow B$ of P, h and a have different predicate symbols.

$$\langle A \cup c, C, S\rangle \rightarrow_c \langle A, C, S \cup c\rangle$$

if c is selected by the computation rule and c is a constraint.

$$\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$$

if $(C', S') = infer(C, S)$.

$$\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle$$

if $consistent(C)$.

$$\langle A, C, S\rangle \rightarrow_s fail$$

if $\neg consistent(C)$.

The $\rightarrow_r$ transitions arise from resolution, $\rightarrow_c$ transitions introduce constraints into the constraint solver, $\rightarrow_s$ transitions test whether the active constraints are consistent, and $\rightarrow_i$ transitions infer more active constraints (and perhaps modify the passive constraints) from the current collection of constraints. We write $\rightarrow$ to refer to a transition of arbitrary type.

The predicate $consistent(C)$ expresses a test for consistency of C. Usually it is defined by: $consistent(C)$ iff $\mathcal{D} \models \tilde{\exists}\, C$, that is, a complete consistency test. However systems may employ a conservative but incomplete (or partial) test: if $\mathcal{D} \models \tilde{\exists}\, C$ then $consistent(C)$ holds, but sometimes $consistent(C)$ holds although $\mathcal{D} \models \neg\tilde{\exists}\, C$. One example of such a system is CAL [4] which computes over the domain of real numbers, but tests consistency over the domain of complex numbers.

The function $infer(C, S)$ computes from the current sets of constraints a new set of active constraints C' and passive constraints S'. Generally it can be understood as abstracting from S (or relaxing S) in the presence of C to obtain more active constraints. These are added to C to form C', and S is simplified to S'. We require

[13] A computation rule is a convenient fiction that abstracts some of the behavior of a CLP system. To be realistic, a computation rule should also depend on factors other than the state (for example, the history of the computation). We ignore these possibilities for simplicity.

that $\mathcal{D} \models (C \wedge S) \leftrightarrow (C' \wedge S')$, so that information is neither lost nor "guessed" by $infer$. The role that $infer$ plays varies widely from system to system. In Prolog, there are no passive constraints and we can define $infer(C, S) = (C \cup S, \emptyset)$. In CLP($\mathcal{R}$), non-linear constraints are passive, and $infer$ simply passes (the linearized version of) a constraint from S to C' when the constraint becomes linear in the context of C, and deletes the constraint from S. For example, if S is $x * y = z \wedge z * y = 2$ and C is $x = 4 \wedge z \leq 0$ then $infer(C, S) = (C', S')$ where C' is $x = 4 \wedge z \leq 0 \wedge 4y = z$ and S' is $z * y = 2$.

In a language like CHIP, $infer$ performs less obvious inferences. For example, if S is $x = y + 1$ and C is $2 \leq x \leq 5 \wedge 0 \leq y \leq 3$ then $infer(C, S) = (C', S')$ where C' is $2 \leq x \leq 4 \wedge 1 \leq y \leq 3$ and $S' = S$. (Note that we could also formulate the finite domain constraint solving of CHIP as having no passive constraints, but having an incomplete test for consistency. However the formulation we give seems to reflect the systems more closely.) Similarly, in languages employing interval arithmetic over the real numbers (such as BNR-Prolog) intervals are active constraints and other constraints are passive. In this case, $infer$ repeatedly computes smaller intervals for each of the variables, based on the constraints in S, terminating when no smaller interval can be derived (modulo the precision of the arithmetic). Execution of language constructs such as the cardinality operator [112], "constructive disjunction" [116] and special-purpose constructs (for example, in [77, 2]) can also be understood as $\rightarrow_i$ transitions, where these constructs are viewed as part of the language of constraints.

Generally, the active constraints are determined syntactically. As examples, in Prolog all equations are active, in CLP($\mathcal{R}$) all linear constraints are active, on the finite domains of CHIP all unary constraints (i.e. constraints on just one variable, such as $x < 9$ or $x \neq 0$) are active, and in the interval arithmetic of BNR-Prolog only intervals are active.

The stronger the collection of active constraints, the earlier failure will be detected, and the less searching is necessary. With this in mind, we might wish $infer$ to be as strong as possible: for every active constraint c, if $infer(C, S) = (C', S')$ and $\mathcal{D} \models (C \wedge S) \rightarrow c$, then $\mathcal{D} \models C' \rightarrow c$. However, this is not always possible[14]. Even if it were possible, it is generally not preferred, since the computational cost of a powerful $infer$ function can be greater than the savings achieved by limiting search.

A *CLP system* is determined by the constraint domain and a detailed operational semantics. The latter involves a computation rule and definitions for *consistent* and $infer$. We now define some significant properties of CLP systems. We distinguish the class of systems in which passive constraints play no role and the global consistency test is complete. These systems correspond to the systems treated in [128, 129].

Definition 5.1. Let $\rightarrow_{ris} = \rightarrow_r \rightarrow_i \rightarrow_s$ and $\rightarrow_{cis} = \rightarrow_c \rightarrow_i \rightarrow_s$. We say that a CLP system is *quick-checking* if its operational semantics can be described by $\rightarrow_{ris}$ and $\rightarrow_{cis}$. A CLP system is *progressive* if, for every state with a nonempty collection of atoms, every derivation from that state either fails, contains a $\rightarrow_r$ transition

[14] For example, in CLP($\mathcal{R}$), where linear constraints are active and nonlinear constraints are passive, if S is $y = x * x$ then we can take c to be $y \geq 2kx - k^2$, for any k. There is no finite collection C' of active constraints which implies all these constraints and is not stronger than S.

or contains a $\rightarrow_c$ transition. A CLP system is *ideal* if it is quick-checking, progressive, $infer$ is defined by $infer(C, S) = (C \cup S, \emptyset)$, and $consistent(C)$ holds iff $\mathcal{D} \models \tilde{\exists}\ C$.

In a quick-checking system, inference of new active constraints is performed and a test for consistency is made each time the collection of constraints in the constraint solver is changed. Thus, within the limits of *consistent* and $infer$, it finds inconsistency as soon as possible. A progressive system will never infinitely ignore the collection of atoms and constraints in the first part of a state during execution. All major implemented CLP systems are quick-checking and progressive, but most are not ideal.

A *derivation* is a sequence of transitions $\langle A_1, C_1, S_1 \rangle \rightarrow \cdots \rightarrow \langle A_i, C_i, S_i \rangle \rightarrow \cdots$. A state which cannot be rewritten further is called a *final state*. A derivation is *successful* if it is finite and the final state has the form $\langle \emptyset, C, S \rangle$. Let G be a goal with free variables $\tilde{x}$, which initiates a derivation and produces a final state $\langle \emptyset, C, S \rangle$. Then $\exists_{-\tilde{x}}\ C \wedge S$ is called the *answer constraint* of the derivation.

A derivation is *failed* if it is finite and the final state is $fail$. A derivation is *fair* if it is failed or, for every i and every $a \in A_i$, a is rewritten in a later transition. A computation rule is *fair* if it gives rise only to fair derivations. A goal G is *finitely failed* if, for any one fair computation rule, every derivation from G in an ideal CLP system is failed. It can be shown that if a goal is finitely failed then every fair derivation in an ideal CLP system is failed. A derivation *flounders* if it is finite and the final state has the form $\langle A, C, S \rangle$ where $A \neq \emptyset$.

The *computation tree* of a goal G for a program P in a CLP system is a tree with nodes labelled by states and edges labelled by $\rightarrow_r$, $\rightarrow_c$, $\rightarrow_i$ or $\rightarrow_s$ such that: the root is labelled by $\langle G, \emptyset, \emptyset \rangle$; for every node, all outgoing edges have the same label; if a node labelled by a state $\mathcal{S}$ has an outgoing edge labelled by $\rightarrow_c$, $\rightarrow_i$ or $\rightarrow_s$ then the node has exactly one child, and the state labelling that child can be obtained from $\mathcal{S}$ via a transition $\rightarrow_c$, $\rightarrow_i$ or $\rightarrow_s$ respectively; if a node labelled by a state $\mathcal{S}$ has an outgoing edge labelled by $\rightarrow_r$ then the node has a child for each rule in P, and the state labelling each child is the state obtained from $\mathcal{S}$ by the $\rightarrow_r$ transition for that rule; for each $\rightarrow_r$ and $\rightarrow_c$ edge, the corresponding transition uses the atom or constraint selected by the computation rule.

Every branch of a computation tree is a derivation, and, given a computation rule, every derivation following that rule is a branch of the corresponding computation tree. Different computation rules can give rise to computation trees of radically different sizes. Existing CLP languages use computation rules based on the Prolog left-to-right computation rule (which is not fair). We will discuss linguistic features intended to improve on this rule in Section 9.1.

The problem of finding answers to a query can be seen as the problem of searching a computation tree. Most CLP languages employ a depth-first search with chronological backtracking, as in Prolog (although there have been suggestions to use dependency-directed backtracking [71]). Since depth-first search is incomplete on infinite trees, not all answers are computed. The depth-first search can be incorporated in the semantics in the same way as is done for Prolog (see, for example, [18, 17]), but we will not go into details here. In Section 8 we will discuss a class of CLP languages which use a top-down execution similar to the one outlined above, but do not use backtracking.

Consider the transition

$$\langle A, C, S\rangle \to_g \langle A, C', \emptyset\rangle$$

where C' is a set of equations in $\mathcal{L}^*$ such that $\mathcal{D} \models C' \to (C \wedge S)$ and, for every variable x occurring in C or S, C' contains an equation $x = d$ for some constant d. Thus $\to_g$ grounds all variables in the constraint solver. We also have the transitions

$$\langle A, C, S\rangle \to_g fail$$

if no such C' exists (i.e. $C \wedge S$ is unsatisfiable in $\mathcal{D}$). A *ground derivation* is a derivation composed of $\to_r\to_g$ and $\to_c\to_g$.

We now define three sets that crystallize three aspects of the operational semantics. The success set $SS(P)$ collects the answer constraints to simple goals $p(\tilde{x})$. The finite failure set $FF(P)$ collects the set of simple goals which are finitely failed. The ground finite failure set $GFF(P)$ collects the set of grounded atoms, all of whose fair ground derivations are failed.

$SS(P) = \{p(\tilde{x}) \leftarrow c \mid \langle p(\tilde{x}), \emptyset, \emptyset\rangle \to^* \langle \emptyset, c', c''\rangle,\ \mathcal{D} \models c \leftrightarrow \exists_{-\tilde{x}}\, c' \wedge c''\}$.
$FF(P) = \{p(\tilde{x}) \leftarrow c \mid$ for every fair derivation, $\langle p(\tilde{x}), c, \emptyset\rangle \to^* fail\}$.
$GFF(P) = \{p(\tilde{d}) \mid$ for every fair ground derivation, $\langle p(\tilde{d}), \emptyset, \emptyset\rangle \to^* fail\}$.

6. Soundness and Completeness Results

We now present the main relationships between the declarative semantics and the top-down operational semantics. To keep things simple, we consider only ideal CLP systems. However many of the results hold much more generally. The soundness results hold for any CLP system, because of restrictions we place on *consistent* and *infer*. Completeness results for successful derivations require only that the CLP system be progressive.

Theorem 6.1. Consider a program P in the CLP language determined by a 4-tuple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$ where $\mathcal{D}$ and $\mathcal{T}$ correspond on $\mathcal{L}$, and executing on an ideal CLP system. Then:

1. $SS(P) = lfp(S_P^{\mathcal{D}})$ and $[SS(P)]_{\mathcal{D}} = lm(P, \mathcal{D})$.
2. If the goal G has a successful derivation with answer constraint c, then $P, \mathcal{T} \models c \to G$.
3. Suppose $\mathcal{T}$ is satisfaction complete wrt $\mathcal{L}$. If G has a finite computation tree, with answer constraints $c_1, \ldots, c_n$, then $P^*, \mathcal{T} \models G \leftrightarrow c_1 \vee \cdots \vee c_n$.
4. If $P, \mathcal{T} \models c \to G$ then there are derivations for the goal G with answer constraints $c_1, \ldots, c_n$ such that $\mathcal{T} \models c \to \bigvee_{i=1}^{n} c_i$. If, in addition, $(\mathcal{D}, \mathcal{L})$ has independence of negated constraints then the result holds for $n = 1$ (i.e. without disjunction).
5. Suppose $\mathcal{T}$ is satisfaction complete wrt $\mathcal{L}$. If $P^*, \mathcal{T} \models G \leftrightarrow c_1 \vee \cdots \vee c_n$ then G has a computation tree with answer constraints $c'_1, \ldots, c'_m$ (and possibly others) such that $\mathcal{T} \models c_1 \vee \cdots \vee c_n \leftrightarrow c'_1 \vee \cdots \vee c'_m$.
6. Suppose $\mathcal{T}$ is satisfaction complete wrt $\mathcal{L}$.
   The goal G is finitely failed for P iff $P^*, \mathcal{T} \models \neg G$.

7. $gm(P^*, \mathcal{D}) = \mathcal{B}_\mathcal{D} - GFF(P)$.
8. Suppose $(\mathcal{D}, \mathcal{L})$ is solution compact. $T_P^\mathcal{D} \downarrow \omega = \mathcal{B}_\mathcal{D} - [FF(P)]_\mathcal{D}$.
9. Suppose $(\mathcal{D}, \mathcal{L})$ is solution compact.
   P is $(\mathcal{D}, \mathcal{L})$-canonical iff $[FF(P)]_\mathcal{D} = [\{h \leftarrow c \mid P^*, \mathcal{D} \models \neg(h \wedge c)\}]_\mathcal{D}$.

Most of these results are from [128, 129], but there are also some from [95, 173, 177]. Results 8 and 9 of the above theorem (which are equivalent) are the only results, of those listed above, which require solution compactness. In fact, the properties shown are equivalent to SC_2, the second condition of solution compactness [177]; as mentioned earlier, SC_1 is not needed. In soundness results (2, 3 and half of 6), $\mathcal{T}$ can be replaced by $\mathcal{D}$. If we omit our assumption of a first-order language of constraints (see Section 2) then only results 1, 2, 3, 7, 8, 9 and the soundness half of 6 (replacing $\mathcal{T}$ by $\mathcal{D}$ where necessary) continue to hold.

The strong form of completeness of successful derivations (result 4) [173] provides an interesting departure from the conventional logic programming theory. It shows that in CLP it is necessary, in general, to consider and combine several successful derivations and their answers to establish that $c \rightarrow G$ holds, whereas only one successful derivation is necessary in standard logic programming. The other results in this theorem are more direct liftings of results in the logic programming theory.

The CLP Scheme provides a framework in which the lifting of results from LP to CLP is almost trivial. By replacing the Herbrand universe by an arbitrary constraint domain $\mathcal{D}$, unifiability by constraint satisfaction, Clark's equality theory by a corresponding satisfaction-complete theory, etc., most results (and even their proofs) lift from LP to CLP. The lifting is discussed in greater detail in [177]. Furthermore, most operational aspects of LP (and Prolog) can be interpreted as logical operations, and consequently these operations (although not their implementations) also lift to CLP. One early example is matching, which is used in various LP systems (e.g. GHC, NU-Prolog) as a basis for affecting the computation rule; the corresponding operation in CLP is constraint entailment [173].

The philosophy of the CLP Scheme [129] gives primacy to the structure $\mathcal{D}$ over which computation is performed, and less prominence to the theory $\mathcal{T}$. We have followed this approach. However, it is also possible to start with a satisfaction complete theory $\mathcal{T}$ (see, for example [173]) without reference to a structure. We can arbitrarily choose a model of $\mathcal{T}$ as the structure $\mathcal{D}$ and the same results apply. Another variation [118] considers a collection $\mathbf{D}$ of structures and defines $consistent(C)$ to hold iff for some structure $\mathcal{D} \in \mathbf{D}$ we have $\mathcal{D} \models \tilde{\exists}\, C$. Weaker forms of the soundness and completeness of successful derivations apply in this case.

7. Bottom-Up Execution

Bottom-up execution has its main use in database applications. The set-at-a-time processing limits the number of accesses to secondary storage in comparison to tuple-at-a-time processing (as in top-down execution), and the simple semantics gives great scope for query optimization.

Bottom-up execution is also formalized as a transition system. For every rule r of the form $h \leftarrow c, b_1, \ldots, b_n$ in P and every set A of facts there is a transition

$$A \leadsto A \cup \{h \leftarrow c' \mid a_i \leftarrow c_i, i = 1, \ldots, n \text{ are elements of } A, \mathcal{D} \models c' \leftrightarrow c \wedge \bigwedge_{i=1}^{n} c_i \wedge b_i = a_i\}$$

In brief, then, we have $A \leadsto A \cup S_r^{\mathcal{D}}(A)$, for every set A and every rule r in P ($S_P^{\mathcal{D}}$ was defined in Section 4). An execution is a sequence of transitions. It is *fair* if each rule is applied infinitely often. The limit of ground instances of sets generated by fair executions is independent of the order in which transitions are applied, and is regarded as the result of the bottom-up execution. If Q is an initial set of facts and P is a program, and A is the result of a fair bottom-up execution then $A = SS(P \cup Q) = \langle\!\langle P \rangle\!\rangle(Q)$ and $[\![P]\!]([Q]_{\mathcal{D}}) = [A]_{\mathcal{D}}$.

An execution $Q = X_0 \leadsto X_1 \leadsto \cdots X_i \leadsto \cdots$ *terminates* if, for some m and every $i > m$, $X_i = X_m$. We say P is *finitary* if for every finite initial set of facts Q and every fair execution, there is a k such that $[X_i]_{\mathcal{D}} = [X_k]_{\mathcal{D}}$ for all $i \geq k$. However, execution can be non-terminating, even when the program is finitary and the initial set is finite.

Example 7.1. Consider the following program P on the constraint domain $\Re_{Lin}$:

```
p(X+1) ← p(X)
p(X) ← X ≥ 5
p(X) ← X ≤ 5
```

Straightforward bottom-up computation gives $\{p(x) \leftarrow x \geq 5, p(x) \leftarrow x \geq 6, p(x) \leftarrow x \geq 7, \ldots\} \cup \{p(x) \leftarrow x \leq 5, p(x) \leftarrow x \leq 6, p(x) \leftarrow x \leq 7, \ldots\}$, and does not terminate. We also have $lfp(T_P^{\mathcal{D}}) = T_P^{\mathcal{D}} \uparrow 1 = \{p(d) \mid d \in \Re\}$.

A necessary technique is to test whether a new fact is subsumed by the current set of facts, and accumulate only unsubsumed facts. A fact $p(\tilde{x}) \leftarrow c$ is *subsumed* by the facts $p(\tilde{x}) \leftarrow c_i, i = 1, \ldots, n$ (with respect to $(\mathcal{D}, \mathcal{L})$) if $\mathcal{D} \models c \rightarrow \bigvee_{i=1}^{n} c_i$. The transitions in the modified bottom-up execution model are

$$A \leadsto A \cup reduce(S_P^{\mathcal{D}}(A), A)$$

where $reduce(X, Y)$ eliminates from X all elements subsumed by Y. Under this execution model every finitary program terminates on every finite initial set Q.

Unfortunately, checking subsumption is computationally expensive, in general. If the constraint domain $(\mathcal{D}, \mathcal{L})$ does not satisfy the independence of negated constraints then the problem of showing that a new fact is not subsumed is at least NP-hard (see [236] for the proof in one constraint domain). In constraint domains with independence of negated constraints the problem is not so bad: the new fact only needs to be checked against one fact at a time [177]. (Classical database optimizations are also more difficult without independence of negated constraints [149, 177].) A pragmatic approach to the problem of subsumption in $\Re_{Lin}$ is given in [236]. Some work avoids the problem of subsumption by allowing only ground facts in the database and intermediate computations.

Even with subsumption, there is still the problem that execution might not terminate (for example, if P is not finitary). The approach of [144] is to restrict the constraint domains $\mathcal{D}$ to those which only permit the computation of finitely representable relations from finitely representable relations. This requirement is slightly weaker than requiring that all programs are finitary, but it is not clear that there is a practical difference. Regrettably, very few constraint domains satisfy this condition, and those which do have limited expressive power.

The alternative is to take advantage of P and a specific query (or a class of queries). A transformation technique such as magic templates [206] produces a program P^{mg} that is equivalent to P for the specific query. Other techniques [146, 193, 237, 147] attempt to further limit execution by placing constraints at appropriate points in the program. Analyses can be used to check that execution of the resulting program terminates [151, 211, 35], although most work has ignored the capability of using constraints in the answers.

Comparatively little work has been done on the nuts and bolts of implementing bottom-up execution for CLP programs, with all the work addressing the constraint domain $\Re_{Lin}$. [144] suggested the use of intervals, computed as the projection of a collection of constraints, as the basis for indexing on constrained variables. Several different data structures, originally developed for spatial databases or computational geometry, have been proposed as appropriate for indexing [144, 236, 36]. A new data structure was presented in [145] which minimizes accesses to secondary storage. A sort-join algorithm for joins on constrained variables is given in [36]. That paper also provides a query optimization methodology for conjunctive queries that can balance the cost of constraint manipulation against the cost of traditional database operations.

8. Concurrent Constraint Logic Programming

Concurrent programming languages are languages which allow the description of collections of processes which may interact with each other. In concurrent constraint logic programming (CCLP) languages, communication and synchronization are performed by asserting and testing for constraints. The operational semantics of these languages are quite similar to the top-down execution described in Section 5. However, the different context in which they are used results in a lesser importance of the corresponding logical semantics.

For this discussion we will consider only the flat ask-tell CCLP languages, which were defined in [213, 214] based on ideas from [173]. We further restrict our attention to languages with only committed-choice nondeterminism (sometimes called don't-care nondeterminism); more general languages will be discussed in Section 9. For more details of CCLP languages, see [218, 30].

Just as Prolog can be viewed as a kind of CLP language, obtained by a particular choice of constraint domain, so most concurrent logic languages can be viewed as concurrent CLP languages[15].

A program rule takes the form

$$h \leftarrow ask \ : \ tell \mid B$$

where h is an atom, B is a collection of atoms, and ask and $tell$ are constraints. Many treatments of concurrent constraint languages employ a language based on a process algebra involving ask and tell primitives [214], but we use the syntax above to emphasize the similarities to other CLP languages.

For the sake of brevity, we present a simpler transition system to describe the operational semantics than the transition system in Section 5. However implemented languages can make the same pragmatic compromises on testing consistency (and

[15] Concurrent Prolog [221] is not an ask-tell language, but [213] shows how it can be fit inside the CCLP framework.

implication) as reflected in that transition system. The states in this transition system have the form $\langle A, C\rangle$ where A is a collection of atoms and C is a collection of constraints. Any state can be an initial state. The transitions in the transition system are:

$$\langle A \cup a, C\rangle \rightarrow_r \langle A \cup B, C \cup (a = h) \cup ask \cup tell\rangle$$

if $h \leftarrow ask \ : \ tell \mid B$ is a rule of P renamed to new variables $\tilde{x}$, h and a have the same predicate symbol, $\mathcal{D} \models C \rightarrow \exists\tilde{x}\ a = h \wedge ask$ and $\mathcal{D} \models \tilde{\exists}\ C \wedge a = h \wedge ask \wedge tell$. Roughly speaking, a transition can occur with such a rule provided the accumulated constraints imply the ask constraint and do not contradict the tell constraint. Some languages use only the ask constraint for synchronization. It is shown in [29] that such languages are strictly less expressive than ask-tell languages.

An operational semantics such as the above is not completely faithful to a real execution of the language, since it is possible for two atoms to be rewritten simultaneously in an execution environment with concurrency. The above semantics only allows rewritings to be interleaved. A "true concurrency" semantics, based on graph-rewriting, is given in [191].

All ask-tell CCLP programs have the following *monotonicity* [216] or *stability* [96] property: If $\langle A, C\rangle \rightarrow_r \langle A', C'\rangle$ and $\mathcal{D} \models C'' \rightarrow C'$ then $\langle A, C''\rangle \rightarrow_r \langle A', C''\rangle$. This property provides for simple solutions to some problems in distributed computing related to reliability. When looked at in a more general framework [96], stability seems to be one advantage of CCLP languages over other languages; most programs in conventional languages for concurrency are not stable. It is interesting to note that a notion of global failure (as represented in Section 5 by the state $fail$) destroys stability. Of course, there are also pragmatic reasons for wanting to avoid this notion in a concurrent language. A framework which permits non-monotonic CCLP languages is discussed in [27].

A program is *determinate* if every reachable state is determinate, where a state is determinate if every selected atom gives rise to at most one $\rightarrow_r$ transition. Consequently, for every initial state, every fair derivation rewrites the same atoms with the same rules, or every derivation fails. Thus non-failed derivations by determinate programs from an initial state differ from each other only in the order of rewriting (and the renaming of rules). Substantial parts of many programs are determinate[16]. The interest in determinate programs arises from an elegant semantics for such programs based upon closure operators [217]. For every collection of atoms A, the semantics of A is given by the function $\mathcal{P}_A(C) = \exists_{-\tilde{x}}\ C'$ where $\langle A, C\rangle \rightarrow_r^* \langle A', C'\rangle$, $\tilde{x}$ is the free variables of $\langle A, C\rangle$, and $\langle A', C'\rangle$ is a final state. This semantics is extended in [217] to a compositional and fully abstract semantics of arbitrary programs. A semantics based on traces is given in [28].

For determinate programs we also have a clean application of the classical logical semantics of a program [173]. If $\langle A, C\rangle \rightarrow_r^* \langle A', C'\rangle$ then $P^*, \mathcal{D} \models A \wedge C \leftrightarrow \exists_{-\tilde{x}}\ A' \wedge C'$ where $\tilde{x}$ is the free variables of $\langle A, C\rangle$. In cases where execution can be

[16] For the programs we consider, determinate programs can be characterized syntactically by the following condition: for every pair of rules (renamed apart, except for identical heads) $h \leftarrow ask_1 \ : \ tell_1 \mid B_1$ and $h \leftarrow ask_2 \ : \ tell_2 \mid B_2$ in the program, we have $\mathcal{D} \models \neg(ask_1 \wedge ask_2 \wedge tell_1)$ or $\mathcal{D} \models \neg(ask_1 \wedge ask_2 \wedge tell_2)$. In languages where procedures can be hidden (as in many process algebra formulations) or there is a restriction on the initial states the class of determinate programs is larger, but is not so easily characterized.

guaranteed not to suspend any atom indefinitely, the soundness and completeness results for success and failure hold (see Section 6).

9. Linguistic Extensions

We discuss in this section some additional linguistic features for top-down CLP languages.

9.1. Shrinking the Computation Tree

The aim of $\rightarrow_i$ transitions is to extract as much information as is reasonable from the passive constraints, so that the branching of $\rightarrow_r$ transitions is reduced. There are several other techniques, used or proposed, for achieving this result.

In [202] it is suggested that information can also be extracted from the atoms in a state. The constraint extracted would be an approximation of the answers to the atom. This operation can be expressed by an additional transition rule.

$$\langle A \cup a, C, S\rangle \rightarrow_x \langle A \cup a, C, S \cup c\rangle$$

where $extract(a, C) = c$. Here *extract* is a function satisfying $P^*, \mathcal{D} \models (a \wedge C) \rightarrow c$. The evaluation of *extract*, performed at run-time, involves an abstract (or approximate) execution of $\langle a, C, \emptyset\rangle$. For example, if P defines p with the facts $p(1, 2)$ and $p(3, 4)$ then the constraint extracted by $extract(p(x, y), \emptyset)$ might be $y = x + 1$.

A more widespread technique is to modify the order in which atoms are selected. Most CLP systems employ the Prolog left-to-right computation rule. This improves the "programmability" by providing a predictable flow of control. However, when an appropriate flow of control is data-dependent or very complex (for example, in combinatorial search problems) greater flexibility is required.

One solution to this problem is to incorporate a data-dependent computation rule in the language. The Andorra principle [262] involves selecting determinate atoms, if possible. (A determinate atom is an atom which only gives rise to one $\rightarrow_{ris}$ transition.) A second approach is to allow the programmer to annotate parts of the program (atoms, predicates, clauses, ...) to provide a more flexible computation rule that is, nonetheless, programmed. This approach was pioneered in Prolog II [59] and MU-Prolog [195]. The automatic annotation of programs [194] brings this approach closer to the first. A third approach is to introduce constructs from concurrent logic programming into the language. There are basically two varieties of this approach: guarded rules and guarded atoms. The former introduces a committed-choice aspect into the language, whereas the latter is a variant of the second approach. All these approaches originated for conventional logic programs, but the ideas lift to constraint logic programs, and there are now several proposals based on these ideas [137, 234, 11, 113, 116].

One potential problem with using guarded rules is that the completeness of the operational semantics with respect to the logical semantics of the program can be lost. This incompleteness was shown to be avoided in ALPS [173] (modulo infinitely delayed atoms), but that work was heavily reliant on determinacy. Smolka [234] discusses a language of guarded rules which extends ALPS and a methodology for extending a predicate definition with new guarded rules such that completeness can be retained, even though execution would involve indeterminate committed-choice. The Andorra Kernel Language (AKL) [137] also combines the Andorra principle with guarded rules. There the interest is in providing a language which subsumes the expressive power of CCLP languages and CLP languages.

Guarded atoms and, more generally, guarded goals take the form $c \rightarrow G$ where c is a constraint[17] and G is a goal. G is available for execution only when c is implied by the current active constraints. We call c the *guard constraint* and G the *delayed goal*. Although the underlying mechanisms are very similar, guarded atoms and guarded rules differ substantially as linguistic features, since guarded atoms can be combined conjunctively whereas guards in guarded rules are combined disjunctively.

9.2. Complex Constraints

Several language constructs that can be said simply to be complex constraints have been added to CLP languages. We can classify them as follows: those which implement Boolean combinations of (generally simple) constraints and those which describe an ad hoc, often application-specific, relation. Falling into the first category are some implementations of constraint disjunction [116, 72] (sometimes called "constructive disjunction") and the cardinality operator [112]. Into the second category fall the *element* constraint [77], and the *cumulative* constraint of [2], among others. These constraints are already accounted for in the operational semantics of Section 5, since they can be considered passive constraints in $\mathcal{L}$. However, it also can be useful to view them as additions to a better-known constraint domain (indeed, this is how they arose).

The cardinality operator can be used to express any Boolean combination of constraints. A use of this combinator has the form $\#(L, [c_1, \ldots, c_n], U)$, where the c_i are constraints and L and U are variables. This use expresses that the number of constraints c_i that are true lies between the value of L and the value of U (lower and upper bound respectively). By constraining $L \geq 1$ the combinator represents the disjunction of the constraints; by constraining $U = 0$ the combinator represents the conjunction of the negations of the constraints. The cardinality combinator is implemented by testing whether the constraints are entailed by or inconsistent with the constraint store, and comparing the numbers of entailed and inconsistent constraints with the values of L and U. When L and U are not ground, the cardinality constraint can produce a constraint on these variables. (For example, after one constraint is found to be inconsistent U can be constrained by $U \leq n-1$.)

In constraint languages without disjunction, an intended disjunction $c_1(\tilde{x}) \vee c_2(\tilde{x})$ must be represented by a pair of clauses

$$
\begin{aligned}
p(\tilde{x}) &\leftarrow c_1(\tilde{x}) \\
p(\tilde{x}) &\leftarrow c_2(\tilde{x})
\end{aligned}
$$

In a simple CLP language this representation forces a choice to be made (between the two disjuncts). Constructive disjunction refers to the direct use of a disjunctive constraint without immediately making a choice. Instead an active constraint is computed which is a safe approximation to the disjunction in the context of the current constraint store C. In the constraint domain $\mathcal{FD}$, [116] suggests two possible

[17] We also permit the constraint $ground(x)$.

approximations, one based on approximating each constraint $C \wedge c_i$ using the domain of each variable and the other (less accurately) approximating each constraint using the interval constraints for each variable. The disjunctions of these approximations is easily expressed as an active constraint. For linear arithmetic [72] suggests the use of the convex hull of the regions defined by the two constraints as the approximation. Note that the constructive disjunction behavior could be obtained from the clauses for p using the methods of [202].

In the second category, we mention two constructs used with the finite domain solver of CHIP. $element(X, L, T)$ expresses that T is the X'th element in the list L. Operationally, it allows constraints on either the index X or element T of the list to be reflected by constraints on the other. For example, if X is constrained so that $X \in \{1, 3, 5\}$ then $element(X, [1, 1, 2, 3, 5, 8], T)$ can constrain T so that $T \in \{1, 2, 5\}$ and, similarly, if T is constrained so that $T \in \{1, 3, 5\}$ then X is constrained so that $X \in \{1, 2, 4, 5\}$. Declaratively, the *cumulative* constraint of [2] expresses a collection of linear inequalities on its arguments. Several problems that can be expressed as integer programming problems can be expressed with *cumulative*. Operationally, it behaves somewhat differently from the way CHIP would usually treat the inequalities.

9.3. User-Defined Constraints

Complex constraints are generally "built in" to the language. There are proposals to extend CLP languages to allow the user to define new constraints, together with inference rules specifying how the new constraints react with the constraint store.

A basic approach is to use guarded clauses. The new constraint predicate is defined with guarded clauses, where the guards specify the cases in which the constraint is to be simplified, and the body is an equivalent conjunction of constraints. Using $ground(x)$ (or a similar construct) as a guard constraint, it is straightforward to implement local propagation (i.e. propagation of ground values). We give an example of this use in Section 11.1, and [212] has other examples. Some more general forms of propagation can also be expressed with guarded clauses.

The work [93] can be seen as an extension of this method. The new constraints occur as predicates, and guarded rules (called constraint simplification rules) are used to simplify the new constraints. However, the guarded rules may have two (or more) atoms in the head. Execution matches the head with a collection of constraint atoms in the goal and reduces to an equivalent conjunction of constraints. This method appears able to express more powerful solving methods than the guarded clauses. For example, transitivity of the user-defined constraint *leq* can be specified by the rule

```
leq(X, Y), leq(Y, Z) ==> true | leq(X, Z).
```

whereas it is not clear how to express this in a one-atom-per-head guarded clause. A drawback of having multiple atoms, however, is inefficiency. In particular, it is not clear whether constraint simplification rules can produce incremental (in the sense defined in Section 10.1) constraint solvers except in simple cases.

A different approach [115] proposes the introduction of "indexical" terms which refer to aspects of the *state* of the constraint solver (thus providing a limited form of reflection)[18]. Constraints containing these terms are called indexical constraints, and from these indexical constraints user-defined constraints are built. Specifically, [115] discusses a language over finite domains which can access the current domain and upper and lower bounds on the value of a variable using the indexical terms $dom(X)$, $max(X)$ and $min(X)$ respectively. Indexical constraints have an operational semantics: each constraint defines a method of propagation. For example, the constraint $Y\ in\ 0..max(X)$ continually enforces the upper bound of Y to be less than or equal to the upper bound of X. This same behavior can be obtained in a finite domain solver with the constraint $Y \leq X$, but the advantage of indexical constraints is that there is greater control over propagation: with $Y \leq X$ we also propagate changes in the lower bound of Y to X, whereas we can avoid this with indexical constraints. A discussion of an implementation of indexical constraints is given in [75]. (One application of this work is a constraint solver for Boolean constraints [56]; we describe this application in Section 13.5.)

9.4. Negation

Treatments of negation in logic programming lift readily to constraint logic programming, with only minor adjustments necessary. Indeed many of the semantics for programs with negation are essentially propositional, being based upon the collection of ground instances of program rules. The perfect model [203, 14, 100], well-founded model [101], stable model [102] and Fitting fixedpoint semantics [90], to name but a few, fall into this category. The grounding of variables in CLP rules by all elements of the domain (i.e. by all terms in $\mathcal{L}^*$) and the deletion of all grounded rules whose constraints evaluate to false produces the desired propositional rules (see, for example, [176]).

Other declarative semantics, based on Clark's completion P^* of the program, also extend to CLP[19]. The counterpart of $comp(P)$ [55, 168] is $\mathcal{T}, P^*$, where $\mathcal{T}$ is satisfaction complete. Interestingly, it is necessary to consider the complete theory $\mathcal{T}$ of the domain if the equivalence of three-valued logical consequences of $\mathcal{T}, P^*$ and consequences of finite iterations of Fitting's Φ operator (as shown by Kunen [153]), continues to hold for CLP programs [244].

SLDNF-resolution and its variants are also relatively untouched by the lifting to CLP programs, although, of course, they must use a consistency test instead of unification. The other main modification is that groundness must be replaced by the concept of a variable being determined by the current constraints (see Section 2). For example, a safe computation rule [168] may select a non-ground negated atom provided all the variables in the atom are determined by the current collection of constraints. Similarly, the definition of an allowed rule [168] for a CLP program requires that every variable either appear in a positive literal in the body or be determined by the constraints in the body. With these modifications, various soundness and completeness results for SLDNF-resolution and $comp(P)$ extend easily to ideal CLP systems. An alternative implementation of negation, constructive negation [51], has been expanded and applied to CLP programs by Stuckey [244], who gave the first completeness result for this method.

[18] This approach has been called a "glass-box" approach.

[19] For example, the extension to allow arbitrary first-order formulas in the bodies of rules [169].

9.5. Preferred Solutions

Often it is desirable to express an ordering (or preference) on solutions to a goal. This can provide a basis for computing only the "best" solutions to the query. One approach is to adapt the approach of mathematical programming (operations research) and employ an objective function [107, 178]. An optimization primitive is added to the language to compute the optimal value of the objective function[20].

CHIP and cc($\mathcal{FD}$) have such primitives, but they have a non-logical behavior. Two recent papers [86, 183] discuss optimization primitives based upon the following logical characterization:

m is the minimum value of $f(\tilde{x})$ such that $G(\tilde{x})$ holds iff

$$\exists \tilde{x}\ (G(\tilde{x}) \wedge f(\tilde{x}) = m) \wedge \neg \exists \tilde{y}\ (G(\tilde{y}) \wedge f(\tilde{y}) < m)$$

Optimization primitives can be implemented by a branch and bound approach, pruning the computation tree of G based on the current minimum. A similar behavior can be obtained through constructive negation, using the above logical formulation [86, 183], although a special-purpose implementation is more efficient. [183] gives a completeness result for such an implementation, based on Kunen's semantics for negation.

A second approach is to admit constraints which are not required to be satisfied by a solution, but express a preference for solutions which do satisfy them. Such constraints are sometimes called *soft* constraints. The most developed use of this approach is in hierarchical constraint logic programming (HCLP) [33, 263]. In HCLP, soft constraints have different strengths and the constraints accumulated during a derivation form a constraint hierarchy based on these strengths. There are many possible ways to compare solutions using these constraint hierarchies [33, 178, 263], different methods being suitable for different problems. The hierarchy dictates that any number of weak constraints can be over-ruled by a stronger constraint. Thus, for example, default behavior can be expressed in a program by weak constraints, which will be over-ruled by stronger constraints when non-default behavior is required. The restriction to best solutions of a constraint hierarchy can be viewed as a form of circumscription [219].

Each of the above approaches has some programming advantages over the other, in certain applications, but both have problems as general-purpose methods. While the first approach works well when there is a natural choice of objective function suggested by the problem, in general there is no natural choice. The second approach provides a higher-level expression of preference but it cannot be so easily "fine-tuned" and it can produce an exponential number of best answers if not used carefully. The approaches have the advantages and disadvantages of explicit (respectively, implicit) representations of preference. In the first approach, it can be difficult to reflect intended preferences. In the second approach it is easier to reflect intended preferences, but harder to detect inconsistency in these preferences. It is also possible to "weight" soft constraints, which provides a combination of both approaches.

[20] We discuss only minimization; maximization is similar.

Part II
Implementation Issues

The main innovation required to implement a CLP system is clearly in the manipulation of constraints. Thus the main focus in this part of the survey is on constraint solver operations, described in the section below. The next section then considers the problem of extending the LP inference engine to deal with constraints. Here the discussion is not tied down to a particular constraint domain.

It is important to note that the algorithms and data structures in this part are presented in view of their use in top-down systems and, in particular, systems with backtracking. At the present, there is little experience in implementing bottom-up CLP systems, and so we do not discuss them here. However, some of the algorithms we discuss can be used, perhaps with modification, in bottom-up systems.

10. Algorithms for Constraint Solving

In view of the operational semantics presented in part I, there are several operations involving constraints to be implemented. These include: a satisfiability test, to implement *consistent* and *infer*; an entailment test, to implement guarded goals; and the projection of the constraint store onto a set of variables, to compute the answer constraint from a final state. The constraint solver must also be able to undo the effects of adding constraints when the inference engine backtracks. In this section we discuss the core efficiency issues in the implementation of these operations.

10.1. Incrementality

According to the folklore of CLP, algorithms for CLP implementations must be *incremental* in order to be practical. However this prescription is not totally satisfactory, since the term incremental can be used in two different senses. On one hand, incrementality is used to refer to the *nature* of the algorithm. That is, an algorithm is incremental if it accumulates an internal state and a new input is processed in combination with the internal state. Such algorithms are sometimes called *on-line* algorithms. On the other hand, incrementality is sometimes used to refer to the *performance* of the algorithm. This section serves to clarify the latter notion of incrementality as a prelude to our discussion of algorithms in the following subsections. We do not, however, offer a formal definition of incrementality.

We begin by abstracting away the inference engine from the operational semantics, to leave simply the constraint solver and its operations. We consider the state of the constraint solver to consist of the constraint store C, a collection of constraints G that are to be entailed, and some backtrack points. In the initial state, denoted by $\emptyset$, there are no constraints nor backtrack points. The constraint solver reacts to a sequence of operations, and results in (a) a new state, and (b) a response.

Recall that the operations in CLP languages are:

- augment C with c to obtain a new store, determine whether the new store is satisfiable, and if so, determine which constraints in G are implied by the new store;

- add a new constraint to G;
- set a backtrack point (and associate with it the current state of the system);
- backtrack to the previous backtrack point (i.e. return the state of the system to that associated with the backtrack point);
- project C onto a fixed set of variables.

Only the first and last of these operations can produce a response from the constraint solver.
Consider the application of a sequence of operations $o_1, \ldots, o_k$ on a state Δ; denote the updated state by $\mathcal{F}(\Delta, o_1 \ldots o_k)$, and the sequence of responses to the operations by $\mathcal{G}(o_1 \ldots o_k)$. In what follows we shall be concerned with the average cost of computing $\mathcal{F}$ and $\mathcal{G}$. Using standard definitions, this cost is parameterized by the distribution of (sequences of) operations (see, for example, [255]). We use average cost assuming the true distribution, the distribution that reflects what occurs in practice. Even though this distribution is almost always not known, we often have some hypotheses about it. For example, one can identify typical and often occurring operation sequences and hence can approximate the true distribution accordingly. The informal definitions below therefore are intended to be a guide, as opposed to a formal tool for cost analysis.

For an expression $exp(\tilde{o})$ denoting a function of $\tilde{o}$, define $AV[exp(\tilde{o})]$ to be the average value of $exp(\tilde{o})$, over all sequences of operations $\tilde{o}$. Note that the definition of average here is also dependent on the distribution of the $\tilde{o}$. For example, let $cost(\tilde{o})$ denote the cost of computing $\mathcal{F}(\emptyset, \tilde{o})$ by some algorithm, for <u>each</u> fixed sequence $\tilde{o}$. Then $AV[cost(\tilde{o})]$ denotes the average cost of computing $\mathcal{F}(\emptyset, \tilde{o})$ over <u>all</u> $\tilde{o}$.

Let Δ be shorthand for $\mathcal{F}(\emptyset, o_1 \ldots o_{k-1})$. Let A denote an algorithm which applies a sequence of operations on the initial state, giving the same response as does the constraint solver, but not necessarily computing the new state. That is, A is the batch (or off-line) version of our constraint solver. In what follows we discuss what it means for an algorithm to be incremental relative to some algorithm A. Intuitively A represents the best available batch algorithm for the operations.

At one extreme, we consider that an algorithm for $\mathcal{F}$ and $\mathcal{G}$ is "non-incremental" relative to A if the average cost of applying an extra operation o_k to Δ is no better than the cost of the straightforward approach using A on $o_1 \ldots o_k$. We express this as

$$AV\left[cost(\Delta, o_k)\right] \quad \geq \quad AV\left[cost_A(o_1 \ldots o_k)\right].$$

At the other extreme, we consider that an algorithm for $\mathcal{F}$ and $\mathcal{G}$ is "perfectly incremental", relative to A, if its cost is no worse than that of A. In other words, no cost is incurred for the incremental nature of the algorithm. We express this as

$$AV\left[cost(\emptyset, o_1 \ldots o_{k-1}) + cost(\Delta, o_k)\right] \quad \leq \quad AV\left[cost_A(o_1 \ldots o_k)\right].$$

In general, any algorithm lies somewhere in between these two extremes. For example, it will not be perfectly incremental as indicated by the cost formula above, but instead we have

$$AV\left[cost(\emptyset, o_1 \ldots o_{k-1}) + cost(\Delta, o_k)\right] \quad = \quad AV\left[cost_A(o_1 \ldots o_k)\right] + extra\_cost(o_1 \ldots o_k)$$

where the additional term $extra\_cost(o_1 \ldots o_k)$ denotes the extra cost incurred by the on-line algorithm over the best batch algorithm. Therefore, one possible "definition" of an incremental algorithm, good enough for use in a CLP system, is simply that its *extra_cost* factor is negligible.

In what follows, we shall tacitly bear in mind this expression to obtain a rough definition of incrementality[21]. Although we have defined incrementality for a collection of operations, we will review the operations individually, and discuss incrementality in isolation. This can sometimes be an oversimplification; for example, [180] has shown that the standard unification problem does not remain linear when backtracking is considered. In general, however, it is simply too complex, in a survey article, to do otherwise.

10.2. Satisfiability (Non-incremental)

We consider first the basic problem of determining satisfiability of constraints independent of the requirement for incrementality. As we will see in the brief tour below of our sample domains, the dominant criteria used by system implementers is not the worst-case time complexity of the algorithm.

For the domain $\mathcal{FT}$, linear time algorithms are known [199], and for $\mathcal{RT}$, the best known algorithms are almost linear time [126]. Even so, most Prolog systems implement an algorithm for the latter[22] because the *best*-case complexity of unification in $\mathcal{FT}$ is also linear, whereas it is often the case that unification in $\mathcal{RT}$ can be done without inspecting all parts of the terms being unified. Hence in practice Prolog systems are really implementations of CLP($\mathcal{RT}$) rather than CLP($\mathcal{FT}$). In fact, many Prolog systems choose to use straightforward algorithms which are slower, in the worst case, than these almost linear time algorithms. The reason for this choice (of algorithms which are quadratic time or slower in the worst case) is the belief that these algorithms are faster on average [13].

For the arithmetic domain of $\Re_{LinEqn}$, the most straightforward algorithm is based on Gaussian elimination, and this has quadratic worst-case complexity. For the more general domain $\Re_{Lin}$, polynomial time algorithms are also known [141], but these algorithms are not used in practical CLP systems. Instead, the Simplex algorithm (see eg. [54]), despite its exponential time worst case complexity [148], is used as a basis for the algorithm. However, since the Simplex algorithm works over non-negative numbers and non-strict inequalities, it must be extended for use in CLP systems. While such an extension is straightforward in principle, implementations must be carefully engineered to avoid significant overhead. The main differences between the Simplex-based solvers in CLP systems is in the specific realization of this basic algorithm. For example, the CLP($\mathcal{R}$) system uses a floating-point representation of numbers, whereas the solvers of CHIP and Prolog III use exact precision rational number arithmetic. As another example, in the CLP($\mathcal{R}$) system a major design decision was to separately deal with equations and

[21] There are similar notions found in the (non-CLP) literature; see the bibliography [207].

[22] This is often realized simply by omitting the "occur-check" operation from a standard unification algorithm for $\mathcal{FT}$. Some Prolog systems perform such an omission naively, and thus obtain an incomplete algorithm which may not terminate in certain cases. These cases are considered pathological and hence are ignored. Other systems guarantee termination at slightly higher cost, but enjoy the new feature of cyclic data structures.

inequalities, enjoying a faster (Gaussian-elimination based) algorithm for equations, but enduring a cost for communication between the two kinds of algorithms [133]. Some elements of the CHIP solver are described in [114]. Disequality constraints can be handled using entailment of the corresponding equation (discussed in Section 10.4) since an independence of negative constraints holds [162].

For the domain of word equations $\mathcal{WE}$, an algorithm is known [179] but no efficient algorithm is known. In fact, the general problem, though easily provable to be NP-hard, is not known to be in NP. The most efficient algorithm known still has the basic structure of the Makanin algorithm but uses a far better bound for termination [150]. Systems using word equations, Prolog III for example, thus resort to partial constraint solving using a standard delay technique on the lengths of word variables. Rajasekar's "string logic programs" [204] also uses a partial solution of word equations. First, solutions are found for equations over the lengths of the word variables appearing in the constraint; only then is the word equation solved.

As with word equations, the satisfiability problem in finite domains such as $\mathcal{FD}$ is almost always NP-hard. Partial constraint solving is once again required, and here is a typical approach. Attach to each variable x a data structure representing $dom(x)$, its current possible values[23]. Clearly $dom(x)$ should be a superset of the projection space w.r.t. x. Define $min(x)$ and $max(x)$ to be the smallest and largest numbers in $dom(x)$ respectively. Now, assume that every constraint is written in so that each inequality is of the form $x < y$ or $x \leq y$, each disequality is of the form $x \neq y$, and each equation is of the form $x = n$, $x = y$, $x = y + z$, where x, y, z are variables and n a number. Clearly every constraint in $\mathcal{FD}$ can be rewritten into a conjunction of these constraints.

The algorithm considers one constraint at a time and has two main phases. First, it performs an action which is determined by the form of the constraint: (a) for constraints $x \leq y$, ensure that $min(x) \leq max(y)$ by modifying $dom(x)$ and/or $dom(y)$ appropriately[24]. (b) for $x < y$, ensure that $min(x) < max(y)$; (c) for $x \neq y$, consider three subcases: if $dom(x) \cap dom(y) = \emptyset$ then the constraint reduces to *true*; otherwise, if $dom(x) = \{n\}$, then remove n from $dom(y)$ (and similarly for the case when $dom(y)$ is a singleton[25]); otherwise, nothing more need be done; (d) for $x = n$, simple make $dom(x) = \{n\}$; (e) for $x = y$, make $dom(x) = dom(y) = dom(x) \cap dom(y)$; (f) for $x = y + z$, ensure that $max(x) \geq min(y) + min(z)$ and $min(x) \leq max(y) + max(z)$. If at any time during steps (a) through (f) the domain of a variable becomes empty, then unsatisfiability has been detected. The second phase of this algorithm is that for each x such that $dom(x)$ that is changed by some action in steps (a) through (f), all constraints (but the current one that gave rise to this action) that contain x are re-considered for further action. Termination is, of course, assured simply because the domains are finite.

In the domain of Boolean algebra $\mathcal{BOOL}$, there are a variety of techniques for testing satisfiability. Since the problem is NP-complete, none of these can be expected to perform efficiently over all constraints. An early technique, pioneered by

[23] The choice of such a data structure should depend on the size of the finite domains. For example, with small domains a characteristic vector is appropriate.

[24] In this case, simply remove from $dom(x)$ all elements bigger than $max(y)$, and remove from $dom(y)$ all elements smaller than $min(x)$. We omit the details of similar operations in the following discussion.

[25] If both are singletons, clearly the constraint reduces to either *true* or *false*.

Davis and Putnam, is based upon variable elimination. The essential idea reduces a normal form representation into two smaller problems, each with one less variable. Binary decision diagrams [38] provide an efficient representation. One of the two Boolean solvers of CHIP, for example, uses variable elimination and these diagrams. A related technique is based on enumeration and propagation. The constraints are expressed as a conjunction of simple constraints and then local propagation simplifies the conjunction after each enumeration step. See [56], for example. The method used in Prolog III [21] is a modification of SL-resolution whose main element is the elimination of redundant expressions. Another technique comes from Operations Research. Here the boolean formula is restated in arithmetic form, with variables constrained to be 0 and 1. Then standard techniques for integer programming, for example cutting-planes, can be used. See [53] for a further discussion of this technique. This technique has not been used in CLP systems. A more recent development is the adaptation of Buchberger's Groebner basis algorithm to Boolean algebras [210], which is used in CAL. Finally, there is the class of algorithms which perform boolean unification; see the survey [185] for example. Here satisfiability testing is only part of the problem addressed, and hence we will discuss these algorithms in the next section.

The satisfiability problem for feature trees is essentially the same as the satisfiability problem for rational trees, provided that the number of features that may occur is bounded by a known constant [12]. (Generally this bounding constant can be determined at compile-time.) Two different sort constraints on the same variable clash in the same way that different function symbols on terms in $\mathcal{RT}$ clash. An equation between two feature tree variables (of the same sort) induces equations between all the subtrees determined by the features of the variables, in the same way as occurs in $\mathcal{RT}$. The main difference is that some sorts or features may be undetermined (roughly, unbound) in $\mathcal{FEAT}$.

10.3. Satisfiability (Incremental)

As alluded to above, it is crucial that the algorithm that determines the satisfiability of a tentatively new constraint store be incremental. For example, a linear-time algorithm for a satisfiability problem is often as good as one can get. Consider a sequence of constraints $c_1, \cdots, c_k$ of approximately equal size N. A naive application of this linear-time algorithm to decide c_1, then $c_1 \wedge c_2$, $\cdots$, and finally $c_1 \wedge \cdots \wedge c_k$ could incur a cost proportional to Nk^2, on average. In contrast, a perfectly incremental algorithm as discussed in Section 10.1 has a cost of $O(Nk)$, on average.

In practice, most algorithms represent constraints in some kind of *solved form*, a format in which the satisfiability of the constraints is evident. Thus the satisfiability problem is essentially that of reducibility into solved form. For example, standard unification algorithms for $\mathcal{FT}$ represent constraints by (one variant of) its mgu, that is, in the form $x_1 = t_1(\tilde{y}), \cdots, x_n = t_n(\tilde{y})$ where each $t_i(\tilde{y})$ denotes a term structure containing variables from $\tilde{y}$, and no variable x_i appears in $\tilde{y}$. Similarly, linear equations in $\Re_{LinEqn}$ are often represented in parametric form $x_1 = le_1(\tilde{y}), \cdots, x_n = le_n(\tilde{y})$ where each $le_i(\tilde{y})$ denotes a linear expression containing variables from $\tilde{y}$, and no variable x_i appears in $\tilde{y}$. In both these examples, call the x_i *eliminable* variables, and the y_i *parametric* variables. For linear inequalities in $\Re_{Lin}$, the Simplex algorithm represents the constraints in an $n \times m$ matrix form $A\tilde{x} = B$ where A contains an $n \times n$ identity submatrix, defining the basis variables, and all numbers in the column vector B are nonnegative. For domains based on a unitary equality theory [224], the standard representation is the mgu, as in the case of $\mathcal{FT}$ (which corresponds to the most elementary equality theory). Word equations over $\mathcal{WE}$, however, are associated with an infinitary theory, and thus a unification algorithm for these equations [127] may not terminate. A solved form for word equations, or any closed form solution for that matter, is not known.

The first two kinds of solved form above are also examples of *solution forms*, that is, a format in which the set of all solutions of the constraints is evident. Here, any instance of the variables $\tilde{y}$ determines values for $\tilde{x}$ and thus gives one solution. The set of all such instances gives the set of all solutions. The Simplex format, however, is not in solution form: each choice of basis variables depicts just one particular solution.

An important property of solution forms (and sometimes of just solved forms) is that they define a convenient representation of the *projection* of the solution space with respect to any set of variables. More specifically, each variable can be equated with a substitution expression containing only parametric variables, that is, variables whose projections are the entire space. This property, in turn, aids incrementality as we now show via our sample domains.

In each of the following examples, let C be a (satisfiable) constraint in solved form and let c be the new constraint at hand. For $\mathcal{FT}$, the substitution expression for a variable x is simply x if x is not eliminable; otherwise it is the expression equated to x in the solved form C. This mapping is generalized to terms in the obvious way. Similarly we can define a mapping of linear expressions by replacing the eliminable variables therein with their substitution expressions, and then collecting like terms. For the domain $\Re_{Lin}$, in which case C is in Simplex form, the substitution expression for a variable x is simply x if x is not basic; otherwise it is the expression obtain by writing the (unique) equation in C containing x with x as the subject. Once again, this mapping can be generalized to any linear expression in an obvious way. In summary, a solution form defines a mapping θ which can be used to map any expression t into an equivalent form $t\theta$ which is free of eliminable variables.

The basic step of a satisfiability algorithm using a solution form is essentially this.

Algorithm 10.1. Given C, (a) Replace the newly considered constraint c by $c\theta$ where θ is the substitution defined by C. (b) Then write $c\theta$ into equations of the form $x = ...$, and this involves choosing the x and rearranging terms. Unsatisfiability is detected at this stage. (c) If the previous step succeeds, use the new equations to substitute out all occurrences of x in C. (d) Finally, simply add the new equations to C, to obtain a solution form for $C \wedge c$.
Note that the nonappearance of eliminable variables in substitution expressions is needed in (b), to ensure that the new equations themselves are in solved form, and in (c), to ensure that C augmented with the new equations remains in solution form.

The belief that this methodology leads to an incremental algorithm is based upon believing that the cost of dealing with c is more closely related to the size of c (which is small on average) than that of C (which is very large on average). This, in turn, is based upon believing that

- the substitution expressions for the eliminable variables in c, which largely determine the size of $c\theta$, often have a size that is independent of the size of C, and

- the number of occurrences of the new eliminable variable x in C, which largely determines the cost of substituting out x in C, is small in comparison to the size of C.

The domain $\mathcal{FT}$ provides a particularly good example of a solved form for which the basic algorithm 10.1 is incremental. Consider a standard implementation in which there is only one location for each variable, and all references to x are implemented by pointers. Given C in solved form, and given a new constraint c, there is really nothing to do to obtain $c\theta$ since the eliminable (or in this case, bound) variables in c are already pointers to their substitution expressions. Now if $c\theta$ is satisfiable and we obtain the new equations $x = ...$, then just one pointer setting of x to its substitution expression is required, and we are done. In other words, the cost of this pointer-based algorithm is focussed on determining the satisfiability of $c\theta$ and extracting the new equations; in contrast, step (c) of global substitution using the new equations incurs no cost.

For $\Re_{LinEqn}$, the size of $c\theta$ can be large, even though the finally obtain equations may not be. For example, if C contained just $x_1 = u-v$, $x_2 = v-w$, $x_3 = w-u$, and c were $y = x_1 + x_2 + x_3$, then $c\theta$ is as big as C. Upon rearrangement, however, the finally obtained new equation is simply $y = 0$. Next, the substitution phase using the new equation also can enlarge the equations in C (even if temporarily), and rearrangement is needed in each equation substituted upon. In general, however, the beliefs above hold in practice, and the algorithm behaves incrementally.

We next consider the domain $\mathcal{RT}$ whose universally used solved form (due to [60]) is like that of $\mathcal{FT}$ with one important change: constraints are represented in the form $x_1 = t_1, \cdots, x_n = t_n$ where each t_i is an *arbitrary* term structure. Thus this solved form differs from that of $\mathcal{FT}$ in that the t_i can contain the variables x_j, and hence algorithm 1 is not directly applicable. It is easy to show that a constraint is satisfiable iff it has such a solved form, and further, the solved form is fairly explicit representation of the set of all solutions (though not as explicit as the solution forms for $\mathcal{FT}$ or $\Re_{LinEqn}$). A straightforward satisfiability algorithm [60] is roughly as follows. Let x stand for a variable, and s and t stand for non-variable terms. Now perform the following rewrite rules until none are applicable. (a) discard each $x = x$; (b) for any $x = y$, replace x by y throughout; (c) replace $t = x$ by $x = t$; (d) replace $f(s_1, \cdots, s_n) = f(t_1, \cdots, t_n)$, $n \geq 0$, by n equations $s_i = t_i$, $1 \leq i \leq n$; (e) replace $f(...) = g(...)$ by *false* (and thus the entire collection of constraints is unsatisfiable); (f) replace every pair of equations $x = t_1, x = t_2$, and say t_1 is not bigger than t_2, by $x = t_1, t_1 = t_2$. Termination needs to be argued, but we will leave the details to [60].

We now discuss algorithms which do not fit exactly with Algorithm 1, but which employ a solved form. Consider first the Simplex algorithm for the domain $\Re_{Lin}$. The basic step of one pivoting operation within this algorithm is essentially the same as Algorithm 1. The arguments for incrementality for Algorithm 1 thus apply. The main difference from Algorithm 1 is that, in general, several pivoting operations are required to produce the final solved form. However, empirical evidence from CLP systems has shown that often the number of pivoting operations is small [133].

In the Boolean domain, Boolean unification algorithms [185], conform to the structure of Algorithm 1. One unification algorithm is essentially due to Boole, and we borrow the following presentation from [109]. Without loss of generality, assume the constraints are of the form $t(x_1, \cdots, x_n) = 0$ where the x_i are the variables in t. Assuming $n \geq 2$, rewrite $t = 0$ into the form

$$g(x_1, \cdots, x_{n-1}) \wedge x_n \oplus h(x_1, \cdots, x_{n-1}) = 0$$

so that the problem for $t = 0$ can be reduced to that of

$$\neg g(x_1, \cdots, x_{n-1}) \wedge h(x_1, \cdots, x_{n-1}) = 0$$

which contains one less variable. If this equation is satisfiable, then the "assignment"

$$x_n = h(x_1, \cdots, x_{n-1}) \oplus \neg g(x_1, \cdots, x_{n-1}) \wedge y_n$$

where y_n is a new variable, describes all the possible solutions for x_n. This reduction clearly can be repeatedly applied until we are left with the straightforward problem of deciding the satisfiability of equations of the form $t \wedge x \oplus u = 0$ where t and u are ground. The unifier desired is given simply by collecting (and substituting out all assigned variables in) the assignments, such as that for x_n above.

The key efficiency problem here is, of course, that the variable elimination process gives rise to larger expressions, an increase which is exponential in the number of eliminated variables, in the worst case. So even though this algorithm satisfies the structure of Algorithm 1, it does not satisfy our assumption about the size of expressions obtained after substitution, and hence our general argument for incrementality does not apply here. Despite this, and the fact that Boole's work dates far back, this method is still used, for example in CHIP [45].

Another unification algorithm is due to Löwenhein, and we adapt the presentation of [185] here. Let $f(x_1, \cdots, x_n) = 0$ be the equation considered. Let $\tilde{a}$ denote a solution. The unifier is then simply given by

$$x_i = y_i \vee f(\tilde{y}) \wedge (y_i \vee a_i), 1 \leq i \leq n$$

where the y_i are new variables. The basic efficiency problem is of course to determine $\tilde{a}$. The obtained unifiers are only slightly larger than f, in contrast to Boole's method. Thus Löwenhein's method provides a way of extending a constructive satisfiability test into a satisfiability test which has an incremental form. However, this method is not, to our knowledge, used in CLP languages.

Other algorithms for testing the satisfiability of Boolean constraints are considerably different from Algorithm 1. The Groebner Basis algorithm produces a basis for the space of Boolean constraints implied by the constraint store. It is a solved form but not a solution form. The remaining algorithms mentioned in the previous subsection do not have a solved form. The algorithm used in Prolog III retains the set of literals implied to be true by the constraints, but the normal form does not guarantee solvability: that must be tested beforehand. Enumeration algorithms have the same behavior: they exhibit a solution, and may retain some further information, but they do not compute a solved form.

In the domain of feature trees $\mathcal{FEAT}$ equations occur only between variables. Thus Algorithm 1 does not address the whole problem. Existing algorithms [12, 235] employ a solved form in which all implied equations between variables are explicit and there are no clashes of sort. Such solved forms are, in fact, solution forms. The implied equations are found by congruence closure, treating the features as (partial) functions, analogously to rule (d) in the algorithm for $\mathcal{RT}$.

In summary for this subsection, an important property for algorithms to decide satisfiability is that they have good average case behavior. More important, and even crucially so, is that the algorithm is incremental. Toward this goal, a common technique is to use a solved form representation for satisfiable constraints.

10.4. Entailment

Given satisfiable C, guard constraints G such that no constraint therein is entailed by C, and a new constraint c, the problem at hand is to determine the subset G_1 of G of constraints entailed by $C \wedge c$. We will also consider the problem of detecting groundness which is not, strictly speaking, an entailment problem. However, it is essentially the same as the problem of detecting groundness to a specific value, which is an entailment problem. In what follows the distinction is unimportant.

We next present a rule-of-thumb to determine whether an entailment algorithm is incremental in the sense discussed earlier. The important factor is not the number of constraints entailed after a change in the store, but instead, the number of constraints *not* entailed. That is, the algorithm must be able to ignore the latter constraints so that the costs incurred depend only on the number of entailed constraints, as opposed to the total number of guard constraints. As in the case of incremental satisfiability, the property of incremental entailment is a crucial one for the implementation of practical CLP systems.

We now briefly discuss modifications to some of the previously discussed algorithms for satisfiability, which provide for incremental entailment.

Consider the domain $\mathcal{FT}$ and suppose G contains only guard constraints of the form $x = t$ where t is some ground term[26]. Add to a standard implementation of a unification algorithm an index structure mapping variables x to just those guard constraints in G which involve x. (See [46] for a detailed description.) Now add to the process of constructing a solved form a check for groundness when variables are bound (and this is easily detectable). This gives rise to an incremental algorithm because the only guard constraints that are inspected are those $x = t$ for which x has just become ground, and not the entire collection G.

Just as with satisfiability, testing entailment is essentially the same over the domains $\mathcal{RT}$ and $\mathcal{FEAT}$. Four recent works have addressed this problem, all in the context of a CLP system, but with slightly differing constraint domains. We will discuss them all in terms of $\mathcal{RT}$. With some modifications, these works can also apply to $\mathcal{FT}$.

In [235, 12] a theoretical foundation is built. [235] then proposes a concrete algorithm, very roughly as follows: the to-be-entailed constraint c is added to the constraint store C. The satisfiability tester has the capability of detecting whether c is entailed by or inconsistent with C. If neither is detected then c', essentially a simplified form of c, is stored and the effect of adding c to C is undone. Every time a constraint is added to C that affects c' this entailment test is repeated (with c' instead of c).

[26] As mentioned above, this discussion will essentially apply to guard constraints of the form *ground*(x).

The algorithm of [201] has some similarities to the previous algorithm, but avoids the necessity of undoing operations. Instead, operations that might affect C are delayed and/or performed on a special data-structure separate from C. Strong incrementality is claimed: if we replace average-case complexity by worst-case complexity, the algorithm satisfies our criterion for perfect incrementality.

[205] goes beyond the problem of entailing equations to give an algorithm for entailment when both equations and disequations ($\neq$) are considered constraints. This algorithm has a rather different basis than those discussed above; it involves memoization of pairs of terms (entailments and disequations) and the use of a reduction of disequation entailment to several equation entailments.

For $\Re_{LinEqn}$, and let G contain arbitrary equations e. Add to the algorithm which constructs the solved form a representation of each such equation e in which all eliminable variables are substituted out. Note, however, that even though these equations are stored with the other equations in the constraint store, they are considered as a distinct collection, and they play no direct role in the question of satisfiability of the current store. For example, a constraint store containing $x = z+3, y = z+2$ would cause the guard equation $y+z = 4$ to be represented as $z = 1$. It is easy to show that a guard equation e is entailed iff its representation reduces to the trivial form $0 = 0$, and similarly, the equation is refuted if its representation is of the form $0 = n$ where n is a nonzero number. (In our example, the guard equation is entailed or refuted just in case z becomes ground.) In order to have incrementality we must argue that the substitution operation is often applied only to very few of the guard constraints. This is tantamount to the second assumption made to argue the incrementality of Algorithm 1. Hence we believe our algorithm is incremental.

We move now to the domain $\Re_{Lin}$, but allow only equations in the guard constraints G. Here we can proceed as in the above discussion for $\Re_{LinEqn}$ to obtain an incremental algorithm, but we will have the further requirement that the constraint store contains all implicit equalities[27] explicitly represented as equations. It is then still easy to show that the entailment of a guard equation e need be checked only when the representation of e is trivial. The argument for incrementality given above for $\Re_{LinEqn}$ essentially holds here, provided that the cost of computing implicit equalities is sufficiently low.

There are two main works on the detection of implicit equalities in CLP systems over $\Re_{Lin}$. In [243], the existence of implicit equalities is detected by the appearance of an equation of a special kind in the Simplex tableau at the end of the satisfiability checking process. Such an equation indicates some of the implicit equalities, but more pivoting (which, in turn, can give rise to more special equations) is generally required to find all of them. An important characteristic of this algorithm is that the extra cost incurred is proportional to the number of implicit equalities. This method is used in CLP($\mathcal{R}$) and Prolog III. CHIP uses a method based on [114]. In this method, a solved form which is more restrictive than the usual Simplex solved form is used. An equation in this form does not contribute to any implicit equality, and a whole tableau in this solved form implies that there are no implicit equalities. The basic idea is then to maintain the constraints in the solved form and when a new constraint is encountered, the search for implicit equalities can be first limited

[27] These are equalities which are entailed by the store because of the presence of inequalities. For example, the constraint store $x + y \leq 3, x + y \geq 3$ entails the implicit equality $x + y = 3$.

to variables in the new constraint. One added feature of this solved form is that directly accomodates strict inequalities and disequations.

Next consider still the domain $\Re_{Lin}$, but now allow inequalities to be in G. Here it is not clear how to represent a guard inequality, say $x \geq 5$, in such a way that its entailment or refutation is detectable by some simple format in its representation. Using the Simplex tableau format as a solved form as discussed above, and using the same intuition as in the discussion of guard equations, we could substitute out x in $x \geq 5$ in case x is basic. However, it is not clear to which format(s) we should limit the resulting expression in order to avoid explicitly checking whether $x \geq 5$ is entailed[28]. Thus an incremental algorithm for checking the entailment of inequalities is yet to be found.

For $\mathcal{BOOL}$ there seems to be a similar problem in detecting the entailment of Boolean constraints. However, in the case of groundness entailment some of the algorithms we have previously discussed are potentially incremental. The Prolog III algorithm, in fact, is designed with the detection of groundness as a criterion. The algorithm represents explicitly all variables that are grounded by the constraints. The Groebner basis algorithm will also contain in its basis an explicit representation of grounded variables. Finally, for the unification algorithms, the issue is clearly the form of the unifier. If the unifier is in fully simplified form then every ground variable will be associated to a ground value.

In summary for this subsection, the problem of detecting entailment is not limited just to the cost of determining if a particular constraint is entailed. Incrementality is crucial, and this property can be defined roughly as limiting the cost to depend on the number of guard constraints affected by each change to the store. In particular, dealing (even briefly) with the entire collection of guard constraints each time the store changes is unacceptable.

Below, in Section 11.1, an issue related to entailment is taken up. Here we have focussed on how to adapt the underlying satisfiability algorithm to be incremental for determining entailment. There we will consider the generic problem, independent of the constraint domain, of managing delayed goals which awake when certain constraints become entailed.

10.5. Projection

The problem at hand is to obtain a useful representation of the projection of constraints C w.r.t. a given set of variables. More formally, the problem is: given *target* variables $\tilde{x}$ and constraints $C(\tilde{x}, \tilde{y})$ involving variables from $\tilde{x}$ and $\tilde{y}$, express $\exists \tilde{y}\ C(\tilde{x}, \tilde{y})$ in the most usable form. While we cannot define usability formally, it typically means both conciseness and readability. An important area of use in the output phase of a CLP system: the desired output from running a goal is the projection of the answer constraints with respect to the goal variables. Here it is often useful to have only the target variables output (though, depending on the domain, this is not always possible). For example, the output of $x = z + 1, y = z + 2$ w.r.t. to x and y should be $x = y - 1$ or some rearrangement of this, but it should not involve any other variable. Another area of use is in meta-programming where a description of the current store may be wanted for further manipulation. For example,

[28] And this can of course be done, perhaps even efficiently, but the crucial point is, once again, that we cannot afford to do this every time the store changes.

projecting $\Re_{Lin}$ constraints onto a single variable x can show if x is bounded, and if so, this bound can be used in the program. Projection also provides the logical basis for eliminating variables from the accumulated set of constraints, once it is known that they will not be referred to again.

There are few general principles that guide the design of projection algorithms across the various constraint domains. The primary reason is, of course, that these algorithms have to be intimately related to the domain at hand. We therefore will simply resort to briefly mentioning existing approaches for some of our sample domains.

The projection problem is particularly simple for the domain $\mathcal{FT}$: the result of projection is $\tilde{x} = \tilde{x}\theta$ where θ is the substitution obtained from the solved form of C. Now, we have described above that this solved form is simply the mgu of C, that is, equations whose r.h.s. does not contain any variable on the l.h.s. . For example, $x = f(y), y = f(z)$ would have the solved form $x = f(f(z)), y = f(z)$. However, the equations $x = f(y), y = f(z)$ are more efficiently stored internally as they are (and this is done in actual implementations). The solved form for x therefore is obtained only when needed (during unification for example) by fully dereferencing y in the term $f(y)$. A direct representation of the projection of C on a variable x, as required in a printout for example, can be exponential in the size of C. This happens, for example, if C is of the form $x = f(x_1, x_1)$, $x_1 = f(x_2, x_2)$, $\cdots$, $x_n = f(a, a)$ because $x\theta$ would contain 2^{n+1} occurrences of the constant a. A solution would be to present the several equations equivalent to $x = x\theta$, such as the $n+1$ equations in this example. This however is a less explicit representation of the projection; for example, it would not always be obvious if a variable were ground.

Projection in the domain $\mathcal{RT}$ can be done by simply presenting those equations whose l.h.s. is a target variable and, recursively, all equations whose l.h.s. appears in anything already presented. Such a straightforward presentation is in general not the most compact. For example, the equation $x = f(f(x,x), f(x,x))$ is best presented as $x = f(x,x)$. In general, the problem of finding the most compact representation is roughly equivalent to the problem of minimizing states in a finite state automaton [60].

For $\Re_{LinEqn}$ the problem is only slightly more complicated. Recall that equations are maintained in parametric form, with eliminable and parametric variables. A relatively simple algorithm can be obtained by using a form of Gaussian elimination, and is informally described in Figure 1. It assumes there is some ordering on variables, and ensures that lower priority variables are represented in terms of higher priority variables. This ordering is arbitrary, except for the fact that the target variables should be of higher priority than other variables. We remark that a crucial point for efficiency is that the main loop in Figure 1 iterates n times, and this number (the number of target variables) is often far smaller than the total number of variables in the system. More details on this algorithm can be found in [132].

For $\Re_{Lin}$, there is a relatively simple projection algorithm. Assume all inequalities are written in a standard form ... ≤ 0. Let C_x^+ (C_x^-) denote the subset of constraints C in which x has only positive (negative) coefficients. Let C_x^0 denote those inequalities in C not containing x at all. We can now describe an algorithm, due to Fourier [91], which eliminates a variable x from a given C. If constraints c and c' have a positive and a negative coefficient of x, we can define $elim_x(c, c')$ to be a linear

let $x_1, \ldots, x_n$ be the target variables;
for $(i = 1; i \leq n; i = i + 1)$ {
  if (x_i is a parameter) **continue**;
  let e denote the equation $x_i = r.h.s.(x_i)$ at hand;
  if ($r.h.s.(x_i)$ contains a variable z of lower priority than x_i) {
    choose the z of lowest priority;
    rewrite the equation e into the form $z = t$;
    if (z is a target variable) mark the equation e as *final*;
    substitute t for z in the other equations ;
  } **else** mark the equation e as *final*;
}
return all final equations;

Figure 1. *Projection Algorithm for Linear Equations*

combination of c and c', which does not contain x.[29] A *Fourier step* eliminates x from a set of constraints C by computing $F_x(C) = \{elim_x(c, c') : c \in C_x^+, c' \in C_x^-\}$. It is easy to show that $\exists x C \leftrightarrow F_x(C)$. Clearly repeated applications of F eliminating all non-target variables result in an equivalent set of constraints in which the only variables (if any) are target variables.

The main problem with this algorithm, is that the worst-case size of $F_x(C)$, is $O(N^2)$ where N is the number of constraints in C. (It is in fact precisely $|C_x^0| + (|C_x^+| \times |C_x^-|) - (|C_x^+| + |C_x^-|)$.) In principle, the number of constraints needed to describe C using inequalities over variables $var(C) - \{x\}$ is far larger than the number of inequalities in C. In practice, however, the Fourier step generates many *redundant* constraints[30]. See [159] for discussion on such redundancy. Work by Černikov [48] proposed tests on the generated constraints to detect and eliminate some redundant constraints. The output module of the CLP($\mathcal{R}$) system [132] furthered these ideas, as did Imbert [125]. (Imbert [124] also considered the more general problem in which there are disequations.) All these redundancy elimination methods are *correct* in the following sense: if $\{\mathcal{C}_i\}_{i=1,2,\ldots}$ is the sequence of constraints generated during the elimination of variables $x_1, \cdots, x_i$ from C, then $\mathcal{C}_i \leftrightarrow \exists x_1 \ldots x_i\ \mathcal{C}$, for every i.

The survey [52] contains further perspectives on the Fourier variable elimination techique. It also contains a discussion on how the essential technique of Fourier can be adapted to perform projection in other domains such as linear integer constraints and the boolean domain.

We finish here by mentioning the non-Fourier algorithms of [123, 158]. In some circumstances, especially when the matrix representing the constraints is dense, the algorithm of [123] can be far more efficient. It is, however, believed that typical CLP programs produce sparse matrices. The algorithm of [158] has the advantageous property that it can produce an approximation of the projection if the size of the

[29] Obtained, for example, by multiplying c by $1/m$ and c' by $(-1/m')$, where m and m' are the coefficients of x in c and c' respectively, and then adding the resulting equations together.

[30] A constraint $c \in C$ is redundant in C if $C \leftrightarrow C - \{c\}$.

projection is unmanageably large.

10.6. Backtracking

The issue here is to restore the state of the constraint solver to a previous state (or, at least, an equivalent state). The most common technique, following Prolog, is the trailing of constraints when they are modified by the constraint solver and the restoration of these constraints upon backtracking. In Prolog, constraints are equations between terms, represented internally as bindings of variables. Since variables are implemented as pointers to their bound values[31], backtracking can be facilitated by the simple mechanism of an untagged trail [261, 6]. This identifies the set of variables which have been bound since the last choice point. Upon backtracking, these variables are simply reset to become unbound. Thus in Prolog, the only information to be trailed is which variables have just become bound, and untrailing simply unbinds these variables.

For CLP in general, it is necessary to record *changes* to constraints. While in Prolog a variable's expression simply becomes more and more instantiated during (forward) execution, in CLP, an expression may be completely changed from its original form. In $\Re_{LinEqn}$, for example, a variable x may have an original linear form and subsequently another. Assuming that a choice point is encountered just before the change in x, the original linear form needs to be trailed in case of backtracking. This kind of requirement in fact holds in all our sample domains with the exception of $\mathcal{FT}$ and $\mathcal{RT}$. Thus we have our first requirement on our trailing mechanism: the trail is a *value trail*, that is, each variable is trailed together with its associated expression. (Strictly speaking, we need to trail constraints rather than the expression a variable is associated to. However, constraints are typically represented internally as an association between a variable and an expression.)

Now, the trailing of expressions is in general far more costly than the trailing of the variables alone. For this reason, it is often useful to avoid trailing when there is no choice point between the time a variable changes value from one expression to another. A standard technique facilitating this involves the use of *time stamps*: a variable is always time stamped with the time that it last changed value, and every choice point is also time stamped when created. Now just before a variable's value is to be changed, its time stamp n is compared with the time stamp m of the most recent choice point, and if $n > m$, clearly no trailing is needed[32].

Next consider the use of a *cross-reference* table for solved forms, such as those discussed for the arithmetic domains, which use parametric variables. This is an index structure which maps each parametric variable to a list of its occurrences in the solved form. Such a structure is particularly useful, and can even be crucial for efficiency, in the process of substituting out a parametric variable (step (c) in algorithm 10.1). However, its use adds to the backtracking problem. A straightforward approach is to simply trail the entries in this table (according to time stamps). However, since these entries are in general quite large, and since the cross reference table is redundant from a semantic point of view, a useful approach is to *reconstruct* the table upon backtracking. The details of such reconstruction are

[31] Recall that this means that eliminable variables are not explicitly dereferenced in the r.h.s. of the equations in the solved form.

[32] In Prolog, one can think of the stack position of a variable as the time stamp.

straightforward but tedious, and hence are omitted here; see [133] for the case of the CLP($\mathcal{R}$) system. A final remark: this reconstruction approach has the added advantage of incurring cost only when backtracking actually takes place.

In summary, backtracking in CLP is substantially more complex than in Prolog. Some useful concepts to be added to the Prolog machinery are as follows: a value trail (and, in practice, a tagged trail as well because most systems will accommodate variables of different types, for example, the functor and arithmetic variables in CLP($\mathcal{R}$)); time stamps, to avoid repeated trailing for a variable during the lifetime of the same choice point; and finally, reconstruction of cross-references, rather than trailing.

11. Inference Engine

This section deals with extensions to the basic inference engine for logic programming needed because of constraints. What follows contains two main sections. In the first, we consider the problem of an incremental algorithm to manage a collection of delayed goals and constraints. This problem, discussed independently of the particular constraint domain at hand, reduces to the problem of determining which of a given set of guard constraints (cf Section 9) are affected as a result of change to the constraint store. The next section discusses extensions to the WAM, in both the design of instruction set as well as to the main elements of the runtime structure. Finally, we give a brief discussion of work on parallel implementations.

11.1. Delaying/Wakeup of Goals and Constraints

The problem at hand is to determine when a delayed goal is to be woken or when a passive constraint becomes active. The criteria for such an event is given by a guard constraint, that is, awaken the goal or activate the constraint when the guard constraint is entailed by the store[33]. In what follows, we use shall use the term delayed constraint to be synonymous with passive constraint to emphasize the similarities with delayed goals.

The underlying implementation issue, as far as the constraint solver is concerned, is how to efficiently process just those guard constraints that are affected as a result of a new input constraint[34] Specifically, to achieve incrementality, the cost of processing a change to the current collection of guard constraints should be related to the guard constraints affected by the change, and not to all the guard constraints. The following two items seem necessary to achieve this end.

First is a representation of what further constraints are needed so that a given guard constraint is entailed. For example, consider the delayed CLP($\mathcal{R}$) constraint $pow(x, y, z)$ (meaning $x = y^z$) which in general awaits the grounding of two of the three variables x, y, z. In constrast, the constraint $pow(1, y, z)$, only awaits the grounding of y (to a nonzero number) or z (to 1). In general, a delayed constraint is awoken by not one but a conjunction of several input constraints. When a subset of

[33] For guarded clauses, the problem is extended to determining which clause is to be chosen.

[34] However, significant changes to the inference engine is needed to handle delayed goals and guarded clauses. But these issues are the same as those faced by extending logic programming systems to implement delayed goals and/or guarded clauses (see, for example, [250]).

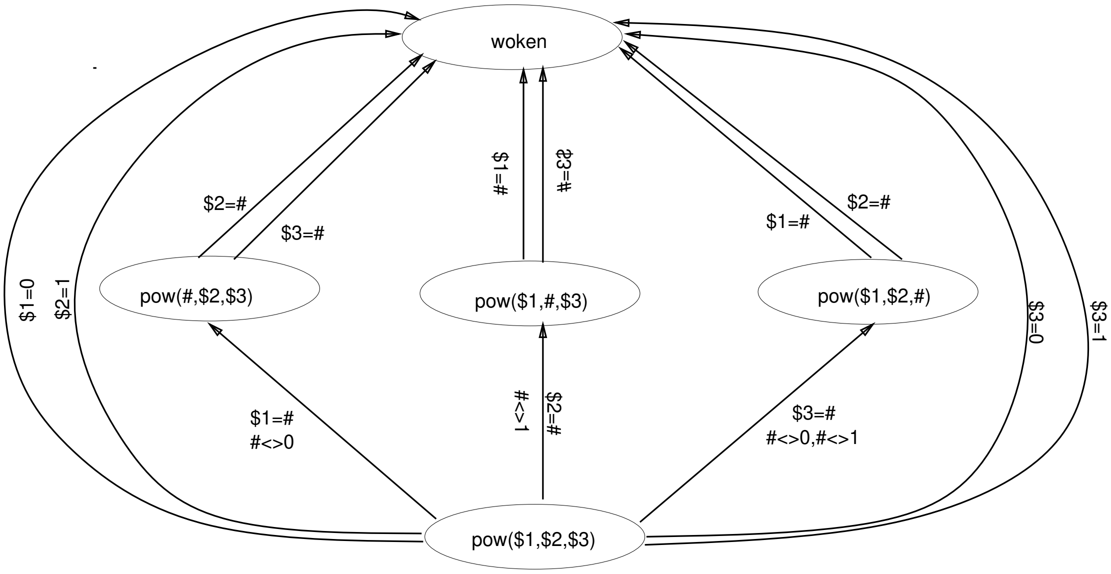

Figure 2. Wakeup system for *pow/3*

such input constraints has already been encountered, the runtime structure should relate the delayed constraint to (the disjunction of) just the remaining kinds of constraints which will awaken it.

Second, we require some index structure which allows immediate access to just the guard constraints affected as the result of a new input constraint. The main challenge is how to maintain such a structure in the presence of backtracking. For example, if changes to the structure were trailed using some adaptation of Prolog techniques [261], then a cost proportional to the number of entries can be incurred even though no guard constraints are affected.

The following material is a condensation of [134].

11.1.1. Wakeup Systems For the purposes of this section, we will describe an instance of a constraint in the form $p(\$_1, \cdots, \$_n) \wedge C$ where p is the n-ary constraint symbol at hand, $\$_1, \cdots, \$_n$ are distinguished variables used as templates for the arguments of p, and C is a constraint (which determines the values of $\$_1, \cdots, \$_n$).

A *wakeup degree* represents a subset of the p constraints, and a *wakeup system* consists of a set of wakeup degrees, and further, these degrees are organized into an automaton where transitions between degrees are labelled by constraints called *wakeup conditions*[35]. Intuitively, a transition occurs when a wakeup condition becomes entailed by the store. There is a distinguished degree called *woken* which represents active p constraints. We proceed now with an example.

Consider the CLP($\mathcal{R}$) constraint $pow(x, y, z)$ and see figure 2. A wakeup degree may be specified by means of constraints containing $\$_1, \cdots, \$_3$ (for describing the three arguments) and some *meta-constants* $\#, \#_1, \#_2, \cdots$ (for describing unspecified values). Thus, for example, $\$_2 = \#$ specifies that the second argument is ground.

[35] These are templates for the guard constraints.

Such a meta-language can also be used to specify the wakeup conditions. Thus, for example, the wakeup condition $\$_2 = \#, \# <> 0, \# <> 1$ attached to the bottom-most degree in figure 2 represents a transition of a constraint $pow(\$_1, \$_2, \$_3) \wedge C$, where C does not ground $\$_2$, into $pow(\$_1, \$_2, \$_3) \wedge C \wedge c$, where $C \wedge c$ does ground $\$_2$ into a number different from 0 and 1. The wakeup condition $\$_2 = 1$ which represents a transition to the degree *woken*, represents the fact that $pow(\$_1, 1, \$_3)$ is an active constraint (equivalent to $\$_1 = 1$). Similarly, $\$_3 = 1$ represents the transition to the active constraint $\$_1 = \$_2$. Note that there is no wakeup condition $\$_2 = 0$ because $pow(\$_1, 0, \$_3)$ (which is equivalent to $(\$_1 = 0 \wedge \$_3 \neq 0) \vee (\$_1 = 1 \wedge \$_3 = 0)$) is not active.

In general, there will be certain requirements on the structure of such an automaton to ensure that it does in fact define a mapping from p constraints into wakeup degrees, and that this mapping satisfies certain properties like: it defines a partition, it maps only active constraints into *woken*, it is consistent with the wakeup conditions specifying the transitions, etc. A starting point will be a formalization of meta-language used. These formal aspects are beyond the scope of this survey.

In summary, wakeup systems are an intuitive way to specify the organization of guard constraints. The wakeup degrees represent the various different cases of a delayed constraint which should be treated differently for efficiency reasons. Associated with each degree is a number of wakeup conditions which specify when an input constraint changes the degree of a delayed constraint. What is intended is that the wakeup conditions represent all the situations in which the constraint solver can efficiently update its knowledge about what further constraints are needed to wake the delayed constraint.

Before embarking on the runtime structure to implement delayed constraints such as *pow*, we amplify the abovementioned point about the similarities between delayed constraints and guarded clauses. Consider the guarded clause program:

```
pow(X,Y,Z) :- Y=1 | X=1.
pow(X,Y,Z) :- ground(X), X≠0, ground(Y), Y≠1 | Z=log(X)/log(Y).
pow(X,Y,Z) :- ground(X), X≠0, ground(Z) | Y=ᶻ√X.
pow(X,Y,Z) :- ground(Y), Y≠1, ground(Z) | X=Yᶻ.
pow(X,Y,Z) :- X=0 | Y=0, Z≠0.
pow(X,Y,Z) :- Z=0 | X=1.
pow(X,Y,Z) :- Z=1 | X=Y.
```

This program could be compiled into the wakeup system in Figure 2, where the three intermediate nodes reflect subexpressions in the guards that might be entailed without the entire guard being entailed. (More precisely, several *woken* nodes would be used, one for each clause body.) Thus wakeup systems express a central part of the implementation of (flat) guarded clauses. Since a guarded atom can be viewed as a one-clause guarded clause program for an anonymous predicate, wakeup systems are also applicable to implementing these constructs.

11.1.2. Runtime Structure Here we present an implementational framework in the context of a given wakeup system. There are three major operations with delayed goals or delayed constraints which correspond to the actions of delaying, awakening and backtracking:

1. adding a goal or delayed constraint to the current collection;
2. awakening a delayed goal or delayed constraint as the result of inputting a new (active) constraint, and
3. restoring the entire runtime structure to a previous state, that is, restoring the collection of delayed goals and delayed constraints to some earlier collection, and restoring all auxiliary structures accordingly.

In what follows, we concentrate on delayed constraints; as mentioned above, the constraint solver operations to handle delayed goals and guarded clauses are essentially the same.

The first of our two major structures is a stack containing the delayed constraints. Thus implementing operation 1 simply requires a push operation. Additionally, the stack contains constraints which are newer forms of constraints deeper in the stack. For example, if the constraint $pow(x, y, z)$ were in the stack, and if the input constraint $y = 3$ were encountered, then the new constraint $pow(x, 3, z)$ would be pushed, together with a pointer from the latter to the former. In general, the collection of delayed constraints contained in the system is described by the sub-collection of stacked constraints which have no inbound pointers.

Now consider operation 2. In order to implement this efficiently, it is necessary to have some access structure mapping an entailed constraint to just those delayed constraints affected. Since there are in general an infinite number of possible entailed constraints, a finite classification of them is required. A guard constraint, or simply guard for short, is an instance of a wakeup condition obtained by renaming the distinguished argument variables $\$_i$ into runtime variables. It is used as a template for describing the collection of entailed constraints (its instances) which affect the same sub-collection of delayed constraints. For example, suppose that the only delayed constraint is $pow(5, y, z)$ whose degree is $pow(\#, \$_2, \$_3)$ with wakeup conditions $\$_2 = \#$ and $\$_3 = \#$. Then only two guards need be considered: $y = \#$ and $z = \#$.

We now specify an index structure which maps a delayed constraint into a doubly linked list of occurence nodes. Each node contains a pointer to a stack element containing a delayed constraint[36]. Corresponding to each occurrence node is a reverse pointer from the stack element to the occurrence node. Call the list associated with a delayed constraint $\mathcal{DW}$ a $\mathcal{DW}$*-list*, and call each node in the list a $\mathcal{DW}$*-occurrence node*.

Initially the access structure is empty. The following specifies what is done for the basic operations:

Delay Push the constraint C onto the stack, and for each wakeup condition associated with (the degree of) C, create the corresponding guard and $\mathcal{DW}$-list. All occurrence nodes here are pointed to C.

Process Entailment Say $x = 5$ is now entailed. Find all guards which are implied by $x = 5$. If there are none, we are done. Otherwise, for each $\mathcal{DW}$-list L corresponding to each of these conditions, and for each constraint $C = p(\ldots) \wedge C'$ pointed to in L, (a) delete all occurrence nodes pointing to C (using the reverse pointers), push the new delayed constraint $C'' = p(\ldots) \wedge C' \wedge x = 5$

[36] The total number of occurrence nodes is generally larger than the number of delayed constraints.

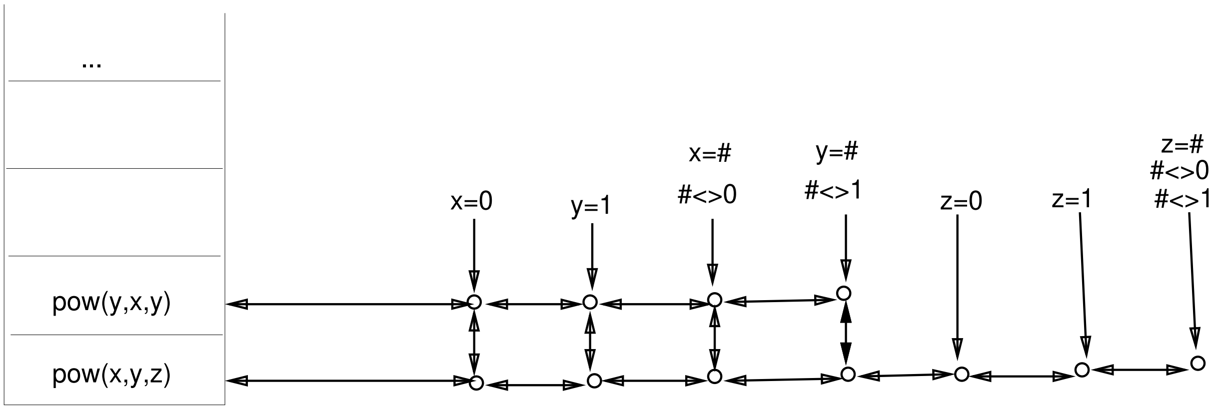

Figure 3. The index structure

with a (downward) pointer to C, and finally, (c) construct the new $\mathcal{DW}$-lists corresponding to C'' as defined above for the delay operation.

Backtrack Restoring the stack during backtracking is easy because it only requires a series of pops. Restoring the list structure, however, is not so straightforward because no trailing/saving of the changes was performed. In more detail, the operation of backtracking is the following: (a) pop the stack, and let C denote the constraint just popped. (b) Delete all occurrence nodes pointed to by C. If there is no pointer from C (and so it was a constraint that was newly delayed) to another constraint deeper in the stack, then nothing more need be done. (c) If there is a pointer from C to another constraint C' (and so C is the reduced form of C'), then perform the modifications to the access structure *as though* C' were being pushed onto the stack. These modifications, described above, involve computing the guards pertinent to C', inserting occurrence nodes, and setting up reverse pointers. Note that the index structure obtained in backtracking may not be structurally the same as that of the previous state. What is important, however, is that it depicts the same *logical* structure as that of the previous state.

Figure 3 illustrates the entire runtime structure after the two constraints $pow(x, y, z)$ and $pow(y, x, y)$ are stored, in this order. Figure 4 illustrates the structure after a new input constraint makes $x = 5$ entailed.

In summary, a stack is used to store delayed constraints and their reduced forms. An access structure maps a finite number of guards to lists of delayed constraints. The constraint solver is assumed to identify those conditions which are entailed. The cost of one primitive operation on delayed constraints (delaying a constraint, upgrading the degree of one delayed constraint, including awakening the constraint, and undoing the delay/upgrade of one constraint) is bounded by the (fixed) size

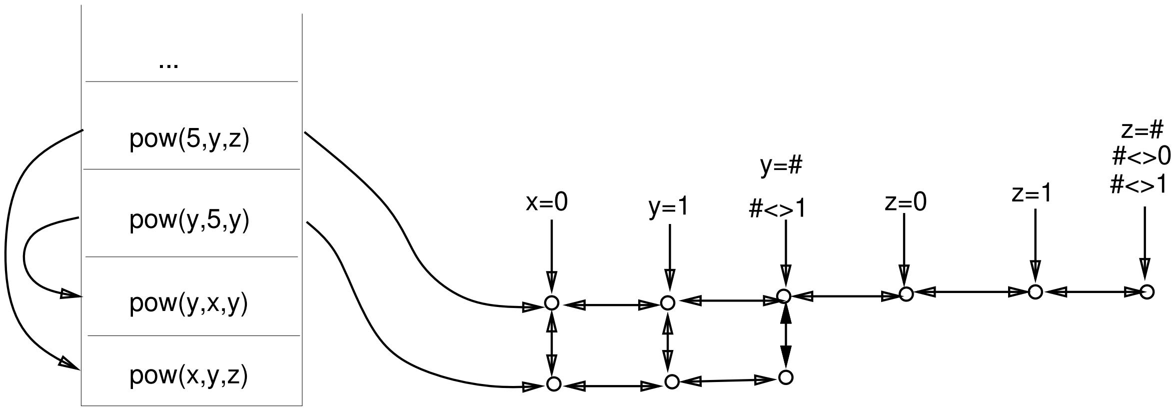

Figure 4. The index structure after $x = 5$ is entailed

of the underlying wakeup system. The total cost of an operation (delaying a new constraint, processing an entailed constraint, backtracking) on delayed constraints is proportional to the number of the delayed constraints affected by the operation.

11.2. Abstract Machine

This section discusses some major issues in the design of an abstract machine for the execution of CLP programs. The primary focus here will be on the design of the instruction set, with emphasis on the interaction between their use and information obtained from a potential program analyzer. Some elements of the runtime structure will also be mentioned.

In general, the essential features of the parts of an abstract machine dealing with constraints will differ greatly over CLP languages using different constraint domains. This is exemplified in the literature on CLP($\mathcal{R}$) [135], CHIP [3], and CLP($\mathcal{FD}$) [75]. The following presentation, though based on one work [135], contains material that is relevant to abstract machines for many CLP languages.

We begin by arguing that an abstract machine is the right approach in the first place. Abstract machines have been used for implementing programming languages for many reasons. Portability is one: only an implementation of the abstract machine needs to be made available on each platform. Another is simply convenience: it is easier to write a native code compiler if the task is first reduced to compiling for an abstract machine that is semantically closer to the source language. The best abstract machines sit at just the right point on the spectrum between the conceptual clarity of the high-level source language and the details of the target machine. In doing so they can often be used to express programs in exactly the right form for tackling the efficiency issues of a source language. For example, the Warren Abstract Machine [261, 6] revolutionized the execution of Prolog, since translating programs to the WAM exposed many opportunities for optimization that were not apparent at the source level. The benefit from designing an appropriate abstract machine for a given source language can be so great that even executing the abstract instruction code by interpretation can lead to surprisingly efficient implementations of a language. Many commercial Prolog systems compile to WAM-like code. Certainly more efficiency can be obtained from native code compilation, but the step that made Prolog usable was that of compiling to the WAM.

While the WAM made Prolog practical, global analysis shows the potential of making another major leap. For example, [248] and [208] used fairly efficient analyzers to generate high quality native code. Based on certain examples, they showed that the code quality was comparable to that obtained from a C compiler. In the case of CLP, the opportunities for obtaining valuable information from analysis are even greater than in Prolog. This is because the constraint solving step is in general far more involved than the unification step.

11.2.1. Instructions Next we consider the design of an abstract machine instruction set, in addition to the basic instruction set of the WAM. While the examples presented will be for CLP($\mathcal{R}$), the discussions are made for CLP systems in general. More details on this material can obtained from the theses [188, 266].

Our first requirement is a basic instruction for invoking the constraint solver. The format can be of the form

solve_xxx $\mathtt{X}_1$ $\mathtt{X}_2$ ... $\mathtt{X}_n$

where xxx indicates the kind of constraint and the $\mathtt{X}_i$ denotes the arguments. Typically these arguments are, as in the WAM, stack locations or registers. For example, in CLP($\mathcal{R}$), there are instructions of the form `initpf` n and `addpf` n, `X`, where n is a number and X a (solver) variable. The former initializes a parametric form to contain just the number n. The latter adds an entry of the form $n * pf(X)$ to the parametric form being stored in an *accumulator*, where $pf(X)$ is the parametric form for X in the store. Thus the accumulator in general stores an expression exp of the form $n + n_1{*}\mathtt{X}_1 + \cdots + n_k{*}\mathtt{X}_k$. Then, the instruction `solve_eq0` tests for the consistency of $exp = 0$ in conjunction with the store. If consistent, the solver adds the equation to the store; otherwise, backtracking occurs. There are similar instructions for inequalities.

There are important special kinds of constraints that justify making specialized versions of this basic instruction. While there are clearly many kinds of special cases, some specific to certain constraint domains, there are three cases which stand out:

1. the constraint is to be added to the store, but no satisfiability check is needed;
2. the constraint need not be added, but its satisfiability in conjunction with the store needs to be checked;
3. the constraint needs to be added and satisfiability needs to be checked, but the constraint is never used later.

To exemplify the special case 1, consider adding the constraint $5 + X - Y = 0$ to the store. Suppose that $Y = Z + 3.14$ is already in the store, and that X is a new variable. A direct compilation results in the following. Note that the rightmost column depicts the current state of the accumulator.

`initpf`	`5`	*accumulator* : 5
`addpf`	`1, X`	*accumulator* : $5 + X$
`addpf`	`-1, Y`	*accumulator* : $1.86 + X - Z$
`solve_eq0`		*solve* : $1.86 + X - Z = 0$

A better compilation can be obtained by using a specialized instruction `solve_no_fail_eq X` which adds the equation $X = exp$ to the store, where exp is the expression in the accumulator. The main difference here with `solve_eq0` is that no satisfiability check is performed. For the above example, we now can have

`initpf`	`-5`	*accumulator* : -5
`addpf`	`-1, Y`	*accumulator* : $-1.86 + Z$
`solve_no_fail_eq`	`X`	*add* : $X = -1.86 + Z$

In summary for this special case, for CLP systems in general, we often encounter constraints which can be organized into a form such that its consistency with the store is obvious. This typically happens when a new variable appears in an equation, for example, and new variables are often created in CLP systems. Thus the instructions of the form `solve_no_fail_`xxx are justified.

Next consider the special case 2, and the following example CLP($\mathcal{R}$) program.

```
sum(0, 0).
sum(N, X) :-
   N >= 1,
   N1 = N - 1,
   X1 = X - N,
   sum(N1, X1).
```

Of concern to us here are constraints that, if added to the store, can be shown to become redundant as a result of future additions to the store. This notion of *future redundancy* was first described in [138]. Now if we execute the goal `sum(N, X)` using the second rule above, we obtain the subgoal

```
?- N >= 1, N1 = N - 1, X1 = X - N, sum(N1, X1).
```

Continuing the execution, we now have two choices: choosing the first rule we obtain the new constraint $N1 = 0$, and choosing the second rule we obtain the constraint $N1 \geq 1$ (among others). In each case the original constraint $N \geq 1$ is made redundant. The main point of this example is that the constraint $N \geq 1$ in the second rule should be implemented simply as a test, and not added to the constraint store. We hence define the new class of instructions `solve_no_add_`xxx.

This example shows that future redundant constraints do occur in CLP systems. However, one apparent difficulty with this special case is the problem of *detecting* its occurrence. We will mention relevant work on program analysis below. Meanwhile, we remark that experiments using CLP($\mathcal{R}$) have shown that this special case leads to the most substantial efficiency gains compared to the other two kinds of special cases discussed in this section [188, 266].

Finally consider special case 3. Of concern here are constraints which are neither entailed by the store as in case 1 nor are eventually made redundant as in case 2, but which are required to be added to the store, and checked for consistency. What makes these constraints special is that after they have been added to the store (and the store is recomputed into its new internal form), their variables appear in those parts of the store that are never again referred to. Consider the `sum` program once again. The following sequence of constraints arise from executing the goal `sum(7, X)`:

$$
\begin{array}{lrcl}
(1) & X1 & = & X - 7 \\
(2) & X1' & = & (X - 7) - 6 \\
(3) & X1'' & = & ((X - 7) - 6) - 5 \\
\ldots & & \ldots &
\end{array}
$$

Upon encountering the second equation $X1' = X1 - 6$ and simplifying into (2), note that the variable $X1$ will never be referred to in future. Hence equation (1) can be deleted. Similarly, upon encountering the third equation $X1'' = X1' - 5$ and simplifying into (3), the variable $X1'$ will never be referred to in future and so (2) can be deleted. In short, only one equation involving X need be stored at any point in the computation. We hence add the class of instructions of the form `add_and_delete X` which informs the solver that after considering the constraint associated to `X`, it may delete all structures associated to `X`. In CLP($\mathcal{R}$), the corresponding instruction is `addpf_and_delete` n, `X`, the obvious variant of the previously described instruction `addpf` n, `X`. Compiling this `sum` example gives

```
(1)   init_pf              -7
      addpf                1, X
      solve_no_fail_eq     X1
(2)   init_pf              -6
      addpf_and_delete     1, X1
      solve_no_fail_eq     X1'
(3)   init_pf              -5
      addpf_and_delete     1, X1'
      solve_no_fail_eq     X1''
...   ...
```

Note that a different set of instructions is required for the first equation from that required for the remaining equations. Hence the first iteration needs to be unrolled to produce the most efficient code. The main challenge for this special case is, as in special case 2, the detection of the special constraints. We now address this issue.

11.2.2. Techniques for CLP Program Analysis The kinds of program analysis required to utilize the specialized instructions include those techniques developed for Prolog, most prominently, detecting special cases of unification and deterministic predicates. Algorithms for such analysis have become familiar; see [73, 74] for example. See [98], for example, for a description of how to extend the general techniques of abstract interpretation applicable in LP to CLP. Our considerations above, however, require rather specific kinds of analyses.

Detecting redundant variables and future redundant constraints can in fact be done without dataflow analysis. One simple method involves unfolding the predicate definition (and typically once is enough), and then, in the case of detecting redundant variables, simply inspecting where variables occur last in the unfolded definitions. For detecting a future redundant constraint, the essential step is determining whether the constraints in an unfolded predicate definition imply the constraint being analyzed.

An early work describing these kinds of optimizations is [138], and some further discussion can also be found in [135]. The latter first described the abstract machine CLAM for CLP($\mathcal{R}$), and the former first defined and examined the problem of our special case 2, that of detecting and exploiting the existence of future redundant constraints in CLP($\mathcal{R}$). More recently, [171] reported new algorithms for the problem of special case 3, that of detecting redundant variables in CLP($\mathcal{R}$). The work [182] describes, in a more general setting, a collection of techniques (entitled refinement, removal and reordering) for optimization in CLP systems. See also [184] for an overview of the status of CLP($\mathcal{R}$) optimization and [188, 266] for detailed empirical results.

Despite the potential of optimization as reported in these works, the lack of (full) implementations leaves open the practicality of using these and other sophisticated optimization techniques for CLP systems in general.

11.2.3. Runtime Structure A CLP abstract machine requires the same basic runtime support as the WAM. Some data structures needed are a routine extension of those for the WAM — the usual register, stack, heap and trail organization. The main new structures pertain to the solver. Variables involved in constraints typically have a *solver identifier*, which is used to refer to that variable's location in the solver data structures.

The modifications to the basic WAM architecture typically would be:

- *Solver identifiers*
  It is often necessary to have a way to index from a variable to the constraints it is involved in. Since the WAM structure provides stack locations for the dynamically created variables, it remains just to have a tag and value structure to respectively (a) identify the variable as a solver variable, and (b) access the constraint(s) associated with this variable. Note that the basic unification algorithm, assuming functors are used in the constraint system, needs to be augmented to deal with this new type.
- *Tagged trail*
  As mentioned in Section 10.6, the trail in the WAM merely consists of a stack of addresses to be reset on backtracking. In CLP systems in general, the trail is also used to store changes to constraints. Hence a tagged value trail is required. The tags specify what operation is to be reversed, and the value component, if present, contains any old data to be restored.
- *Time-stamped data structures*
  Time stamps have been briefly discussed in Section 10.6. The basic idea here is that the data structure representing a constraint may go through several changes without there being a new choice point encountered during this activity. Clearly only one state of the structure need be trailed for each choice point.
- *Constraint accumulator*
  A constraint is typically built up using a basic instruction repeatedly, for example, the `addpf` instruction in CLP($\mathcal{R}$). During this process, the partially constructed constraint is represented in an accumulator. One of the `solve` instructions then passes the constraint to the solver. We can think of this linear form accumulator as a generalization of the accumulator in classical computer architectures, accumulating a partially constructed constraint

instead of a number.

11.3. Parallel Implementations

We briefly outline the main works involving CLP and parallelism. The opportunities for parallelism in CLP languages are those that arise, and have already been addressed, in the logic programming context (such as or-parallelism, and-parallelism, stream-parallelism), and those that arise because of the presence of a potentially computationally costly constraint solver.

The first work in this area [108] was an experimental implementation of an or-parallel for a CLP language with domain $\mathcal{FD}$. That approach has been pursued with the development of the ElipSys system [254], which is the most developed of the parallel implementations of CLP languages.

Atay [15, 16] presents the or-parallelization of 2LP, a language that computes with linear inequalities over reals and integers, but in which rules do not have local variables[37]. Another work deals with the or-parallel implementation of a CLP language over $\mathcal{FD}$ on massively parallel SIMD computers [252]. However the basis for the parallelism is not the nondeterministic choice of rules, as in conventional LP or-parallelism, but the nondeterministic choice of values for a variable.

Work on and-parallelism in logic programming depends heavily on notions of independence of atoms in a goal. [99] addresses this notion in a CLP context, and identify notions of independence for constraint solvers which must hold if the advantages of and-parallelism in LP are to be fully realized in CLP languages. However, there has not, to our knowledge, been any attempt to produce an and-parallel implementation of a CLP language.

Two works address both stream-parallelism and parallelism in constraint solving. GDCC [249] is a committed-choice language that can be crudely characterized as a committed-choice version of CAL. It uses constraints over domains of finite trees, Booleans, real numbers and integers. [249] mainly discusses the parallelization of the Groebner basis algorithms, which are the core of the solvers for the real number and Boolean constraint domains, and a parallel branch-and-bound method that is used in the integer solver. Leung [166] addresses the incorporation of constraint solving in both a committed-choice language and a language based on the Andorra model of computation. He presents distributed solvers for finite domains, the Boolean domain and linear inequalities over the reals. The finite domain solver is based on [108], the solver for the reals parallelizes the Simplex algorithm, and the Boolean solver parallelizes the unification algorithm of [45].

Finally, [43, 42] reports the design and initial implementation of CLP($\mathcal{R}$) with an execution model in which the inference engine and constraint solver compute concurrently and asynchronously. One of the issues addressed is backtracking, which is difficult when the engine and solver are so loosely coupled.

[37] We say that a variable in a rule is *local* if it appears in the body of the rule, but not in the head.

Part III
Programming and Applications

In this final part, we discuss the practical use of CLP languages. The format here is essentially a selected list of successful applications across a variety of problem domains. Each application is given an overview, with emphasis on the particular programming paradigm and CLP features used.

It seems useful to classify CLP applications broadly into two classes. In one class, the essential CLP technique is to use constraints and rules to obtain a transparent representation of the (relationships underlying the) problem. Here the constraints also provide a powerful query language. The other class caters for the many problems which can be solved by enumeration algorithms, the combinatorial search problems. Here the LP aspect of CLP is useful for providing the enumeration facility while constraints serve to keep the search space manageable.

12. Modelling of Complex Problems

We consider here the use of CLP as a specification language: constraints allow the declarative interpretation of basic relationships, and rules combine these for complex relationships.

12.1. Analysis and Synthesis of Analog Circuits

This presentation is adapted from [104], an early application of CLP($\mathcal{R}$). Briefly, the general methodology for representing properties of circuits are that constraints at a base level describe the relationship between variables corresponding to a subsystem, such as Ohm's law, and constraints at a higher level describe the interaction between these subsystems, such as Kirchoff's law.

Consider the following program fragment defining the procedure `circuit(N, V, I)` which specifies that, across an electrical network `N`, the potential difference and current are `V` and `I` respectively. The network is specified in an obvious way by a term containing the functors `resistor`, `series` and `parallel`. In this program, the first rule states the required voltage-current relationship for a resistor, and the remaining rules combine such relationships in a network of resistors.

```
circuit(resistor(R), V, I) :- V = I * R.
circuit(series(N1, N2), V, I) :-
   I = I1,
   I = I2,
   V = V1 + V2,
   circuit(N1, V1, I1),
   circuit(N2, V2, I2).
circuit(parallel(N1, N2), V, I) :-
   V = V1, V = V2,
   I = I1 + I2,
   circuit(N1, V1, I1),
   circuit(N2, V2, I2).
```

For example, the query

```
?- circuit(series(series(resistor(R),resistor(R)),resistor(R)),V,5)
```

asks for the voltage value if a current value of 5 is flowing through a network containing just three identical resistors in series. (The answer is `R = 0.0666667*V`.) Additional rules can be added for other devices. For example, the piece-wise linear model of a diode described by the voltage-current relationship

$$I = \begin{cases} 10V + 1000 & \text{if } V < -100 \\ 0.001V & \text{if } -100 \leq V \leq 0.6 \\ 100V - 60 & \text{if } V > 0.6 \end{cases}$$

is captured by the rules:

```
circuit(diode, V, 10 * V + 1000) :- V < -100.
circuit(diode, V, 0.0001 * V) :- -100 <= V, V <= 0.6.
circuit(diode, V, 100 * V - 60) :- V > 0.6.
```

This basic idea can be extended to model AC networks. For example, suppose we wish to reason about an RLC network in steady-state. First, we dispense with complex numbers by representing $X + iY$ as a CLP($\mathcal{R}$) term `c(X, Y)`, and use:

```
c_equal(c(Re, Im), c(Re, Im)).
c_add(c(Re1, Im1), c(Re2, Im2), c(Re1 + Re2, Im1 + Im2)).
c_mult(c(Re1, Im1), c(Re2, Im2), c(Re3, Im3)) :-
   Re3 = Re1 * Re2 - Im1 * Im2,
   Im3 = Re1 * Im2 + Re2 * Im1.
```

to implement the basic complex arithmetic operations of equality, addition and multiplication.

Now consider the following procedure `circuit(N, V, I, W)` which is like its namesake above except that the voltage and current values are now complex numbers, and the new parameter `W`, a real number, is the angular frequency. It is noteworthy that this significant extension of the previous program fragment for `circuit` has been obtained so easily.

```
circuit(resistor(R), V, I, W) :- c_mult(V, I, c(R, 0)).
circuit(inductor(L), V, I, W) :- c_mult(V, I, c(0, W * L)).
circuit(capacitor(C), V, I, W) :- c_mult(V, I, c(0, -1 / (W * C))).
circuit(series(N1, N2), V, I, W) :-
   c_equal(I, I1), c_equal(I, I2),
   c_add(V, V1, V2),
   circuit(N1, V1, I1, W),
   circuit(N2, V2, I2, W).
circuit(parallel(N1, N2), V, I, W) :-
   c_equal(V, V1), c_equal(V, V2),
   c_add(I, I1, I2),
   V = V1, V = V2,
   I = I1 + I2,
   circuit(N1, V1, I1, W),
   circuit(N2, V2, I2, W).
```

We close this example application by mentioning that the work in [104] not only contains further explanation of the above technique, but also addresses other problems such as the *synthesis* of networks and digital signal flow. Not only does the CLP approach provide a concise framework for modelling circuits (previously done in a more ad-hoc manner), but it also provides additionality functionality because relationships, as opposed to values, are reasoned about. Evidence that this approach can be practical was given; for example, the modelling can be executed at the rate of about a hundred circuit components per second on a RS6000 workstation.

12.2. Options Trading Analysis

Options are contracts whose value is contingent upon the value of some underlying asset. The most common type of option are those on company shares. A *call option* gives the holder the right to buy a fixed number of shares at a fixed *exercise price* until a certain maturity/expiration date. Conversely, a *put option* gives the holder the right to sell at a fixed price. The option itself may be bought or sold. For example, consider a call option costing \$800 which gives the right to purchase 100 shares at \$50 per share within some period of time. This call option can be sold at the current market price, or exercised at a cost of \$5000. Now if the price of the share is \$60, then the option may be exercised to obtain a profit of \$10 per share; taking the price of the option into account, the net gain is \$200. After the specified period, the call option, if not exercised, becomes worthless. Figure 5 shows *payoff diagrams* which are a simple model of the relationship between the value of a call option and the share price. Sell options have similar diagrams. Note that `c` denotes the cost of the option and `x` the exercise price.

Options can be combined in arbitrary ways to form artificial financial instruments. This allows one to tailor risk and return in flexible ways. For example, the *butterfly* strategy in figure 5 consists of buying two calls, one at a lower strike price `x` and one at a higher price `z` and selling two calls at the middle strike price `y`. This makes a profit if the share stays around the middle strike price and limits the loss if the movement is large.

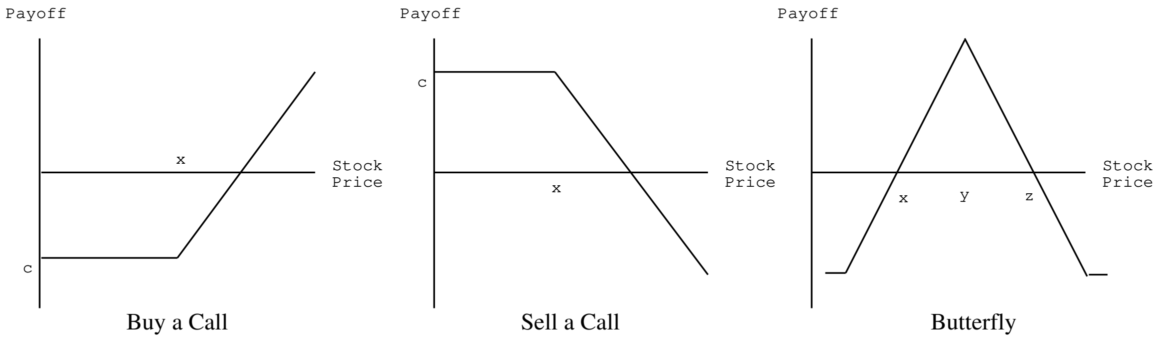

Figure 5. Payoff Diagrams

The following presentation is due to Yap [266], based in his work using CLP($\mathcal{R}$). This material appeared in [157], and the subsequently implemented OTAS system is described in [122]. There are several main reasons why CLP, and CLP($\mathcal{R}$) in particular, are suitable for reasoning about option trading: there are complex trading strategies used which are usually formulated as rules; there is a combinatorial aspect to the problem as there are many ways of combining options; a combination of symbolic and numeric computation is involved; there are well developed mathematical valuation models and constraints on the relationships involved in option pricing, and finally, flexible "what-if" type analysis is required.
A simple mathematical model of valuing options, and other financial instruments such as stocks and bonds, is with linear piecewise functions. Let the heavyside function h and the ramp function r be defined as follows:

$$h(x,y) = \begin{cases} 0 & \text{if } x > y \\ 1 & \text{otherwise} \end{cases} \quad and \quad r(x,y) = \begin{cases} 0 & \text{if } x > y \\ y - x & \text{otherwise} \end{cases}$$

The payoff function for call and put options can now be described by the following matrix product which creates a linear piecewise function:

$$payoff = [h_1, h_2, r_1, r_2] \times \begin{bmatrix} h(b_1, s) \\ h(b_2, s) \\ r(b_1, s) \\ r(b_2, s) \end{bmatrix}$$

where s is the share price, b_i is either the strike price or 0, and h_i and r_i are multipliers of the heavyside and ramp functions. In the following program, the variables `S, X, R` respectively denote the stock price, the exercise price and the interest rate.

```
h(X, Y, Z) :- Y < X, Z = 0.
h(X, Y, Z) :- Y >= X, Z = 1.
r(X, Y, Z) :- Y < X, Z = 0.
r(X, Y, Z) :- Y >= X, Z = Y - X.
value(Type, Buy_or_Sell, S, C, P, R, X, B, Payoff) :-
   sign(Buy_or_Sell, Sign),
   data(Type, S, C, P, R, X, B, B1, B2, H1, H2, R1, R2),
   h(B1, S, T1), h(B2, S, T2), r(B1, S, T3), r(B2, S, T4),
   Payoff = Sign*(H1*T1 + H2*T2 + R1*T3 + R2*T4).
```

The parameters for the piecewise functions can be expressed symbolically in the following tables, implemented simply as CLP facts.

```
sign(buy, -1).
sign(sell, 1).
data(stock, S, C, P, R, X, B, 0, 0, S*R, 0, -1, 0).
data(call, S, C, P, R, X, B, 0, X, C*R, 0, 0, -1).
data(put, S, C, P, R, X, B, 0, X, P*R-X, 0, 1, -1).
data(bond, S, C, P, R, X, B, 0, 0, B*R, 0, 0, 0).
```

This program forms the basis for evaluating option combinations. The following direct query evaluates the sale of a call option that expires *in-the-money*[38],

```
?- Call = 5, X = 50, R = 1.05, S = 60,
   value(call, sell, S, Call, _, R, X, _, Payoff).
```

giving the answer, `Payoff = -4.75`. More general queries make use of the ability to reason with inequalities. We can ask for what share price does the value exceed 5,

```
?- Payoff > 5, C = 5, X = 50, R = 1.05, S = 60,
   value(call, sell, S, C, _, R, X, _, Payoff).
```

The answer constraints returned[39] illustrates the piecewise nature of the model,

```
Payoff = 5.25, S < 50;
Payoff = 55.25 - S, 50 <= S, S <= 50.25.
```

More complex combinations can be constructed by composing them out of the base financial instruments and linking them together with constraints. For example, the

[38] That is, when the strike price is less than the share price.

[39] We will use ';' to separate different sets of answer constraints in the output.

following is a combination of two calls and two puts,

```
?- R = 0.1, Payoff = Payoff1 + Payoff2 + Payoff3 + Payoff4,
   P1 = 10, K1 = 20, value(put, sell, S, _, P1, R, K1, _, Payoff1),
   P2 = 18, K2 = 40, value(put, buy, S, _, P2, R, K2, _, Payoff2),
   C3 = 18, K3 = 60, value(call, buy, S, C3, _, R, K3, _, Payoff3),
   C4 = 18, K4 = 60, value(call, sell, S, C4, _, R, K4, _, Payoff4).
```

The answer obtained illustrates how combinations of options can be tailored to produce a custom linear piecewise payoff function.

```
Payoff = 5.7, S < 20;
Payoff = 25.7 - S, 20 <= S, S < 40;
Payoff = -14.3, 40 <= S, S < 60;
Payoff = S - 74.3, 60 <= S, S < 80;
Payoff = 5.7, 80 <= S.
```

The above is just a brief overview of the core ideas behind the work in [157]. Amongst the important aspects that are omitted are consideration of option pricing models, and details of implementing the decision support system OTAS [122]. As in the circuit modelling application described above, the advantages of using CLP($\mathcal{R}$) here are that the program is concise and that the query language is expressive.

12.3. Temporal Reasoning

It is natural and common to model time as an arithmetic domain, and indeed we do this in everyday life. Depending upon the application, a discrete representation (such as the integers) or a continuous representation (such as the reals) may be appropriate, and varying amounts of the arithmetic signature are needed (for example, we might use only the ordering, or use only a successor function). In this brief discussion we assume that time is linearly ordered, although this is not a universally accepted choice [84].

Temporal logic [84] is often used as a language for expressing time-related concepts. Temporal logic adds to standard first-order logic such constructs as *next* (meaning, roughly, "in the next time instant"[40]), *always* (meaning "in every future time instant"), and *sometime* (meaning "in some future time instant"). The language Templog [1] was designed based on a Horn-like subset of temporal logic in which the meaning of function symbols does not vary with time, but the meaning of predicate symbols does. It was shown in [39] that the operational behavior of Templog could be mimicked by a CLP language via the following natural translation: every predicate receives another argument, representing time. Then, at time t, *next* is represented by $t' = t + 1$, and the future (for *always* and *sometime*) is represented by $t' \geq t$. In later work [40], Brzoska has presented a more powerful temporal logic language which also can be viewed as, and implemented through, a CLP language.

Often we wish to manipulate the time parameter more directly than is possible in conventional temporal logic. For example, we may wish to express durations as well as times. We can do this if we include $+$ in the signature of our domain modelling

[40] We assume that time is modelled by the integers.

time. This is used in applications to scheduling, among others, as discussed in section 13.

The use of simple constraint domains to model time has been explored extensively in the context of temporal databases. In this situation, an item of data might incorporate the time interval for which it is valid. Simple domains have been considered because of over-riding requirements for quick and terminating execution of queries, as discussed in section 7. Furthermore, often the restriction is made that only one or two arguments in a tuple are time-valued, with the other arguments taking constant values. [19] surveys work in this area using an integer model of time.

13. Combinatorial Search Problems

CLP offers an easy realization of enumeration algorithms for the solving of combinatorial problems. Given decision variables $x_1, \cdots, x_n$, one uses a CLP program schema of the form

```
solve(X1, ... , Xn) :-
    constraints(X1, ... , Xn),
    enumerate(X1, ... , Xn).
```

to implement a "constrain-and-generate" enumeration strategy (also called implicit enumeration), as opposed to naive enumerate-and-test strategy, to curtail the search space. We refer to the basic text [107], chapter 2, for further introductory material to this CLP approach.

The above schema is used to represent the set of all solutions to the constraints. Often one desires an *optimal* solution according to some criteria, say the solution $a_1, \cdots, a_n$ to $x_1, \cdots, x_n$ that minimizes some given function $cost(x_1, \cdots, x_n)$. The simplest strategy to obtain this solution is simply to obtain and check each and every solution of `solve`. An easy improvement is obtained by augmenting the search with a *branch-and-bound* strategy. Briefly, the cost of the best solution encountered so far is stored and the continuing search is constrained to find only new solutions of better cost. More concretely, CLP systems typically provide predicates such as `minimize(solve(X1, ... , Xn, Cost), BestCost)` (and similarly `maximize(...)`) where `solve(X1, ... , Xn, Cost)` serves to obtain one solution as explained above, with cost `Cost`, and `BestCost` is a number representing the cost of the best solution found so far. (Initially, this number can be any sufficiently large number.) It is assumed here that the procedure `solve(X1, ... , Xn, Cost)` maintains a lower bound for a variable `Cost`, which is computed as the values of the decision variables are determined. The `minimize` procedure then essentially behaves as a repeated invocation of the goal `?- Cost < BestCost, solve(X1, ... , Xn, Cost)`. In general, the choice of a suitable cost function can be difficult. Finally, we refer the reader to the text [107] (section 4.5.1) for more a detailed explanation of how branch-and-bound is used in CLP systems.

The constraint domain at hand is discrete and typically finite (since the enumeration must cover all candidate values for the sequence $x_1, \cdots, x_n$), and therefore constraint solving is almost always NP-hard. This in turn restricts implementations to the use of partial solvers, that is, not all constraints will be considered active. Recall that partial solvers are, however, required to be conservative in the sense that whenever unsatisfiability is reported, the tested constraints are indeed unsatisfiable.

In general, the primary efficiency issues are:

- How complete is the constraint solver? In general, there is tradeoff between the larger cost of a more complete solver and the smaller search space that such a solver can give rise to.
- What constraints to use to model the problem? A special case of this issue concerns the use of *redundant* constraints, that is, constraints that do not change the meaning of the constraint store. In general, redundant constraints will slow down a CLP system with a complete solver. With partial solvers, however, redundant constraints may be useful to the solver in case the equivalent information in the constraint store is not active.
- In which order do we choose the decision variables for enumeration? And should such order be dynamically determined?
- In which order do we enumerate the values for a given decision variable? And should such order be dynamically determined?

In this section, we will outline a number of CLP applications in specific combinatorial problem areas. In each subsection below, unless otherwise specified, we shall assume that the underlying constraint system is based on the integers.

13.1. Cutting Stock

The following describes a two-dimensional cutting stock problem pertaining to furniture manufacturing, an early application of CHIP [77]. We are given a sawing machine which cuts a board of wood into a number of different sized shelves. The machine is able to cut in several configurations, each of which determines the number of each kind of shelf, and some amount of wood wasted. Let there be N different kinds of shelves, and M different configurations. Let $S_{i,j}$, $1 \leq i \leq M$, $1 \leq j \leq N$, denote the number of shelves j cut in configuration i. Let W_i, $1 \leq i \leq M$, denote the wastage in configuration i. Let R_i, $1 \leq i \leq N$ denote the number of shelves i required. The problem now can be stated as finding the configurations such that the required number of shelves are obtained and the wastage minimized.

In [77], there were 6 kinds of shelves, 72 configurations, and the number of boards to be cut was fixed at 4. Two solutions were then presented, which we now paraphrase.

Let X_i, $1 \leq i \leq 72$, denote the number of boards cut according to configuration i. Thus $X_1 + \cdots + X_{72} = 4$. The requirements on the number of shelves are expressed via the constraints $X_1 * S_{1,j} + \cdots + X_{72} * S_{72,j}$, for $1 \leq j \leq N$. The objective function, to be minimized, is $X_1 * W_1 + \cdots + X_{72} * W_{72}$. The straightforward program representation of all this is given below. The `enumerate` procedure has the range $\{1, 2, 3, 4\}$. Note that `solve` is run repeatedly in the search for the solution of lowest `Cost`.

```
solve(X1, ... , X72, Cost) :-
    X1 + ... + X72 = 4,
    X1 * S1,1 + ... + X72 * S72,1 >= R1,
    X1 * S1,2 + ... + X72 * S72,2 >= R2,
```

```
...
X₁ * S₁,₆ + ... + X₇₂ * S₇₂,₆ >= R₆,
Cost = X₁ * W₁ + ... + X₇₂ * W₇₂,
enumerate(X₁, ... , X₇₂).
```

The second solution uses the special CHIP constraint `element`, described above in section 9.2. Recall that `element(X, List, E)` expresses that the X^{th} element of `List` is `E`. In this second approach to the problem, the variables X_i, $1 \leq i \leq 4$, denote the configurations chosen. Thus $1 \leq X_i \leq 72$. Let $T_{i,j}$, $1 \leq i \leq 4$, $1 \leq j \leq 6$, denote the number of shelves j in configuration i. Let $Cost_i$, $1 \leq i \leq 4$, denote the wastage in configuration X_i. Thus the required shelves are obtained by the constraints $T_{1,j} + \cdots + T_{4,j} \geq R_j$ where $1 \leq j \leq 6$, and the total cost is simply $Cost_1 + \cdots + Cost_4$.

In program below, the constraints $X_1 \leq X_2 \leq X_3 \leq X_4$ serve to eliminate consideration of symmetrical solutions. The following group of 24 `element` constraints serve to compute the $T_{i,j}$ variables in terms of the (given) $S_{i,j}$ values and the (computed) X_i values. The next group of 4 `element` constraints computes the $Cost_i$ variables in terms of the (given) W_i variables. The `enumerate` procedure has the range $\{1, 2, \cdots, 72\}$. Once again, `solve` is run repeatedly here in the search for the lowest `Cost`.

```
solve(X₁, ... , X₄, Cost) :-
    X₁ <= X₂, X₂ <= X₃, X₃ <= X₄,
    element(X₁, [S₁,ⱼ, ... , S₇₂,ⱼ], T₁,ⱼ),      % (1 ≤ j ≤ 6)
    element(X₂, [S₂,ⱼ, ... , S₇₂,ⱼ], T₂,ⱼ),
    element(X₃, [S₃,ⱼ, ... , S₇₂,ⱼ], T₃,ⱼ),
    element(X₄, [S₄,ⱼ, ... , S₇₂,ⱼ], T₄,ⱼ),
    element(X₁, [W₁, ... W₇₂], Cost₁),
    element(X₂, [W₁, ... W₇₂], Cost₂),
    element(X₃, [W₁, ... W₇₂], Cost₃),
    element(X₄, [W₁, ... W₇₂], Cost₄),
    T₁,ⱼ + T₂,ⱼ + T₃,ⱼ + T₄,ⱼ >= Rᵢ,                % (1 ≤ j ≤ 6)
    Cost = Cost₁ + Cost₂ + Cost₃ + Cost₄,
    enumerate(X₁, X₂, X₃, X₄).
```

The second program has advantages over the first. Apart from a smaller search space (approximately 10^7 in comparison with 10^{43}), it was able to avoid encountering symmetrical solutions. The timings given in [77] showed that the second program ran much faster. This comparison exemplifies the abovementioned fact that the way a problem is modelled can greatly affect efficiency.

13.2. DNA Sequencing

We consider a simplified version of the problem of restriction site mapping (RSM). Briefly, a DNA sequence is a finite string over the letters $\{A, C, G, T\}$, and a restriction enzyme partitions a DNA sequence into certain fragments. The problem is then to reconstruct the original DNA sequence from the fragments and other information obtained through experiments. In what follows, we consider an abstraction of this problem which deals only with the lengths of fragments, instead of the fragments themselves.

Consider the use of two enzymes. Let the first enzyme partition the DNA sequence into $A_1, \cdots, A_N$ and the second into $B_1, \cdots, B_M$. Now, a simultaneous use of the two enzymes also produces a partition $D_1, \cdots, D_K$ corresponding to combining the previous two partitions. That is,

$$\forall i \exists j : A_1 \cdots A_i = D_1 \cdots D_j \text{ and } \forall i \exists j : B_1 \cdots B_i = D_1 \cdots D_j, \text{ and conversely,}$$
$$\forall j \exists i : (D_1 \cdots D_j = A_1 \cdots A_i) \vee (D_1 \cdots D_j = B_1 \cdots B_i).$$

Let a_i denote the length of A_i; similarly for b_i and d_i. Let $\vec{a}_i$ denote the subsequence $(a_1, \cdots, a_i)$, $1 \leq i \leq N$. Similarly define $\vec{b}_i$ and $\vec{d}_i$. The problem at hand now can be stated as: given the *multisets* $\tilde{a} = \{a_1, \cdots, a_N\}$, $\tilde{b} = \{b_1, \cdots, b_M\}$ and $\tilde{d} = \{d_1, \cdots, d_K\}$, construct the *sequences* $\vec{a}_N = (a_1, \cdots, a_N)$, $\vec{b}_M = (b_1, \cdots, b_M)$ and $\vec{d}_K = (d_1, \cdots, d_K)$.

Our basic algorithm generates $d_1, d_2, \cdots$ in order and extends the partitions for $\vec{a}$ and $\vec{b}$ using the following invariant property which can be obtained from the problem definition above. Either

- d_k is aligned with a_i, that is, $d_1 + \cdots + d_k = a_1 + \cdots + a_i$, or
- d_k is aligned with b_j (but not with a_i,[41]) that is, $d_1 + \cdots + d_k = b_1 + \cdots + b_j$.

In the program below, the main procedure `solve` takes as input three lists representing $\tilde{a}, \tilde{b}$ and $\tilde{d}$ in the first three arguments, and outputs in the remaining three arguments. Enumeration is done by choosing, at each recursive step of the `rsm` procedure, one of two cases mentioned above. Hence the two rules for `rsm`. Note that the three middle arguments of `rsm` maintains the length of the subsequences found so far, and in all calls, either `lenA = lenD < lenB` or `lenB = lenD < lenA` holds; the procedure `choose_initial` chooses the first fragment, and makes the first call to `rsm` with this invariant holding. Finally, the procedure `choose` deletes some element from the given list and returns the resultant list. Note that one more rule for `rsm` is needed in case the A and B fragments do align anywhere except at the extreme ends; we have omitted this possibility for simplicity.

```
solve(A, B, D, [AFrag|MapA], [BFrag|MapB], [DFrag|MapD]) :-
    choose_initial(A, B, D, AFrag, BFrag, DFrag, A2, B2, D2),
    rsm(A2, B2, D2, AFrag, BFrag, DFrag, MapA, MapB, MapD).
rsm(A, B, D, LenA, LenB, LenD, MapA, MapB, MapD) :-
    empty(A), empty(B), empty(D),
    MapA = [], MapB = [], MapD = [].
rsm(A, B, D, LenA, LenB, LenD, [Ai|MapA], MapB, [Dk|MapD]) :-
    LenA = LenD, LenA < LenB
    Dk <= LenB - LenA, Ai >= Dk,
    choose(Dk, D, D2),
    choose(Ai, A, A2),
    rsm(A2, B, D2, LenA + Ai, LenB, LenD + Dk, MapA, MapB, MapD).
rsm(A, B, D, LenA, LenB, LenD, MapA, [Bj|MapB], [Dk|MapD]) :-
    LenB = LenD, LenB < LenA
```

[41] For simplicity we assume that we never have all three partitions aligned except at the beginning and at the end.

```
        Dk <= LenA - LenB, Bj >= Dk,
        choose(Dk, D, D2),
        choose(Bj, B, B2),
        rsm(A, B2, D2, LenA, LenB + Bj, LenD + Dk, MapA, MapB, MapD).
```

This application of CLP is due to Yap [264, 265] and it is important to note that the above program is a considerable simplification of Yap's program. A major omission is the consideration of errors in the fragment lengths (because these lengths are obtained from experimentation). A major point in Yap's approach is that it gives a robust and uniform treatment of the experimental errors inherent in the data as compared with many of the approaches in the literature. Furthermore, [265] shows how the simple two enzyme problem can be extended to a number of other problem variations. Because a map solution is just a set of answer constraints returned by the algorithm, it is easy to combine this with other maps, compare maps, verify maps, etc. This kind of flexibility is important as the computational problem of just computing a consistent map is intractable and hence when dealing with any substantial amount of data would have to take into account data from many varieties of mapping experiments as well as other information specific to the molecule in question.

13.3. Scheduling

In this class of problems, we are given a number of tasks, and for each task, a task duration. Each task also requires other resources to be performed, and there are constraints on precedences of task performance, and on resource usage. The problem is to schedule the tasks so that the resources are most efficiently used (for example, perform the tasks so that all are done as soon as possible).

Consider now a basic *job-shop* scheduling problem in which is given a number m of machines, j sequences of tasks, the task durations and the machine assigned to each task. The precedence constraints are that the tasks in each sequence (called a job) are performed in the sequence order. The resource constraints are that each machine performs at most one task at any given time.

In the program below, `precedences` sets up the precedence constraints for one job, and is called with two equally long lists. The first contains the task variables, whose values are the start times. The second list contains the durations of the tasks. Thus `precedences` is called once for each job. The procedure `resources` is called repeatedly, once for each pair of tasks T_1 and T_2 which must performed without overlapping; their durations are given by D_1 and D_2.

```
precedences([T1, T2 | Tail], [D1, D2 | Tail2]) :-
    T1 + D1 <= T2,
    precedences(Tail, Tail2).
precedences([], []).

resources(T1, D1, T2, D2) :- T1 + D1 <= T2.
resources(T1, D1, T2, D2) :- T2 + D2 <= T1.
```

A simple way to proceed is to fix an ordering of the tasks performed on each machine. This corresponds to choosing one of the two `resources` rules for each pair of tasks assigned to the same machine. This forms the basis of the `enumerate` procedure below. Once an ordering of tasks is fixed, it is a simple matter to determine the best start times for each task.

This can be done in the manner indicated in the `solve` procedure below. An important efficiency point is that by choosing a precedence between two tasks, the new constraints created by the use of `resources`, in conjunction with the precedence constraints, can reduce the number of possible choices for the remaining pairs. We assume that the procedure `define_cost` defines Cost in such a way that, in conjunction with other constraints, it provides a conservative lower bound of the real cost of the schedule determined so far. Its precise definition, omitted here, can be obtained in a similar way as in the second program of the cutting-stock example above.

```
solve(T1, T2, ... , Tn, Cost) :-
    precedences( ... ), % one per job
    ...
    precedences( ... ),
    define_cost(T1, T2, ... , Tn, Cost),
    enumerate(T1, T2, ... , Tn),
    generate_start_times(T1, T2, ... , Tn).
enumerate(T1, T2, ... , Tn) :-
    resources( ... ), % one per pair of tasks assigned to same machine
    ...
    resources( ... ).
```

Finally, this `solve` procedure can be repeatedly run, within a branch-and-bound framework (with a special `minimize` predicate mechanism as explained above) to obtain the best solution over all possible orderings.

In this presentation of the program we have chosen to simply list all calls to `precedences` in the procedure `solve`, to focus on the important procedures in the program. A real program would use an auxilliary predicate to iterate over the jobs and generate the calls to `precedences`. Similarly, `enumerate` would iterate to generate calls to `resources`. Thus the program would be independent of the number of jobs or the pattern in which tasks are assigned to machines. Similar comments apply to other programs in this section.

There are variations and specializations of CLP approaches to this problem. Section 5.4.2 of [107] and section 2 of [78], on which this presentation is based, further discuss the problem and how particular features of CHIP can be useful. Another CHIP approach, but this time to a specific and practical scheduling problem is reported in [50]. In [2], the focus is on a new feature of CHIP and how it can be used to obtain an optimal solution to a particular 10 jobs and 10 machine problem, which remained open until recently.

Real scheduling problems can involve more kinds of constraints than just those mentioned above. For example, one could require that there is at most a certain time elapsed between the completion of one task and the commencement of another. See [260] for a more complete discussion of the CLP approach to the general scheduling problem.

13.4. Chemical Hypothetical Reasoning

This Prolog III application, described in some detail in [139], uses both arithmetic and boolean constraints. The problem at hand is that of elucidating chemical-reaction pathways, and we quote [139]: given an *instantiation* of the (two-reagent) reaction schema $A + B \leadsto T + P_1 + \cdots + P_k$, determine the *pathway*, that is, the set of constituent reaction *steps*, as well as other molecules (or *species*) formed during the reaction.

The reaction step considered in [139] contains at most two reactant molecules, and at most two product molecules, and so can be described in the form $R_1 + R_2 \longrightarrow P_1 + P_2$ where R_1, R_2, P_1, P_2 are (possibly empty) molecular formulas. The problem then is to determine, given an overall reaction, a collection of basic steps or pathway which explain the overall reaction. For example, given $C_7H_9N + CH_20 \leadsto C_{17}H_{18}N_2 + H_2O$, the following is a pathway which explains the reaction.

$$\begin{array}{l} C_7H_9N \longrightarrow H_2O + C_8H_9N \\ C_8H_9N + C_8H_9N \longrightarrow C_{16}H_{18}N_2 \\ C_8H_9N + C_{16}H_{18}N_2 \longrightarrow C_{17}H_{18}N_2 + C_7H_9N \end{array}$$

Here C_8H_9N and $C_{16}H_{18}N_2$ are the previously unidentified species.

The program imposes constraints to express requirements for a chemical reaction and to exclude uninteresting reactions. In addition to the constraints on the number of molecules, there are two other constraints on reaction steps: for each chemical element, the number of reactant atoms equals the number of product atoms (i.e. the step is *chemically balanced*), and no molecular formula appears in both sides of a step.

There are also constraints on the pathway. Let the reaction schema under consideration be $A + B \leadsto T + P_1 + \cdots + P_k$. Then

- All pathway species must be formable from the two reagents A and B.
- Neither A nor B alone is sufficient to form the target product T. Here boolean variables are used to express the dependency relation "can be formed from". For each pathway step $R_1, R_2 \longrightarrow P_1, P_2$ we state the boolean constraint $a_1 \wedge a_2 \Longrightarrow a_3 \wedge a_4$ where a_1, a_2, a_3, a_4 are boolean variables associated with R_1, R_2, P_1, P_2 respectively. The constraint expresses that both P_1 and P_2 can be formed if both R_1 and R_2 can be formed. Let $\mathcal{B}$ denote the boolean formulas thus constructed over all the steps in a pathway. Then expressing that species R does not, by itself, produce species P is tantamount to the satisfiability of the boolean constraint $\mathcal{B} \wedge \neg(a_R \Longrightarrow a_P)$, where a_R and a_P are the boolean variables associated with R and P respectively. Since we have two original reagents, we will need two sets of boolean variables and two sets of dependency constraints, to avoid any interference between the two conditions.
- There is a notion of *pathway consistency* which is defined to be the satisfiability of a certain arithmetic formula constructed from the occurrences of species in the pathway. Essentially this formula is a conjunction of formulas $n_1 + n_2 = n_3 + n_4$, for each pathway step $R_1 + R_2 \longrightarrow P_1 + P_2$, where $n_1, \cdots, n_4$ are the arithmetic variables of R_1, R_2, P_1, P_2 respectively.
- Finally, in the ultimate output of the program, no two pathways are identical, nor become identical under transformations such as permuting the reactants or products within a step, or switching the reactants and products in a step.

The program representation of a molecular formula is as a list of numbers, each of which specifies the number of atoms of a certain chemical element. We shall assume that there are only four chemical elements of interest in our presentation, and hence a molecular formula is a 4-tuple. A species is also represented by a 4-tuple (n, a, b, f) where n is an arithmetic variable (to be used in the formulation of the arithmetic formula mentioned above), a and b are boolean variables (to be used in expressing the formation dependencies), and f is the species formula. A step $R_1 + R_2 \longrightarrow P_1 + P_2$ is represented by a 4-tuple (r_1, r_2, p_1, p_2) containing the identifiers of the representations of R_1, R_2, P_1 and P_2.

The listing below is simplified and translated version of the Prolog III program in [139]. In the main procedure `solve`, the first argument is a list of fixed size, say n, in which each element is a species template. The first three templates are given, and these represent the two initial reagents R_1, R_2 and final target T. Similarly, `Steps` is a list of fixed size, say m, in which each element is a step template. Thus n and m are parameters to the program. The undefined procedure `formula_of` obtains the species formula from a species, that is, it projects onto the last element of the given 4-tuple. Similarly, `arith_var_of`, `bool_var_a_of` and `bool_var_b_of` project onto the first, second and third arguments respectively.

The procedure `no_duplicates` asserts constraints which prevent duplicate species and steps, and it also prevents symmetrical solutions; we omit the details. Calls to the procedure `formation_dependencies` generate the formation dependencies. The procedure `both_reagents_needed` imposes two constraints, one for each reagent, that, in conjunction with the formation dependencies, assert that R_1 (respectively R_2) alone cannot produce T. Finally, `enumerate_species` is self-explanatory.

```
solve([R1, R2, T | Species], Steps) :-
    no_duplicates( ... ),
    balanced_step( ... ), % for each step in Steps
    pathway_step_consistency( ... ), % for each step in Steps
    formation_dependencies( ... ), % for each step in Steps
    both_reagents_needed(R1, R2, T),
    enumerate_species( ... ).
balanced_step(R1, R2, P1, P2) :-
    formula_of(R1, (C1, H1, N1, O1)), formula_of(R2, (C2, H2, N2, O2)),
    formula_of(P1, (C3, H3, N3, O3)), formula_of(P2, (C4, H4, N4, O4)),
    C1 + C2 = C3 + C4,
    H1 + H2 = H3 + H4,
    N1 + N2 = N3 + N4,
    O1 + O2 = O3 + O4.
pathway_step_consistency(R1, R2, P1, P2) :-
    arith_var_of(R1, N1), arith_var_of(R1, N2),
    arith_var_of(P1, N3), arith_var_of(P1, N4),
    N1 + N2 = N3 + N4.
formation_dependencies(R1, R2, P1, P2) :-
    bool_var_a_of(R1, A1), bool_var_b_of(R1, B1),
    bool_var_a_of(R1, A2), bool_var_b_of(R2, B2),
    bool_var_a_of(P1, A3), bool_var_b_of(P1, B3),
    bool_var_a_of(P1, A4), bool_var_b_of(P2, B4),
    A1 ∧ A2 ⟹ A3 ∧ A4,
```

```
    B1 ∧ B2 ⟹ B3 ∧ B4.
both_reagents_needed(R1, R2, T) :-
    bool_var_a_of(R1, A1), bool_var_b_of(R1, B1),
    bool_var_a_of(R2, A2), bool_var_b_of(R2, B2),
    bool_var_a_of(T, A3), bool_var_b_of(T, B3),
    ¬ (R1 ⟹ T),
    ¬ (R2 ⟹ T).
```

13.5. Propositional Solver

As mentioned above in the discussion about the boolean constraint domain, one approach to solving boolean equations is to use `clp(FD)`, representing the input formulas in a straightforward way using variables constrained to be 0 or 1. See section 3.3.2 of [225] and [56] for example. What follows is from [56].

Assuming, without losing generality, that the input is a conjunction of equations of the form $Z = X \wedge Y$, $Z = X \vee Y$ or $X = \neg Y$, the basic algorithm is simply to represent each equation

$Z = X \wedge Y$	by the $\mathcal{FD}$ constraints	$Z = X \times Y$
		$Z \leq X \leq Z \times Y + 1 - Y$
		$Z \leq Y \leq Z \times X + 1 - X$
$Z = X \vee Y$	by	$Z = X + Y - X \times Y$
		$Z \times (1 - Y) \leq X \leq Z$
		$Z \times (1 - X) \leq X \leq Z$
$X = \neg Y$	by	$X = 1 - Y$
		$Y = 1 - X$

The following is a `clp(FD)` program fragment which realizes these representations. What is not shown is a procedure which takes the input equation and calls the `and`, `or` and `not` procedures appropriately, and an enumeration procedure (over the values 0 and 1) for all variables. In this program $val(X)$ delays execution of an $\mathcal{FD}$ constraint containing it until X is ground, at which time $val(X)$ denotes the value of X. The meaning of $min(X)$ and $max(X)$ are, respectively, the current lower and upper bounds on X maintained by the constraint solver, as discussed in Section 9.3. A constraint X *in* $s..t$ expresses that s and t are, respectively, lower and upper bounds for X.

```
and(X, Y, Z) :-
    Z in min(X)*min(Y) .. max(X)*max(Y),
    X in min(Z) .. max(Z)*max(Y) + 1 - min(Y),
    Y in min(Z) .. max(Z)*max(X) + 1 - min(X).
or(X, Y, Z) :-
    Z in min(X) + min(Y) - min(X)*min(Y) .. max(X) + max(Y) - max(X)*max(Y),
    X in min(Z)*(1 - max(Y)) .. max(Z),
    Y in min(Z)*(1 - max(X)) .. max(Z).
not(X, Y) :-
    X in 1 - val(Y),
    Y in 1 - val(X).
```

We conclude here by mentioning the authors' claim that this approach has great efficiency. In particular, it is several times faster than each of two boolean solvers deployed in CHIP, and some special-purpose stand-alone solvers.

14. Further Applications

The applications discussed in the previous two sections are but a sample of CLP applications. Here we briefly mention some others to indicate the breadth of problems that have been addressed using CLP languages and techniques.

We have exemplified the use of CLP to model analog circuits above. A considerable amount of work has also been done on digital circuits, in particular, on verification [226, 228, 231, 232], diagnosis [229], synthesis [230] and test-pattern generation [227]. Much of these works used the CHIP system. See also [88] for a description of a large application. In civil engineering, [155] used CLP($\mathcal{R}$) for the analysis and partial synthesis of truss structures. As with electrical circuits, the constraints implement physical modelling and are used to verify truss and support components, as well as to generate spatial configurations. There is also work in mechanical engineering; [239] used CLP($\mathcal{R}$) to design gear boxes, and [245] combined techniques from qualitative physics and CLP($\mathcal{R}$) to design mechanical systems from behaviour specifications. In general, engineering applications such as these use CLP to specify a hierarchical composition of complex systems and for rule-based reasoning.

Another important application area for CLP is finance. We mentioned the OTAS work above. Some further work is [119] which also deals with option valuations, and [23, 24, 37] which deal with financial planning. These financial applications have tended to take the form of expert systems involving sophisticated mathematical models.

There have been various proposals for including certainty measures and probabilities in logic programs to provide some built-in evidential reasoning that can be useful when writing expert systems. Original proposals [222, 83] intended Prolog as the underlying language, but it is clear that CLP languages provide for more flexible execution of such expert systems.

The Applause Project [167] has developed applications that use the ElipSys system – a parallel implementation of a CLP language – for manufacturing planning, tourist advice, molecular biology, and environment monitoring and control.

Finally, we mention work on applying CLP languages to: music [251], car sequencing [109], aircraft traffic control [57], building visual language parsers [105], a warehousing problem [25], safety analysis [68], frequency assignment for cellular telephones [47], timetabling [31], floor planning [140], spacecraft attitude control [233], interoperability of fibre optic communications equipment [49], interest rate risk management in banking [97], failure mode and effect analysis of complex systems [97], development of digitally controlled analog systems [197], testing of telecommunication protocols [154], causal graph management [209], factory scheduling [85], etc. The Applause Project [167] has developed applications that use the ElipSys system for manufacturing planning, tourist advice, molecular biology, and environment monitoring and control.

Acknowledgements

We would like to thank the following people for their comments on drafts of this paper and/or help in other ways: M. Bruynooghe, N. Heintze, P. van Hentenryck, A. Herold, J-L. Lassez, S. Michaylov, C. Palamidessi, K. Shuerman, P. Stuckey, M. Wallace, R. Yap. We also thank the anonymous referees for their careful reading and helpful comments.

REFERENCES

1. M. Abadi & Z. Manna, Temporal Logic Programming, *Journal of Symbolic Computation*, 8, 277–295, 1989.
2. A. Aggoun & N. Beldiceanu, Extending CHIP to Solve Complex Scheduling and Packing Problems, In *Journées Francophones De Programmation Logique*, Lille, France, 1992.
3. A. Aggoun & N. Beldiceanu, Overview of the CHIP Compiler System, in: *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 421–435, 1993.
4. A. Aiba, K. Sakai, Y. Sato, D. Hawley & R. Hasegawa, Constraint Logic Programming Language CAL, *Proc. International Conference on Fifth Generation Computer Systems 1988,* 263–276, 1988.
5. H. Aït-Kaci, An Algebraic Semantics Approach to the Effective Resolution of Type Equations, *Theoretical Computer Science 45*, 293–351, 1986.
6. H. Aït-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, 1991.
7. H. Aït-Kaci & R. Nasr, LOGIN: A Logic Programming Language with Built-in Inheritance, *Journal of Logic Programming 3*, 185–215, 1986.
8. H. Aït-Kaci, P. Lincoln & R. Nasr, Le Fun: Logic Equations and Functions, *Proc. Symposium on Logic Programming*, 17–23, 1987.
9. H. Aït-Kaci & A. Podelski, Towards a Meaning of LIFE, *Journal of Logic Programming*, 16, 195–234, 1993.
10. H. Aït-Kaci & A. Podelski, Entailment and Disentailment of Order-Sorted Feature Constraints, manuscript, 1993.
11. H. Aït-Kaci & A. Podelski, A General Residuation Framework, manuscript, 1993.
12. H. Aït-Kaci, A. Podelski & G. Smolka, A Feature-based Constraint System for Logic Programming with Entailment, *Theoretical Computer Science*, to appear. Also in: *Proc. International Conference on Fifth Generation Computer Systems 1992,* Vol. 2, 1992, 1012–1021.
13. L. Albert, R. Casas & F. Fages, Average-case Analysis of Unification Algorithms, *Theoretical Computer Science* 113, 3–34, 1993.
14. K. Apt, H. Blair & A. Walker, Towards a Theory of Declarative Knowledge, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan Kaufmann, 89-148, 1988.
15. C. Atay, A Parallelization of the Constraint Logic Programming Language 2LP, Ph.D. thesis, City University of New York, 1992.
16. C. Atay, K. McAloon & C. Tretkoff, 2LP: A Highly Parallel Constraint Logic Programming Language, *Proc. 6th. SIAM Conf. on Parallel Processing for Scientific Computing*, 1993.
17. R. Barbuti, M. Codish, R. Giacobazzi & M.J. Maher, Oracle Semantics for Prolog, *Proc. 3rd Conference on Algebraic and Logic Programming,* LNCS 632, 100–115, 1992.
18. M. Baudinet, Proving Termination Properties of Prolog: A Semantic Approach, *Proc. 3rd. Symp. Logic in Computer Science*, 334–347, 1988.

19. M. Baudinet, J. Chomicki & P. Wolper, Temporal Deductive Databases, in *Temporal Databases: Theory, Design and Implementation*, A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (Eds), Benjamin/Cummings, 1993.
20. F. Benhamou & A. Colmerauer (Eds.), *Constraint Logic Programming: Selected Research*, MIT Press, 1993.
21. F. Benhamou, Boolean Algorithms in PROLOG III, in: *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 307–325, 1993.
22. F. Benhamou & J-L. Massat, Boolean Pseudo-equations in Constraint Logic Programming, *Proc. 10th International Conference on Logic Programming*, 517–531, 1993.
23. F. Berthier, A Financial Model using Qualitative and Quantitative Knowledge, *Proceeedings of the International Symposium on Computational Intelligence 89*, Milano, 1–9, September 1989.
24. F. Berthier, Solving Financial Decision Problems with CHIP, *Proceeedings of the 2nd Conference on Economics and Artificial Intelligence - CECIOA 2*, Paris, 233–238, June 1990.
25. R. Bisdorff & S. Laurent, Industrial Disposing Problem Solved in CHIP, *Proc. 10th International Conference on Logic Programming*, 831, 1993.
26. A. Bockmayr, Logic Programming with Pseudo-Boolean Constraints, in: *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 327–350, 1993.
27. F. de Boer, J. Kok, C. Palamidessi & J. Rutten, Non-monotonic Concurrent Constraint Programming, *Proc. International Logic Programming Symposium,* 315–334, 1993.
28. F.S. de Boer & C. Palamidessi, A Fully Abstract Model for Concurrent Constraint Programming, *Proc. of TAPSOFT/CAAP*, LNCS 493, 296–319, 1991.
29. F.S. de Boer & C. Palamidessi, Embedding as a Tool for Language Comparison, *Information and Computation*, to appear.
30. F.S. de Boer & C. Palamidessi, *From Concurrent Logic Programming to Concurrent Constraint Programming*, in: *Advances in Logic Programming Theory*, Oxford University Press, to appear.
31. P. Boizumault, Y. Delon & L. Peridy, Solving a real life exams problem using CHIP, *Proc. International Logic Programming Symposium,* 661, 1993.
32. A. Borning, The Programming Language Aspects of ThingLab, a Constraint–Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems*, 3(4), 252–387, October 1981.
33. A. Borning, M.J. Maher, A. Martindale & M. Wilson, Constraint Hierarchies and Logic Programming, *Proc. 6th International Conference on Logic Programming,* 149–164, 1989. Fuller version as Technical Report 88-11-10, Computer Science Department, University of Washington, 1988.
34. A. Bossi, M. Gabbrielli, G. Levi & M.C. Meo, Contributions to the Semantics of Open Logic Programs, *Proc. Int. Conf. on Fifth Generation Computer Systems*, 570–580, 1992.
35. A. Brodsky & Y. Sagiv, Inference of Inequality Constraints in Logic Programs, *Proc. ACM Symp. on Principles of Database Systems*, 1991.
36. A. Brodsky, J. Jaffar & M. Maher, Toward Practical Constraint Databases, *Proc. 19th International Conference on Very Large Data Bases*, 567–580, 1993.
37. J.M. Broek & H.A.M. Daniels, Application of CLP to Asset and Liability Management in Banks, *Computer Science in Economics and Management*, 4(2), 107–116, May 1991.
38. R. Bryant, Graph Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers* 35, 677–691, 1986.

39. C. Brzoska, Temporal Logic Programming and its Relation to Constraint Logic Programming, *Proc. International Logic Programming Symposium,* 661–677, 1991.
40. C. Brzoska, Temporal Logic Programming with Bounded Universal Modality Goals, *Proc. 10th International Conference on Logic Programming,* 239–256, 1993.
41. J. Burg, C. Hughes, J. Moshell & S.D. Lang, Constraint-based Programming: A Survey, Technical Report IST-TR-90-16, Dept. of Computer Science, University of Central Florida, 1990.
42. J. Burg, Parallel Execution Models and Algorithms for Constraint Logic Programming over a Real-number Domain, Ph.D. thesis, Dept. of Computer Science, University of Central Florida, 1992.
43. J. Burg, C. Hughes & S.D. Lang, Parallel Execution of CLP-$\Re$ Programs, Technical Report TR-CS-92-20, University of Central Florida, 1992.
44. H-J. Bürckert, A Resolution Principle for Clauses with Constraints, *Proc. CADE-10*, LNCS 449, 178–192, 1990.
45. W. Büttner & H. Simonis, Embedding Boolean Expressions into Logic Programming, *Journal of Symbolic Computation*, 4, 191–205, 1987.
46. M. Carlsson, Freeze, Indexing and other Implementation Issues in the WAM, *Proc. 4th International Conference on Logic Programming*, 40–58, 1987.
47. M. Carlsson & M. Grindal, Automatic Frequency Assignment for Cellular Telephones Using Constraint Satisfaction Techniques, *Proc. 10th International Conference on Logic Programming*, 647–665, 1993.
48. S.N. Černikov, Contraction of Finite Systems of Linear Inequalities (In Russian), *Doklady Akademiia Nauk SSSR*, Vol. 152, No. 5, 1075–1078, 1963. (English translation in *Soviet Mathematics Doklady*, Vol. 4, No. 5, 1520–1524, 1963.)
49. R. Chadra, O. Cockings & S. Narain, Interoperability Analysis by Symbolic Simulation, *Proc. JICSLP Workshop on Constraint Logic Programming*, 55–58, 1992.
50. A. Chamard, F. Decès & A. Fischler, Applying CHIP to a Complex Scheduling Problem, draft manuscript, Dassualt Aviation, Department of Artificial Intelligence, 1992.
51. D. Chan, Constructive Negation based on Completed Database, *Proc. 5th International Conference on Logic Programming*, 111–125, 1988.
52. V. Chandru, Variable Elimination in Linear Constraints, *The Computer Journal*, 36(5), 463–472, 1993.
53. V. Chandru & J.N. Hooker, Extended Horn Sets in Propositional Logic, *Journal of the ACM*, 38, 205–221, 1991.
54. V. Chvâtal, *Linear Programming*, W.H. Freeman and Co., New York, 1983.
55. K.L. Clark, Negation as Failure, in *Logic and Databases*, H. Gallaire and J. Minker (Eds.), Plenum Press, New York, 293–322, 1978.
56. P. Codognet & D. Diaz, Boolean Constraint Solving using `clp(FD)`, *Proc. International Logic Programming Symposium*, 525–539, 1993.
57. P. Codognet, F. Fages, J. Jourdan, R. Lissajoux & T. Sola, On the Design of Meta(F) and its Applications in Air Traffic Control, *Proc. JICSLP Workshop on Constraint Logic Programming*, 28–35, 1992.
58. J. Cohen, Constraint Logic Programming Languages, *CACM*, 33, 52–68, July 1990.
59. A. Colmerauer, Prolog-II Manuel de Reference et Modele Theorique, Groupe Intelligence Artificelle, U. d'Aix-Marseille II, 1982.
60. A. Colmerauer, Prolog and Infinite Trees, in *Logic Programming*, K.L. Clark and S-A. Tarnlund (Eds), Academic Press, New York, 231–251, 1982.
61. A. Colmerauer, Prolog in 10 Figures, *Proc. 8th International Joint Conference*

on Artificial Intelligence, 487–499, 1983.

62. A. Colmerauer, Equations and Inequations on Finite and Infinite Trees, *Proc. 2nd. Int. Conf. on Fifth Generation Computer Systems*, Tokyo, 85–99, 1984.
63. A. Colmerauer, Opening the Prolog III Universe, *BYTE Magazine*, August 1987.
64. A. Colmerauer, Prolog III Reference and Users Manual, Version 1.1, PrologIA, Marseilles, 1990.
65. A. Colmerauer, An Introduction to Prolog III, *CACM*, 33, 69–90, July 1990.
66. A. Colmerauer, Naive Solving of Non-linear Constraints, in: *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 89–112, 1993.
67. A. Colmerauer, invited talk at *Workshop on the Principles & Practice of Constraint Programming*, Newport, RI, April 1993.
68. M-M. Corsini & A Rauzy, Safety Analysis by means of Fault Trees: an Application for Open Boolean Solvers, *Proc. 10th International Conference on Logic Programming*, 834, 1993.
69. B. Courcelle, Fundamental Properties of Infinite Trees, *Theoretical Computer Science*, 25(2), 95–169, March 1983.
70. J. Darlington and Y-K. Guo, A New Perspective on Integrating Functions and Logic Languages, *Proceedings of the 3rd International Conference on Fifth Generation Computer Systems*, Tokyo, 682–693, 1992.
71. B. De Backer & H. Beringer, Intelligent Backtracking for CLP Languages, An Application to CLP($\mathcal{R}$), *Proc. International Logic Programming Symposium*, 405–419, 1991.
72. B. De Backer & H. Beringer, A CLP Language Handling Disjunctions of Linear Constraints, *Proc. 10th International Conference on Logic Programming,* 550–563, 1993.
73. S. K. Debray, Static Inference of Modes and Data Dependencies in Logic Programs, *ACM Transactions on Programming Languages and Systems* **11** (3), 418–450, 1989.
74. S. K. Debray & D.S. Warren, Functional Computations in Logic Programs, *ACM Transactions on Programming Languages and Systems* **11** (3), 451–481, 1989.
75. D. Diaz & P. Codognet, A Minimal Extension of the WAM for `clp(FD)`, *Proc. 10th International Conference on Logic Programming,* 774–790, 1993.
76. M. Dincbas, P. Van Hentenryck, H. Simonis, & A. Aggoun, The Constraint Logic Programming Language CHIP, *Proceedings of the 2nd. International Conference on Fifth Generation Computer Systems*, 249–264, 1988.
77. M. Dincbas, H. Simonis & P. van Hentenryck, Solving a Cutting-stock Problem in CLP, *Proceedings* 5^{th} *International Conference on Logic Programming*, MIT Press, 1988.
78. M. Dincbas, H. Simonis & P. Van Hentenryck, Solving Large Combinatorial Problems in Logic Programming, *Journal of Logic Programming* 8 (1&2), 75–93, 1990.
79. A. Dovier & G. Rossi, Embedding Extensional Finite Sets in CLP, *Proc. International Logic Programming Symposium*, 540–556, 1993.
80. R. Duisburg, Constraint-based Animation: Temporal Constraints in the Animus System, Technical Report CR-86-37, Tektronix Laboratories, August 1986.
81. R. Ege, D. Maier & A. Borning, The Filter Browser: Defining Interfaces Graphically, *Proc. of the European Conf. on Object-Oriented Programming*, Paris, 155–165, 1987.
82. E. Elcock, Absys: The First Logic Programming Language – A Retrospective and Commentary, *Journal of Logic Programming*, 9, 1–17, 1990.
83. M. van Emden, Quantitative Deduction and its Fixpoint Theory, *Journal of Logic Programming*, 37–53, 1986.

84. E. Emerson, Temporal and Modal Logic, in: *Handbook of Theoretical Computer Science, Vol. B*, Chapter 16, 995–1072, 1990.
85. O. Evans, Factory Scheduling using Finite Domains, in: *Logic Programming in Action*, LNCS 636, Springer-Verlag, 45–53, 1992.
86. F. Fages, On the Semantics of Optimization Predicates in CLP Languages, *Proc. 13th Conf. on Foundations of Software Technology and Theoretical Computer Science*, 1993.
87. M. Falaschi, G. Levi, M. Martelli & C. Palamidessi, Declarative Modelling of the Operational Behavior of Logic Languages, *Theoretical Computer Science* 69, 289–318, 1989.
88. T. Filkorn, R. Schmid, E. Tidén & P. Warkentin, Experiences from a Large Industrial Circuit Design Application, *Proc. International Logic Programming Symposium*, 581–595, 1991.
89. R.E. Fikes, REF-ARF: A system for solving problems stated as procedures, *Artificial Intelligence* 1, 27–120, 1970.
90. M. Fitting, A Kripke-Kleene Semantics for Logic Programs, *Journal of Logic Programming*, 4, 295-312, 1985.
91. J-B.J. Fourier. Reported in: Analyse des travaux de l'Acadamie Royale des Sciences, pendant l'annee 1824, Partie mathematique, *Histoire de l'Academie Royale des Sciences de l'Institut de France*, Vol. 7, xlvii–lv, 1827. (Partial English translation in: D.A. Kohler. Translation of a Report by Fourier on his work on Linear Inequalities. *Opsearch*, Vol. 10, 38–42, 1973)
92. B.N. Freeman-Benson, Constraint Imperative Programming, PhD thesis, Department of Computer Science and Engineering, University of Washington, 1991.
93. T. Früwirth, Constraint Simplification Rules, Technical Report ECRC-92-18, ECRC, 1992.
94. T. Früwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, M. Wallace, Constraint Logic Programming – An Informal Introduction, in: *Logic Programming in Action*, LNCS 636, Springer-Verlag, 3–35, 1992.
95. M. Gabbrielli & G. Levi, Modeling Answer Constraints in Constraint Logic Programs, *Proc. 8th International Conference on Logic Programming,* 238–252, 1991.
96. H. Gaifman, M.J. Maher & E. Shapiro, Replay, Recovery, Replication and Snapshots of Nondeterministic Concurrent Programs, *Proc. 10th. ACM Symposium on Principles of Distributed Computation,* 1991.
97. P-J. Gailly, W. Krautter, C. Bisière & S. Bescos, The Prince project and its Applications, in: *Logic Programming in Action*, LNCS 636, Springer-Verlag, 54–63, 1992.
98. M. García de la Banda & M. Hermenegildo, A Practical Approach to the Global Analysis of Constraint Logic Programs, *Proc. International Logic Programming Symposium*, 437–455, 1993.
99. M. García de la Banda, M. Hermenegildo & K. Marriott, Independence in Constraint Logic Programs, *Proc. International Logic Programming Symposium*, 130–146, 1993.
100. A. van Gelder, Negation as Failure Using Tight Derivations for General Logic Programs, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan Kaufmann, 149–176, 1988.
101. A. van Gelder, K. Ross & J.S. Schlipf, Unfounded Sets and Well-Founded Semantics for General Logic Programs, Journal of the ACM, 38, 620–650, 1991.
102. M. Gelfond & V. Lifschitz, The Stable Model Semantics for Logic Programming, *Proc. 5th International Conference on Logic Programming*, 1070–1080, 1988.
103. W. Havens, S. Sidebottom, G. Sidebottom, J. Jones & R. Ovans, Echidna: A Constraint Logic Programming Shell, *Proc. Pacific Rim International Conference on Artificial Intelligence*, 1992.

104. N. Heintze, S. Michaylov & P.J. Stuckey, CLP($\mathcal{R}$) and Some Electrical Engineering Problems, *Journal of Automated Reasoning* 9, 231–260, 1992.
105. R. Helm, K. Marriott & M. Odersky, Building Visual Language Parsers, *Proc. CHI*, 1991.
106. R. Helm, K. Marriott & M. Odersky, Constraint-based Query Optimization for Spatial Databases, *Proc. 10th ACM Symp. on Principles of Database Systems*, 181–191, 1991.
107. P. van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.
108. P. van Hentenryck, Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys, *Proc. 6th International Conference on Logic Programming*, 165–180, 1989.
109. P. van Hentenryck, Constraint Logic Programming, *The Knowledge Engineering Review* 6, 151–194, 1991.
110. P. van Hentenryck, Constraint Satisfaction using Constraint Logic Programming, *Artificial Intelligence* 58, 113–159, 1992.
111. P. van Hentenryck (Ed), Special Issue on Constraint Logic Programming, *Journal of Logic Programming*, 16, 3&4, 1993.
112. P. van Hentenryck & Y. Deville, The Cardinality Operator: A New Logical Connective and its Application to Constraint Logic Programming, *Proc. International Conference on Logic Programming*, 745–759, 1991.
113. P. van Hentenryck & Y. Deville, Operational Semantics of Constraint Logic Programming over Finite Domains, *Proc. Symp. on Programming Language Implementation and Logic Programming*, LNCS 528, 395–406, 1991.
114. P. van Hentenryck & T. Graf, Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming, *Annals of Mathematics and Artificial Intelligence* 5, 303–319, 1992.
115. P. van Hentenryck, V. Saraswat & Y. Deville, Constraint Processing in cc($\mathcal{FD}$), manuscript, 1991.
116. P. van Hentenryck, V. Saraswat & Y. Deville, Design, Implementations and Evaluation of the Constraint Language cc($\mathcal{FD}$), Technical Report CS-93-02, Brown University, 1993.
117. T. Hickey, Functional Constraints in CLP Languages, in: *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 355–381, 1993.
118. M. Höhfeld & G. Smolka, Definite Relations over Constraint Languages, LILOG Report 53, IBM Deutschland, 1988.
119. D. Homiak, A CLP System for Solving Partial Differential Equations with Applications to Options Valuation, Masters Project, DePaul University, 1991.
120. H. Hong, RISC-CLP(Real): Logic Programming with Non-linear Constraints over the Reals, in: *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 133–159, 1993.
121. A. Horn, On Sentences Which are True of Direct Unions of Algebras, *Journal of Symbolic Logic*, 16, 14–21, 1951.
122. T. Huynh & C. Lassez, A CLP($\mathcal{R}$) Options Trading Analysis System, *Proceedings 5th International Conference on Logic Programming*, 59–69, 1988.
123. T. Huynh, C. Lassez & J-L. Lassez. Practical Issues on the Projection of Polyhedral Sets. *Annals of Mathematics and Artificial Intelligence* 6, 295–315, 1992.
124. J-L. Imbert, Variable Elimination for Disequations in Generalized Linear Constraint Systems, *The Computer Journal 36*, 473–???, 1993.
125. J-L. Imbert, Fourier's Elimination: which to choose? *Proc. 1st Workshop on Principles and Practice of Constraint Programming*, Newport, 119–131, April 1993.

126. J. Jaffar, Efficient Unification over Infinite Terms, *New Generation Computing*, 2, 207–219, 1984.
127. J. Jaffar, Minimal and Complete Word Unification, *Journal of the ACM*, 37(1), 47–85, 1990.
128. J. Jaffar & J-L. Lassez, Constraint Logic Programming, Technical Report 86/73, Department of Computer Science, Monash University, 1986.
129. J. Jaffar & J-L. Lassez, Constraint Logic Programming, *Proc. 14th ACM Symposium on Principles of Programming Languages,* Munich (January 1987), 111–119.
130. J. Jaffar, J-L. Lassez & M.J. Maher, A Theory of Complete Logic Programs with Equality, *Journal of Logic Programming 1,* 211–223, 1984.
131. J. Jaffar, J-L. Lassez & M.J. Maher, A Logic Programming Language Scheme, in: *Logic Programming: Relations, Functions and Equations,* D. DeGroot and G. Lindstrom (Eds), Prentice-Hall, 441–467, 1986.
132. J. Jaffar, M.J. Maher, P.J. Stuckey & R.H.C. Yap, Projecting CLP($\Re$) Constraints, *New Generation Computing*, 11, 449–469, 1993.
133. J. Jaffar, S. Michaylov, P. Stuckey & R. Yap, The CLP($\mathcal{R}$) Language and System, *ACM Transactions on Programming Languages*, 14(3), 339–395, 1992.
134. J. Jaffar, S. Michaylov & R. Yap, A Methodology for Managing Hard Constraints in CLP Systems, *Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 306–316, 1991.
135. J. Jaffar, S. Michaylov, P. Stuckey & R.H.C. Yap, An Abstract Machine for CLP($\mathcal{R}$), *Proceedings ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 128–139, 1992.
136. J. Jaffar and P. Stuckey, Canonical Logic Programs, *Journal of Logic Programming* 3, 143–155, 1986.
137. S. Janson & S. Haridi, Programming Paradigms of the Andorra Kernel Language, *Proc. International Logic Programming Symposium*, 167–183, 1991.
138. N. Jorgensen, K. Marriott & S. Michaylov, Some Global Compile-time Optimizations for CLP($\mathcal{R}$), *Proceedings 1991 International Logic Programming Symposium*, 420–434, 1991.
139. J. Jourdan & R.E. Valdés-Pérez, Constraint Logic Programming Applied to Hypothetical Reasoning in Chemistry, *Proceedings North American Conference on Logic Programming*, 154–172, 1990. (Page 161 should follow page 166.)
140. K. Kanchanasut & C. Sumetphong, Floor Planning Applications in CLP($\mathcal{R}$), *Proc. JICSLP Workshop on Constraint Logic Programming*, 36–44, 1992.
141. L.G. Khachian, A polynomial algorithm in linear programming, *Soviet Math. Dokl.*, 20(1), 191–194, 1979.
142. J. de Kleer and G.J. Sussman, Propagation of Constraints Applied to Circuit Synthesis, *Circuit Theory and Applications* 8, 127–144, 1980.
143. P. Kanellakis, J-L. Lassez & V. Saraswat (Eds), *Principles and Practice of Constraint Programming*, MIT Press, to appear.
144. P. Kanellakis, G. Kuper & P. Revesz, Constraint Query Languages, *Journal of Computer and System Sciences,* to appear. Preliminary version appeared in *Proc. 9th ACM Symp. on Principles of Database Systems*, 299–313, 1990.
145. P. Kanellakis, S. Ramaswamy, D.E. Vengroff & J.S. Vitter, Indexing for Data Models with Constraints and Classes, *Proc. ACM Symp. on Principles of Database Systems*, 1993.
146. D. Kemp, K. Ramarohanarao, I. Balbin & K. Meenakshi, Propagating Constraints in Recursive Deductive Databases, *Proc. North American Conference on Logic Programming*, 981–998, 1989.
147. D. Kemp & P. Stuckey, Analysis based Constraint Query Optimization, *Proc. 10th International Conference on Logic Programming,* 666–682, 1993.
148. V. Klee & G.J. Minty, How good is the Simplex algorithm?, in: *Inequalities-III*,

O. Sisha (Ed), Academic Press, New York, 159–175, 1972
149. A. Klug, On Conjunctive Queries Containing Inequalities, *Journal of the ACM* 35, 1, 146–160, 1988.
150. A. Koscielski & L. Pacholski, Complexity of Unification in Free Groups and Free Semigroups, *Proc. 31st Symp. on Foundations of Computer Science*, 824–829, 1990.
151. R. Krishnamurthy, R. Ramakrishnan & O. Shmueli, A Framework for Testing Safety and Effective Computability of Extended Datalog, *Proc. ACM Symp. on Management of Data*, 154–163, 1988.
152. C.A.C. Kuip, Algebraic Languages for Mathematical Programming, *European Journal of Operations Research* 67, 25–51, 1993.
153. K. Kunen, Negation in Logic Programming, *Journal of Logic Programming*, 4, 289–308, 1987.
154. D. Ladret & M. Rueher, Contribution of Logic Programming to support Telecommunications Protocol Tests, *Proc. 10th International Conference on Logic Programming,* 845–846, 1993.
155. S. Lakmazaheri & W. Rasdorf, Constraint Logic Programming for the Analysis and Partial Synthesis of Truss Structures, *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing* 3(3), 157–173, 1989.
156. C. Lassez, Constraint Logic Programming: a Tutorial, in *BYTE Magazine*, August 1987.
157. C. Lassez, K. McAloon & R. Yap, Constraint Logic Programming and Options Trading, *IEEE Expert* 2(3), Special Issue on Financial Software, August 1987, 42-50.
158. C. Lassez & J-L. Lassez, Quantifier Elimination for Conjunctions of Linear Constraints via a Convex Hull Algorithm, in: *Symbolic and Numeric Computation for Artificial Intelligence*, B. Donald, D. Kapur and J.L. Mundy (Eds), Academic Press, to appear. Also, IBM Research Report RC16779, T.J. Watson Research Center, 1991.
159. J-L. Lassez, T. Huynh & K. McAloon, Simplifcation and Elimination of Redundant Linear Arithmetic Constraints, *Proc. North American Conference on Logic Programming*, Cleveland, 35–51, 1989.
160. J-L. Lassez, M. Maher & K.G. Marriott, Unification Revisited, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan Kaufmann, 587–625, 1988.
161. J-L. Lassez & K.G. Marriott, Explicit Representation of Terms Defined by Counter Examples, *Journal of Automated Reasoning*, 3, 301–317, 1987.
162. J-L. Lassez & K. McAloon, A Canonical Form for Generalized Linear Constraints, *Journal of Symbolic Computation*, to appear.
163. J-L. Lassez & K. McAloon, A Constraint Sequent Calculus, *Proc. of Symp. on Logic in Computer Science*, 52–62, 1990.
164. J-L. Lauriere, A Language and a Program for Stating and Solving Combinatorial Problems, *Artificial Intelligence* 10, 29–127, 1978.
165. W. Leler, *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, 1988.
166. H.F. Leung, *Distributed Constraint Logic Programming*, Vol. 41, World-Scientific Series in Computer Science, World-Scientific, 1993.
167. Liang-Liang et.al., APPLAUSE: Applications Using the ElypSys Parallel CLP System, *Proc. 10th International Conference on Logic Programming,* 847–848, 1993.
168. J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Second Edition, 1987.
169. J.W. Lloyd & R.W. Topor, Making Prolog More Expressive, *Journal of Logic*

Programming, 1, 93–109, 1984.
170. K. McAloon & C. Tretkoff, 2LP: A Logic Programming and Linear Programming System, Brooklyn College Computer Science Technical Report No 1989-21, 1989.
171. A. McDonald, P. Stuckey & R. Yap, Redundancy of Variables in CLP($\mathcal{R}$), *Proc. International Logic Programming Symposium*, 75–93, 1993.
172. J. McKinsey, The Decision Problem for Some Classes of Sentences Without Quantifiers, *Journal of Symbolic Logic,* 8, 61–76, 1943.
173. M.J. Maher, Logic Semantics for a Class of Committed-Choice Programs, *Proc. 4th International Conference on Logic Programming,* 858–876, 1987.
174. M.J. Maher, Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees, *Proc. 3rd. Symp. Logic in Computer Science*, 348–357, 1988. Full version: IBM Research Report, T.J. Watson Research Center.
175. M.J. Maher, A CLP View of Logic Programming, *Proc. Conf. on Algebraic and Logic Programming,* LNCS 632, 364–383, 1992.
176. M.J. Maher, A Transformation System for Deductive Database Modules with Perfect Model Semantics, *Theoretical Computer Science* 110, 377–403, 1993.
177. M.J. Maher, A Logic Programming View of CLP, *Proc. 10th International Conference on Logic Programming,* 737–753, 1993. Full version: IBM Research Report, T.J. Watson Research Center.
178. M.J. Maher & P.J. Stuckey, Expanding Query Power in CLP Languages, *Proc. North American Conference on Logic Programming,* 1989, 20–36.
179. G.S. Makanin, The Problem of Solvability of Equations in a Free Semigroup, *Math. USSR Sbornik* 32(2), 129–198, 1977. (English translation, AMS 1979).
180. H. Mannila & E. Ukkonen, On the Complexity of Unification Sequences, *Proc. 3rd International Conference on Logic Programming,* 122–133, 1986.
181. A. Mantsivoda, Flang and its Implementation, *Proc. Symp. on Programming Language Implementation and Logic Programming*, LNCS 714, 151–166, 1993.
182. K.G. Marriott & P.J. Stuckey, The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering, *Proc. 20th ACM Symp. Principles of Programming Languages*, 334–344, 1993.
183. K.G. Marriott & P.J. Stuckey, Semantics of CLP Programs with Optimization, Technical Report, University of Melbourne, 1993.
184. K. Marriott, H. Søndergaard, P.J. Stuckey, R.H.C. Yap, Optimising Compilation for CLP(R), *Proc. Australian Computer Science Conf.*, 1994.
185. U. Martin & T. Nipkow, Boolean Unification - The Story So Far, *Journal of Symbolic Computation*, 7, 275–293, 1989.
186. MathLab, *MACSYMA Reference Manual*, The MathLab Group, Laboratory for Computer Science, MIT, 1983.
187. S Michaylov & F. Pfenning, Higher-Order Logic Programming as Constraint Logic Programming, *1st. Workshop on Principles and Practice of Constraint Programming*, 1993.
188. S. Michaylov, Design and Implementation of Practical Constraint Logic Programming Systems, Ph.D. Thesis, Carnegie Mellon University, Report CMU-CS-92-16, August 1992.
189. D. Miller, A Logic Programming Language with Lambda-abstraction, Function Variables, and Simple Unification, in *Extensions of Logic Programming: International Workshop*, Springer-Verlag LNCS 475, 253–281, 1991.
190. D. Miller & G. Nadathur, Higher-Order Logic Programming, *Proc. 3rd International Conference on Logic Programming,* 448–462, 1986.
191. U. Montanari & F. Rossi, Graph Rewriting for a Partial Ordering Semantics of Concurrent Constraint Programming, *Theoretical Computer Science* 109, 225–256, 1993.
192. K. Mukai, Anadic Tuples in Prolog, Technical Report TR-239, ICOT, Tokyo,

1987.
193. I.S. Mumick, S.J. Finkelstein, H. Pirahesh & R. Ramakrishnan, Magic Conditions, *Proc. 9th ACM Symp. on Principles of Database Systems*, 314–330, 1990.
194. L. Naish, Automating Control for Logic Programs, *Journal of Logic Programming*, 2, 167–183, 1985.
195. L. Naish, *Negation and Control in PROLOG,* Lecture Notes in Computer Science 238, Springer-Verlag, 1986.
196. G. Nelson, JUNO, A Constraint Based Graphics System, *Computer Graphics* 19(3), 235–243, 1985.
197. A. Nerode & W. Kohn, Hybrid Systems and Constraint Logic Programming, *Proc. 10th International Conference on Logic Programming,* 18–24, 1993.
198. W. Older & F. Benhamou, Programming in CLP(BNR), *1st. Workshop on Principles and Practice of Constraint Programming*, 1993.
199. M.S. Paterson & M.N. Wegman, Linear Unification, *Journal of Computer and System Sciences* 16, 158–167, 1978.
200. F. Pfenning, Logic Programming in the LF Logical Framework, in: *Logical Frameworks*, G. Huet and G. Plotkin (Eds), Cambridge University Press, 149–181, 1991.
201. A. Podelski & P. van Roy, The Beauty and the Beast Algorithm: Testing Entailment and Disentailment Incrementally, draft manuscript, 1993.
202. T. Le Provost & M. Wallace, Generalized Constraint Propagation over the CLP Scheme, *Journal of Logic Programming*, 16, 319–359, 1993.
203. T. Przymusinski, On the Declarative Semantics of Deductive Databases and Logic Programs, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan Kaufmann, 193-216, 1988.
204. A. Rajasekar, String Logic Programs, draft manuscript, Dept. of Computer Science, Univ. of Kentucky, 1993.
205. V. Ramachandran & P. van Hentenryck, Incremental Algorithms for Constraint Solving and Entailment over Rational Trees, *Proc. 13th Conf. on Foundations of Software Technology and Theoretical Computer Science*, 1993.
206. R. Ramakrishnan, Magic Templates: A Spellbinding Approach to Logic Programs, *Journal of Logic Programming*, 11, 189–216, 1991.
207. G. Ramalingam & T. Reps, A Categorized Bibliography on Incremental Computation, *Proc. 17th ACM Symp. on Principles of Programming Languages*, 502–510, 1993.
208. P. van Roy & A.M. Despain, The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler, *Proceedings 1990 North American Conference on Logic Programming*, 501–515, 1990.
209. M. Rueher, A First Exploration of PrologIII's Capabilities, *Software–Practice and Experience* 23, 177–200, 1993.
210. K. Sakai, Y. Sato & S. Menju, Boolean Groebner Bases, to appear.
211. Y. Sagiv & M. Vardi, Safety of Datalog Queries over Infinite Databases, *Proc. ACM Symp. on Principles of Database Systems*, 160–171, 1989.
212. V. Saraswat, CP as a General-purpose Constraint-language, Proc. AAAI-87, 1987.
213. V. Saraswat, A Somewhat Logical Formulation of CLP Synchronization Primitives, *Proc. 5th International Conference Symposium on Logic Programming*, 1298–1314, 1988.
214. V. Saraswat, Concurrent Constraint Programming Languages, Ph.D. thesis, Carnegie-Mellon University, 1989. Revised version appears as *Concurrent Constraint Programming,* MIT Press, 1993.
215. V. Saraswat, The Category of Constraint Systems is Cartesian-Closed, *Proc. Symp. on Logic in Computer Science*, 341–345, 1992.

216. V. Saraswat, D. Weinbaum, K. Kahn, & E. Shapiro, Detecting Stable Properties of Networks in Concurrent Logic Programming Languages, *Proc. 7th. ACM Symp. Principles of Distributed Computing*, 210–222, 1988.
217. V. Saraswat, M. Rinard & P. Panangaden, Semantic Foundation of Concurrent Constraint Programming, *Proc. 18th ACM Symp. on Principles of Programming Languages*, 333–352, 1991.
218. V. Saraswat, A Retrospective Look at Concurrent Logic Programming, in preparation.
219. K. Satoh & A. Aiba, Computing Soft Constraints by Hierarchical Constraint Logic Programming, *Journal of Information Processing*, 7, 1993, to appear.
220. D. Scott, Domains for denotational semantics, *Proc. ICALP*, ??? LNCS 140, 1982.
221. E. Shapiro, A Subset of Concurrent Prolog and its Interpreter, Technical Report CS83-06, Dept of Applied Mathematics, Weizmann Institute of Science, 1983.
222. E. Shapiro, Logic Programs with Uncertainties: A Tool for Implementing Expert Systems, *Proc. 8th. IJCAI*, 529–532, 1983.
223. J.R. Shoenfield, *Mathematical Logic*, Addison-Wesley, 1967.
224. J. Siekmann, Unification Theory, *Journal of Symbolic Computation*, 7, 207–274, 1989.
225. H. Simonis & M. Dincbas, Propositional Calculus Problems in CHIP, in: *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 269–285, 1993.
226. H. Simonis, Formal Verification of Multipliers, *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, November 1989.
227. H. Simonis, Test Generation using the Constraint Logic Programming language CHIP, *Proc. 6th International Conference on Logic Programming*, 1989.
228. H. Simonis & M. Dincbas, Using an extended prolog for digital circuit design, *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, Munich, W.Germany, 165–188, October 1987.
229. H. Simonis & M. Dincbas, Using Logic Programming for Fault Diagnosis in Digital Circuits, *German Workshop on Artificial Intelligence (GWAI-87)*, Geseke, W. Germany, 139–148, September 1987.
230. H. Simonis & T. Graf, Technology Mapping in CHIP, Technical Report TR-LP-44, ECRC, Munich, 1990.
231. H. Simonis, H. N. Nguyen & M. Dincbas, Verification of digital circuits using CHIP, *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988.
232. H. Simonis & T. Le Provost, Circuit verification in chip: Benchmark results. *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, 125–129, November 1989.
233. R. Skuppin & T. Buckle, CLP and Spacecraft Attitude Control, *Proc. JICSLP Workshop on Constraint Logic Programming*, 45–54, 1992.
234. G. Smolka, Residuation and Guarded Rules for Constraint Logic Programming, in: *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (Eds.), MIT Press, 405–419, 1993.
235. G. Smolka & R. Treinen, Records for Logic Programming, Journal of Logic Programming, to appear. Also in: *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 240–254, 1992.
236. D. Srivastava, Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints, *Annals of Mathematics and Artificial Intelligence 8*, 315–343, 1993.
237. D. Srivastava & R. Ramakrishnan, Pushing Constraint Selections, *Journal of*

Logic Programming, 16, 361–414, 1993.

238. R.M. Stallman & G.J. Sussman, Forward Reasoning and Dependency Directed Backtracking in a System for Computer Aided Circuit Analysis, *Artificial Intelligence* 9, 135–196, 1977.
239. T. Sthanusubramonian, A Transformational Approach to Configuration Design, Master's thesis, Engineering Design Research Center, Carnegie Mellon University, 1991.
240. G. Steele & G.J. Sussman, CONSTRAINTS - A Language for Expressing Almost Hierarchical Descriptions, *Artificial Intelligence* 14(1), 1–39, 1980.
241. G.L. Steele, The Implementation and Definition of a Computer Programming Language Based on Constraints, Ph.D. Dissertation (MIT-AI TR 595), Dept. of Electrical Engineering and Computer Science, M.I.T. 1980.
242. M. Stickel, Automated Deduction by Theory Resolution, *Journal of Automated Reasoning* 1, 333-355, 1984.
243. P.J. Stuckey, Incremental Linear Constraint Solving and Detection of Implicit Equalities, *ORSA Journal of Computing*, 3, 269–274, 1991.
244. P. Stuckey, Constructive Negation for Constraint Logic Programming, *Proc. Logic in Computer Science Conference,* 328–339, 1991.
245. D. Subramanian & C-S. Wang, Kinematic synthesis with configuration spaces, *Proc. Qualitative Reasoning 1993*, D. Weld (ed), 228-239, 1993.
246. I. Sutherland, A Man-Machine Graphical Communication System, PhD thesis, Massachusetts Institute of Technology, January 1963.
247. A. Tarski, *A Decision Method for Elementary Algebra and Geometry*, University of California Press, 1951.
248. A. Taylor, LIPS on a MIPS: Results from a Prolog Compiler for a RISC, *Proceedings 7th International Conference on Logic Programming*, 174–185, 1990.
249. S. Terasaki, D.J. Hawley, H. Sawada, K. Satoh, S. Menju, T. Kawagishi, N. Iwayama & A. Aiba, Parallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers, *Proc. International Conference on Fifth Generation Computer Systems 1992,* Volume I, 330–346, 1992.
250. E. Tick, The Deevolution of Concurrent Logic Programming Languages, draft manuscript, 1993.
251. J.C. Tobias, II, Knowledge Representation in the Harmony intelligent tutoring system, Master's thesis, Department of Computer Science, University of California at Los Angeles, 1988.
252. B.M. Tong & H.F. Leung, Concurrent Constraint Logic Programming on Massively Parallel SIMD Computers, *Proc. International Logic Programming Symposium*, 388–402, 1993.
253. E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.
254. A. Véron, K. Schuerman, M. Reeve & L.L. Li, Why and How in the ElipSys OR-parallel CLP system, *Proc. Conf. on Parallel Architectures and Languages Europe*, 291–303, 1993.
255. J.S. Vitter & Ph. Flajolet, Average-case Analysis of Algorithms and Data Structures, *Handbook of Theoretical Computer Science*, Vol. A, Elsevier Science Publishers, Amsterdam, 431–524, 1990.
256. P. Voda, The Constraint Language Trilogy: Semantics and Computations, Technical Report, Complete Logic Systems, 1988.
257. P. Voda, Types of Trilogy, *Proc. 5th International Conference on Logic Programming*, 580–589, 1988.
258. C. Walinsky, $CLP(\Sigma*)$: Constraint Logic Programming with Regular Sets, *Proc. 6th International Conference on Logic Programming,* 181–196, 1989.
259. M. Wallace, A Computable Semantics for General Logic Programs, *Journal of Logic Programming* 6, 269–297, 1989.

260. M. Wallace, Applying Constraints for Scheduling, in: *Constraint Programming*, B. Mayoh, E. Tyugu and J. Penjaam (Eds), NATO Advanced Science Institute Series, Springer-Verlag, 1994.
261. D.H.D. Warren, An Abstract PROLOG Instruction Set, Technical note 309, AI Center, SRI International, Menlo Park (October 1983).
262. D.H.D. Warren, The Andorra Principle, presented at the Gigalips Workshop, 1987.
263. M. Wilson & A. Borning, Hierarchical Constraint Logic Programming, *Journal of Logic Programming*, 16, 277–318, 1993.
264. Roland Yap, Restriction Site Mapping in CLP($\mathcal{R}$), *Proceedings* 8^{th} *International Conference on Logic Programming*, MIT Press, June 1991, 521–534.
265. Roland Yap, A Constraint Logic Programming Framework for Constructing DNA Restriction Maps, *Artificial Intelligence in Medicine*, 5, 447–464, 1993.
266. Roland Yap, Contributions to CLP($\mathcal{R}$), Ph.D. thesis, Department of Computer Science, Monash University, January 1994 (expected).