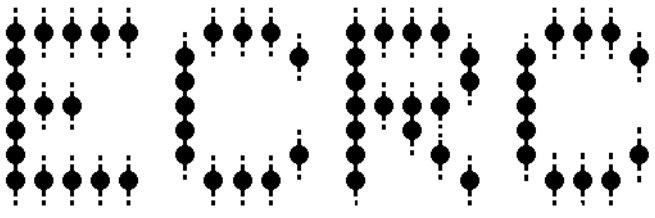
technical report ECRC–92–1

Generalised Constraint Propagation Over the CLP Scheme

Thierry Le Provost
Mark Wallace



EUROPEAN COMPUTER-INDUSTRY RESEARCH CENTRE

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

About this paper:
This paper appears in the Journal of Logic Programming, 1993

Abstract

Constraint logic programming is often described as logic programming with unification replaced by constraint solving over a computation domain. There is another, very different, CLP paradigm based on constraint satisfaction, where program-defined goals can be treated as constraints and handled using propagation. This paper proposes a generalisation of propagation, which enables it to be applied on arbitrary computation domains, revealing that the two paradigms of CLP are orthogonal, and can be freely combined. The main idea behind generalised propagation is to use whatever constraints are available over the computation domain to express restrictions on problem variables. Generalised propagation on a goal G requires that the system extracts a constraint approximating all the answers to G. The paper introduces a generic algorithm for generalised propagation called *topological branch and bound* which avoids enumerating all the answers to G. Generalised propagation over the Herbrand universe has been implemented in a system called *Propia*, and we describe its behaviour on some applications.

1 Introduction

1.1 The CLP Scheme

Constraint logic programming is often described as logic programming with unification replaced by constraint solving over a computation domain. This is captured in a theoretical framework called the CLP scheme [JL87]. A $CLP(X)$ program comprises rules of the form

$$h \leftarrow c_1, \ldots c_n, b_1, \ldots b_m$$

where the c_i are constraints over the domain X and the b_j are (user-defined or built-in) logic programming goals. During computation when goals are unfolded using program clauses, the constraints in their bodies are collected up and tested for consistency. In this paper we shall often refer to constraints in the $CLP(X)$ framework as *basic constraints*. One point to note is that the basic constraint predicates are built-into the system, and cannot be defined by program clauses. A second point is that the consistency check covers all the basic constraints which have been collected up during the computation (which distinguishes constraints from ordinary built-in predicates [Mah87]). This check must, in theory, be effective.

1.2 CSP in Logic Programming

There is another, very different, CLP paradigm which is based on constraint satisfaction techniques dating back to 1965 [GB65, Fik70, Mon74]. In the constraint satisfaction problem (CSP) paradigm the constraints are problem-specific, and defined by sets of tuples. When CSP is embedded into logic programming, a constraint can be defined in the program as a set of facts, or even as a set of rules [Van89]. We shall often refer to constraints in the CSP framework as "propagation constraints".

For solving CSP problems in traditional logic programming systems, backtrack search is used. The aim is to perform relevant "tests" as soon as possible after instantiating a variable. Dynamic computation rules, such as *freeze* [Col85] and *delay* [Nai86, MAC+89] can be used to determine which goal to evaluate next. However even such dynamic rules can only postpone evaluation until the propagation constraints are partially or fully instantiated. Evaluating partially instantiated propagation constraints will generate values for variables, usually creating undesirable branches in the search tree. Waiting till the constraint is ground before evaluating, is to use it as an a posteriori test. To summarise, logic programs can only use propagation constraints *passively*. Our motivation for constraints logic programming is to support the *active* use of constraints [Gal85]. This is provided by techniques developed for solving constraint satisfaction problems.

It should be noted that constraint solving over a computation domain, as described in section 1.1 above, is replaced in this paradigm by constraint propagation over value domains [Fik70, Mon74, Mac77, HE80]. Informally constraint propagation operates by looking ahead at yet unsolved goals to see what locally consistent valuations there remain for individual problem variables. In the CSP framework there is no guarantee that, after a complete propagation sequence, the propagation constraints are globally consistent, by contrast with constraint solving for basic constraints in the CLP scheme. However such propagation techniques can have a dramatic effect in cutting down the size of the search space. Evidence of the practical effectiveness of constraints propagation in logic programming is given in [DSV90].

1.3 Restrictions on Propagation in Logic Programming

One prerequisite for applying CSP techniques is that problem variables should have an associated domain of possible values. Traditionally [Mac77, HE80] this is a finite domain, though more recently continuous intervals have been studied [Dav87]. Up to now, constraint logic programming systems based on the CSP paradigm (eg CHIP [DVS+88]) have only been defined for finite domain variables. For each problem variable a finite domain declaration is required. Each such variable can only take a finite number of values, and looking ahead is a way of deterministically ruling out certain locally inconsistent values and thus reducing the domains.

This restriction has prevented the application of propagation to new computation domains introduced by the CLP scheme and related approaches. In addition propagation as currently defined only exploits a fraction of the power of its native universe of discourse. For instance it cannot reason on compound terms, thereby enforcing an unnatural and potentially inefficient encoding of structured data as collections of constants.

This has meant that the two approaches to integrating constraints into logic programming, as *basic* constraints and as *propagation* constraints, have had to remain quite separate. Even in the CHIP system [DVS+88] which utilises both types of integration, propagation is excluded from those parts of the programs involving new computation domains, such as Boolean algebra or linear rational arithmetic.

In this paper we alleviate two restrictions. Firstly we lift the restriction to *finite* domains, for propagation in logic programming. Secondly we lift the restriction that *domains* are needed for propagation at all. In this second sense, generalised propagation makes a contribution not just to the field of CLP, but also to the general field of constraint satisfaction.

1.4 Generalised Propagation

This paper proposes a generalisation of propagation, which enables it to be applied on arbitrary computation domains. Generalised propagation can be applied in $CLP(X)$ programs, whatever the domain X. We shall call $GP(X)$ the system applying generalised propagation in $CLP(X)$. Finite domain propagation in logic programming is just $GP(FD)$.

The basic concepts, theoretical foundations, and abstract operational semantics of $GP(X)$ can be defined independently of the computation domain, X. This allows programmers to reason about the efficiency of $GP(X)$ in an intuitive and uniform way. This generality carries over to the implementation, where algorithms for executing generalised propagation apply across a large range of basic constraint theories. Last but not least, the declarative semantics of $CLP(X)$ programs is preserved in $GP(X)$.

The main idea behind $GP(X)$ is to use whatever constraints are available over the computation domain X to express restrictions on problem variables. (Associating finite domains with variables is one specific application of this concept.) Goals designated as propagation constraints are repeatedly approximated as closely as possible using these constraints. When no further refinement of the current resolvent's approximation is feasible, a resolution step is performed and propagation starts again.

Consider a toy example. The query to be answered is $\leftarrow and(X,Y,Z), eqv(X,Y)$ against the program
$and(true, true, true) \leftarrow$
$and(true, false, false) \leftarrow$
$and(false, true, false) \leftarrow$
$and(false, false, false) \leftarrow$

$eqv(true, true) \leftarrow$
$eqv(false, false) \leftarrow$

To use finite domain propagation we declare that X, Y and Z can each take two possible values $\{true, false\}$. This is their finite domain. Now propagation on each atomic goal in the query could be used to attempt to reduce the possible values for each variable (by eliminating impossible ones). Propagation on $and(X,Y,Z)$ yields no domain reductions, however, because, for each variable, every domain value appears in some clause for and. Nor does propagation on $eqv(X,Y)$ produce any domain reductions.

However generalised propagation can be more successfully applied. We shall assume, for this example, that the domain of computation is just the usual one of Prolog (so there is no built-in boolean constraint solver). The constraints built-into the system are just equations, treated as usual by unification. Generalised propagation over this domain of computation (which we call $GP(HU)$) can extract from propagation only equations between terms.

Propagation on $and(X,Y,Z)$ does not, initially, produce any equations, but propagation on $eqv(X,Y)$ does produce the equation $X = Y$. This information is extracted since it holds for both answers $X = true \wedge Y = true$ and $X = false \wedge Y = false$ to the subquery $\leftarrow eqv(X,Y)$. Now propagation is retried on

$and(X, X, Z)$ (which is $and(X, Y, Z)$ in the environment $X = Y$). This time an equation can be extracted $X = Z$, which holds of both answers to the subquery. Thus propagation on both atoms in the original query yields the equations $X = Y \wedge X = Z$ which are guaranteed to hold for all answers to the query. Notice that the information thus extracted could not be expressed using variable domains.

The practical relevance of generalised propagation has been tested by implementing it in the underlying constraint theory of first-order terms with syntactic equality [Cla79], which is $GP(HU)$. Programs are just sets of Prolog rules with annotations identifying the goals to be used for propagation. The language has enabled us to write programs which are simple, yet efficient, without the need to resort to constructs without a clear declarative semantics such as demons. Applications tackled include a set of propositional satisfiability problems collected as a benchmark for theorem provers [MR91], temporal reasoning, and disjunctive scheduling problems. The performance results have been very encouraging.

In the next section we shall describe finite domain propagation in logic programming, and introduce generalised propagation with an example. Then in section 3 we shall specify generalised propagation, discussing its logical and operational semantics and introducing a generic algorithm for its implementation over arbitrary computation domains. In section 4 we shall describe an implementation of generalised propagation called *Propia*. In section 5 we shall compare generalised propagation with some related approaches, and we shall conclude in section 6.

2 Constraint Propagation

In this section we briefly review the motivation of finite domain propagation in logic programming and describe its behaviour with some examples. Then we shall introduce generalised propagation in logic programming.

2.1 Propagation in Constraint Logic Programming

The idea behind local propagation methods for CSP's is to work on each propagation constraint independently, and deterministically to extract information about locally consistent assignments. This has lead to various consistency algorithms for networks of constraints, the most widely applicable of these being arc-consistency [RHZ75, Mon74]. Consistency can be applied as a preliminary to the search steps or interleaved with them [HE80]. The application of these techniques in logic programming can be related back to the enforcement of link consistency in connections graphs [Kow79]. Finite domain propagation in logic programming was accomplished through two complementary extensions [VD86, Van89]

- explicit *finite domains* of values to allow the expression of range restrictions, together with the corresponding extension of unification (FD-resolution)
- new *inference rules*, based on looking ahead at "future" computations, to reduce finite domains in a deterministic way

The effect of looking ahead on a goal is to reduce the domains associated with the variables in the goal, so that the resulting domains approximate *as closely as possible* the set of remaining solutions. Application of these inference rules is repeated on all propagation constraint goals until no more domain reductions are possible, forming a *propagation sequence*. Propagation constraint goals that are satisfied by any combination of values in the domains of their arguments can now be dropped.

One algorithm for implementing lookahead is to enumerate all combinations of values for the constraint's arguments and check each combination by calling the goal instantiated with these values. The reduced domains are then formed by projecting successful combinations onto each argument. CHIP in addition implements a variety of predefined constraint predicates, which efficiently perform domain reduction by specialised algorithms. (The drawback of such dedicated algorithms is that they cannot be applied to program-defined predicates.) An example problem encoded in a CHIP-like syntax follows:

$csp(X1, X2, X3, X4) \leftarrow$
$\{X1, X2, X3, X4\} :: \{a, b, c\},$
constraint $p(X3, X1),$ [1]
constraint $p(X2, X3),$ [2]
constraint $p(X2, X4),$ [3]
constraint $p(X3, X4)$ [4]

$p(a, b) \leftarrow$
$p(a, c) \leftarrow$
$p(b, c) \leftarrow$

The *constraint* annotations identify goals that must be treated by the new inference rule. Annotations can be ignored for a declarative reading.

For our example problem, the initial propagation sequence is sufficient to produce the only solution. A possible computation sequence is as follows (though the ordering is immaterial for the final result):

Goal	Result of propagation
constraint $p(X3, X1)$	[1] produces $X3 :: \{a, b\}, X1 :: \{b, c\}$
constraint $p(X2, X3 :: \{a, b\})$	[2] produces $X2 = a, X3 = b$
constraint $p(a, X4)$	[3] produces $X4 :: \{b, c\}$
constraint $p(b, X4 :: \{b, c\})$	[4] produces $X4 = c$
constraint $p(b, X1 :: \{b, c\})$	[1] produces $X1 = c$
constraint $p(a, b)$	[2] succeeds
constraint $p(a, c)$	[3] succeeds
constraint $p(b, c)$	[4] succeeds

Note that the propagation constraint [1] takes part in two propagation steps before it is solved. In general constraints may be involved in any number (> 0) of propagation steps.

This example is deliberately very simple. Normally an answer is not obtained by propagation alone. For example if we add the fact $p(c, a) \leftarrow$ to the program definition of p above, then propagation produces no domain reductions at first. If a propagation sequence terminates without producing an answer, then variables are instantiated non-deterministically to values in their domains: this can be achieved by an explicit "labelling" routine (as in CHIP) or by an implicit labelling performed automatically by the system.

2.2 A Motivating Example of Generalised Propagation

We briefly motivate generalised propagation with a simple example. The problem we consider is that of compiling crosswords from an empty grid and a lexicon of available words.

The problem can be expressed as a logic program by recording the lexicon as a set of facts, and the grid as a rule, thus:
$w3(a, c, t) \leftarrow$
$w3(a, r, t) \leftarrow$
...
$w4(a, b, l, e) \leftarrow$
$w4(b, a, k, e) \leftarrow$
...

$grid([A1, A2, \ldots, Zn]) \leftarrow$
$w5(A1, A2, A3, A4, A5),$
$w4(A3, B3, C3, D3),$
$w3(A5, B5, C5),$
...

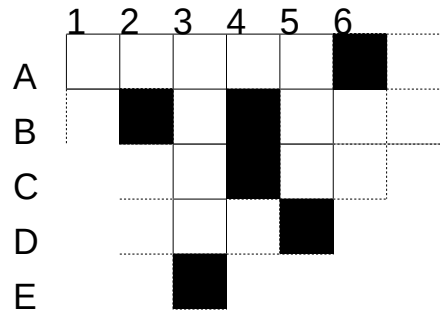


Figure 2.1: Part of a crossword grid, showing three blank words

With this encoding, the search space for the query $\leftarrow grid([A1, \ldots, Zn])$ for any non-trivial crossword is unfortunately too large for any hope of obtaining a solution without further guidance.

The problem suggests itself for finite domain propagation, where a domain of $\{a, b, \ldots, z\}$ can be associated with each variable, and each word in the grid is used as a propagation constraint. Unfortunately finite domain propagation still does not enable the program to yield a solution within any reasonable elapsed time. One reason for this inefficiency is that too much time is spent removing letters from the domains of the different variables without a compensating improvement in the search behaviour. For example the removal of rare letters such as x from the domains of the variables provides little useful "pruning" of the search space.

Nevertheless finite domain propagation in logic programming has been applied to the problem of crossword compilation [Van89]. A successful CLP program was written in which a domain variable was associated with each blank word in the crossword, whose domain was the set of words in the lexicon that could fit there. The correct choice of words was enforced by constraints on their intersections. The finite domains associated with the variables had around 30 possible words. In fact the total lexicon only had around 150 words. The two drawbacks of this solution were that the representation of the problem was unnatural, and it only worked for small lexicons.

The problem was tackled using $GP(HU)$. Syntactically the only change necessary to the above program was to annotate each blank word as a propagation constraint, thus:

$grid([A1, A2, \ldots, Zn]) \leftarrow$
    constraint $w5(A1, A2, A3, A4, A5)$,
    constraint $w4(A3, B3, C3, D3)$,
    constraint $w3(A5, B5, C5)$,
    $\ldots$

Instead of propagating domain reductions, the system propagates equalities. Thus the information that is produced is positive information, which helps the search converge on a solution, rather than negative information which prunes impossible, but often irrelevant, branches. Most importantly, if the lexicon grows the quality of information produced remains good. For larger domains, by contrast, the value of negative information is reduced. (We are reminded of the paradox which says that a black raven is - logically - evidence that all cows are purple, since it provides negative evidence of the fact that all non-purple objects are not cows!)

Not only does the $GP(HU)$ program allow the original problem, with a lexicon of 150 words, to be

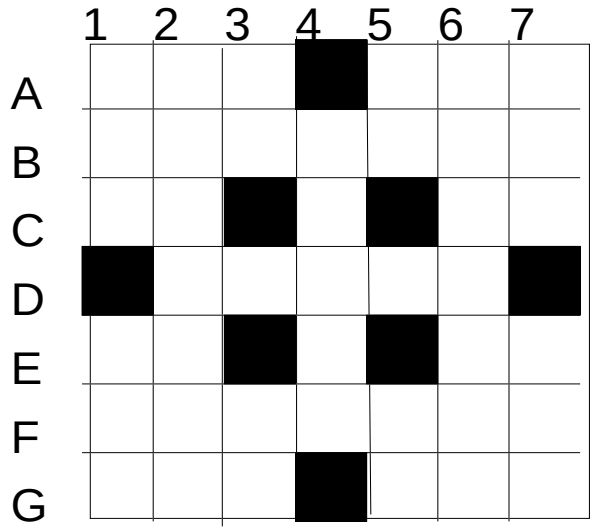


Figure 2.2: A blank crossword grid, with four difficult corners

solved, it enables the problem to be scaled up to realistic proportions. Using generalised propagation, this program compiles crosswords from a lexicon of 25000 words. A more detailed discussion of this application follows in section 4.1.1 below.

2.3 Granularity of Propagation Constraints

A nice property of constraint logic programming is the fine level of control it offers over problem solving. A propagation constraint goal can be defined by rules and therefore can be arbitrarily complex. As an example consider the following small crossword grid: This can be encoded as a single grid, using propagation, as above, on each blank word:

$grid2([A1, A2, \ldots, G7]) \leftarrow$
constraint $w3(A1, A2, A3)$,
constraint $w7(B1, B2, B3, B4, B5, B6, B7)$,
constraint $w2(C1, C2)$,
constraint $w3(A1, B1, C1)$,
constraint $w7(A2, B2, C2, D2, E2, F2, G2)$,
constraint $w2(A3, B3)$,
constraint $w3(A5, A6, A7)$,
...

Such an encoding performs propagation at a very fine level of granularity, enforcing very local consistency. However a more coarse granularity of propagation suggests itself for such a problem: we should check, for each corner of the crossword and the centre, what constraints it imposes on the words which cross the boundaries. The problem can be expressed naturally as five subproblems, with propagation performed on each:

$grid2([A1, A2, \ldots, G7]) \leftarrow$
constraint $topleft([A1, A2, A3, B1, \ldots, B7, C1, C2, D2, \ldots, G2])$,

constraint $botleft([G1, G2, G3, F1, \ldots, F7, E1, E2, D2, \ldots, A2])$,
constraint $centre([B4, C4, D3, \ldots, D6, E4, F4])$,
...

Now the system will iterate over solutions to the subproblems and try to extract equations common to all these solutions. In case the subproblems themselves are hard, it is of course possible to perform propagation within the search for their subsolutions:

$topleft([A1, \ldots, G2]) \leftarrow$
constraint $w3(A1, A2, A3)$,
constraint $w7(B1, \ldots, B7)$,
...

Clearly it brings nothing to define the *whole* problem as a single propagation constraint. However the facility to combine clusters of constraints into a single larger constraint means that propagation can be used to enforce consistency just as local or global as necessary for the problem at hand. The only practical necessity is to treat efficiently constraints involving a number of variables. Generalised propagation provides a framework where local and global propagation are practical alternatives.

3 A Specification of $GP(X)$

3.1 Definitions

The language syntax and semantics used in this paper are based on first order logic. Atomic formulae are built from variables, predicate symbols, function symbols and constant symbols. If Φ is any open formula, then $\exists\Phi$ and $\forall\Phi$ denote respectively the existential and universal closure of Φ as usual. We also introduce the following syntax: if ψ is an unquantified formula, then $\exists_{\setminus\psi}\Phi$ is the existential quantification over all those variables in Φ which do not occur in ψ. For example $\exists_{\setminus p(X,Z)}.X \geq Y \wedge Y \geq Z$ is the formula $\exists Y.(X \geq Y \wedge Y \geq Z)$. This syntax is useful for expressing answers to queries. For example if $\leftarrow p(X, Z)$ were a query, then the above example could denote an answer.

The predicate symbols are divided into *interpreted* predicates and *uninterpreted* predicates. The function and constant symbols are divided similarly.

For a given computation domain X, the interpreted symbols have a predefined interpretation, independent of the programs in which they appear. The $=$ predicate symbol is always an interpreted predicate, interpreted as equality in the underlying domain. Two further predicates which are always interpreted are $true$ and $false$. Some examples of interpreted function symbols are $+$ and $-$ over numerical domains such as the integers.

By contrast the semantics of the uninterpreted predicates is dictated by the program. Uninterpreted functions and constants have the free interpretation in the underlying domain.

An atom containing only interpreted predicates, functions and constants is termed an *interpreted constraint*. An atom containing only uninterpreted predicates, functions and constants is termed a *user* atom. An atom cannot contain both interpreted and uninterpreted symbols.[1]

We admit an additional syntax for atoms *constraint* A where A is a user atom. This syntax yields another kind of constraints called *propagation* constraints. Unlike interpreted constraints, propagation constraints have uninterpreted predicates whose semantics are dictated by the program.

We now further distinguish two classes of interpreted constraints. These are the *basic* constraints and *approximation* constraints. The conjunction of a set of basic or approximation constraints is also termed a basic or approximation constraint respectively. Constraints of the form $V = C$ where V is a variable

[1] In practice one can admit such atoms (e.g. $p(1 + X)$) and view them as abbreviations for a conjunction where the equalities are made explicit (e.g. $p(Y) \wedge Y = (1 + X)$).

and C is a constant are always classed as basic constraints. Similarly $true$ and $false$ are basic constraints. Their interpretation in any computation domain is obvious.

Approximation constraints "approximate" basic constraints in the sense that for any approximation constraint AC there are basic constraints C such that $X \models (C \rightarrow AC)$. An example of an approximation constraint is $X :: \{1, 2, 3\}$ which states that either $X = 1$ or $X = 2$ or $X = 3$. It approximates each of the basic equality constraints $X = 1$, $X = 2$ and $X = 3$. Approximation constraints are a generalisation of Davis' *labels* [Dav87]. The approximation constraints and the basic constraints need not be disjoint: in other words basic constraints could approximate themselves.

A $GP(X)$ program is a set of clauses of the form $Head \leftarrow Goal_1, \ldots, Goal_s$ The head $Head$ is a user atom. The body $Goal_1, \ldots, Goal_s$ is a set of atoms, which may include user atoms and constraints. A clause with an empty head $\leftarrow Goal_1, \ldots, Goal_s$ is termed a *query*. The set of clauses whose heads have the same predicate p are termed the *program definition* of p.

An example clause is

$$
\begin{aligned}
profit(Company, P) \leftarrow & \\
& \text{constraint } public(Company), \\
& income(Company, I), \\
& expenditure(Company, E), \\
& P = I - E
\end{aligned}
$$

It has four atomic goals in its body, of which "constraint $public(Company)$" is a propagation constraint, "$income(Company, I)$" and "$expenditure(Company, E)$" are user atoms, and "$P = I - E$" is a basic constraint.

3.2 Declarative and Operational Semantics for GP(X)

3.2.1 A Framework for Evaluation in $GP(X)$

The framework for evaluation in $GP(X)$ is based on the constraint logic programming scheme of Jaffar and Lassez [JL87, JL86], extended with the concept of propagation agents. An evaluation in $GP(X)$ involves at any time a current goal, a current set of propagation agents, and a current constraint store. Thus the state of an evaluation is represented by a triple $< Goal, Agents, Store >$.

3.2.2 Declarative Semantics for GP(X)

We base our semantics on that introduced for the CLP scheme in [JL86]. The computation domain X provides an interpretation for the interpreted predicates, functions, and constants. The language L_P of a $CLP(X)$ program includes uninterpreted predicates, functions and constants. An interpretation I of L_P is *based on* X if I has the same underlying domain as X, and the same interpretation for the interpreted predicates, functions and constants.

We say an L_P formula F_1 *X-entails* a formula F_2, written $F_1 \models_X F_2$, to mean that for every interpretation I based on X, if $I \models F_1$ then $I \models F_2$. If an L_P formula F is true in every interpretation based on X we write $\models_X F$. If a valuation θ of the variables makes a formula F true in X, we write $X \models F\theta$.

Logically a propagation constraint *constraint* A is equivalent to the user atom A. In fact we shall define for any $GP(X)$ program P and query Q a $CLP(X)$ program $clp(P)$ and query $clp(Q)$ which result by replacing all propagation constraints "*constraint* A" by the user atom A. The declarative semantics for P and Q is, by definition, the declarative semantics for $clp(P)$ and $clp(Q)$. Thus the declarative semantics for $GP(X)$ programs and queries reduce to the semantics for $CLP(X)$ programs and queries.

A clause $Head \leftarrow Goal_1, \ldots, Goal_s$ with free variables $X_1, \ldots X_t$ as usual denotes the formula $\forall X_1, \ldots X_t.(Head \vee \neg Goal_1 \vee \ldots \vee \neg Goal_s)$.
The meaning of a program is given by the conjunction of its clauses. The denotation of a query $\leftarrow Goal_1, \ldots, Goal_s$, is the formula $\neg Goal_1 \vee \ldots \vee \neg Goal_s$.

A solution to a $GP(X)$ query $\leftarrow G$ against a program P is a variable valuation θ for which P X-entails $G\theta$. For the purposes of the formalisation of soundness and completeness we use a more general definition of an *solution under a constraint*

Definition 1 *For a given $GP(X)$ program P, a solution to a query $\leftarrow G$ under a constraint S is an valuation θ for which $X \models S\theta$ and $P \models_X G\theta$*

3.2.3 Operational Semantics for GP(X)

We have chosen a transformational semantics for our constraint logic programming system following the approach of [Sar89] and [HD91].

GP(X) States At any point in a $GP(X)$ evaluation, the current state is formalised as a triple $< \{G_1, \ldots, G_r\}, \{A_1, \ldots, A_s\}, \{C_1, \ldots, C_t\} >$. The current goal $\{G_1, \ldots, G_r\}$ is a set of atoms, which may include user atoms and atomic constraints. The current set of propagation agents $\{A_1, \ldots, A_s\}$ is a set of user atoms. Finally the constraint store $\{C_1, \ldots, C_t\}$ is a set of interpreted constraints, which may include both basic constraints and approximation constraints.

A state $< G, A, S >$ has a logical denotation, which we will often use in reasoning about soundness and completeness of the operational semantics. The logical denotation of an atomic goal and an atomic constraint has been discussed in the previous section. The propagation agent A_i has the same denotation as the user atom A_i. The denotation of a set of atomic goals, or agents, or constraints is their logical conjunction. In the following we sometimes use the symbols G, A and S to refer to sets, and sometimes to logical conjunctions, depending on the context.

A derivation via a $GP(X)$ program P can be formalised as sequence of state transitions, $< G1, A1, S1 > \mapsto < G2, A2, S2 > \mapsto \ldots \mapsto < Gn, An, Sn >$., where the possible transitions depend on P. To avoid ambiguity we usually make the program explicit by referring to "P-derivations", and later "P-refutations" and "P-computed answers". A P-derivation starting with the state $< G1, A1, S1 >$ and ending in the state $< Gn, An, Sn >$ is written $< G1, A1, S1 > \Longrightarrow < Gn, An, Sn >$.

A query $\leftarrow G$ is evaluated against a $GP(X)$ program by initialising the goal to G,[2] the empty constraint store and an empty set of propagation agents. Thus the initial state is $< G, \emptyset, \emptyset >$. In general the goal G may contain both propagation constraints (such as "*constraint public*(*company*)" in the example in section 3.1 above), basic constraints (such as "$P = I - E$") and user atoms (such as "$income(Company, I)$").

There are two kinds of terminal state, *success* states, and $failure$ states. A $failure$ state is a state in which the constraint store contains the atom $false$. As described below, this atom is added whenever the constraints in the constraint store S become unsatisfiable (i.e. $X \models \neg\exists S$). A *success* state is one that has an empty goal and an empty set of propagation agents and whose constraint store is consistent. Thus $< \emptyset, \emptyset, S >$ is a *success* state whenever S does not contain $false$. A P-refutation is a P-derivation of the form $< G, \emptyset, S1 > \Longrightarrow < \emptyset, \emptyset, S2 >$, whose final state is a success state. (In practice propagation agents may be present, as long as no propagation on the agents is performed.)

State Transitions From certain states several alternative transitions are possible. Thus a $GP(X)$ evaluation involves the search of a tree whose branches correspond to alternative $GP(X)$ derivations. However in this section we concern ourselves purely with the definition of individual state transitions.

We shall use as an example the integers as a computation domain with basic constraints $T_1 = T_2$, and approximation constraints $T_3 \geq T_4$ and $T_3 \leq T_4$. The terms T_3 and T_4 are restricted to constants or variables. Any pair of approximation constraints $T_a \leq T_b \wedge T_b \leq T_c$ will be abbreviated to $T_a \leq T_b \leq T_c$. Our example program will comprise two predicate definitions:

[2] Strictly the goal is the set of atoms in the body of $\leftarrow G$

$p1(3,0) \leftarrow$
$p1(1,1) \leftarrow$
$p1(2,3) \leftarrow$

$p2(3,2) \leftarrow$
$p2(1,1) \leftarrow$
$p2(3,4) \leftarrow$

State Transitions Inherited from CLP(X) As in $CLP(X)$, a user atom $p(t1,\ldots,tm)$ is processed by selecting a clause $p(u1,\ldots,um) \leftarrow B_1,\ldots B_n$ from the program definition of p, (The variables in the clause are renamed so that they are different from the variables occurring in the current state.) The atom $p(t1,\ldots,tm)$ is then replaced in the goal by the set $\{u1 = t1,\ldots,um = tm, B_1 \ldots B_n\}$. If the program definition of p is empty, then the atom $p(t1,\ldots,tm)$ is replaced in the goal by $false$. Otherwise, each clause in the definition of p defines an alternative transition. The transition can be expressed in the following form (based on [Sar89]):

$$\frac{(p(u1,\ldots,um) \leftarrow B1,\ldots,Bn) \in P}{< (G \cup \{p(t1,\ldots,tm)\}), A, S > \mapsto < (G \cup \{u1 = t1,\ldots,um = tm, B1,\ldots Bn\}), A, S >}$$

and

$$\frac{\neg\exists X1,\ldots,Xn, B.(p(X1,\ldots,Xm) \leftarrow B) \in P}{< (G \cup \{p(t1,\ldots,tm)\}), A, S > \mapsto < (G \cup \{false\}), A, S >}$$

Against our example program, a possible transition is
$< \{p1(X,Y), p2(X,Y)\}, \emptyset, \emptyset > \mapsto < \{X = 3, Y = 0, p2(X,Y)\}, \emptyset, \emptyset >$.

As in $CLP(X)$, when a basic constraint is selected it is removed from the current goal and added to the constraint store using a variant of the *tell* operation. The *tell* adds constraints to the constraint store if they are consistent. The operation $tell(C,S)$ checks the new interpreted constraint C for consistency with the current store S ($X \models \exists.(S \wedge C)$), and if consistency is established the constraint store becomes $S \cup C$. If consistency is not established ($X \models \neg\exists.(S \wedge C)$) then the basic constraint $false$ is added to the constraint store. The resulting state is therefore a $failure$ state.

$$tell(C,S) = \begin{cases} C \cup S & \text{if } X \models \exists.(S \wedge C) \\ \{false\} \cup S & \text{otherwise} \end{cases}$$

The transition is expressed as follows:

$$\frac{}{< (G \cup \{C\}), A, S > \mapsto < G, A, tell(C,S) >}$$

A simple example is the transition
$< \{X = 3, Y = 0, p2(X,Y)\}, \emptyset, \emptyset > \mapsto < \{Y = 0, p2(X,Y)\}, \emptyset, \{X = 3\} >$.

To minimise the number of choice points in the evaluation tree in practical systems the previous two transactions are combined with the test of the constraints in the body, yielding the single transition:

$$\frac{\begin{array}{l}(p(u1,\ldots,um) \leftarrow Body) \in P \\ Body = \{c1,\ldots,ck\} \cup \{B1,\ldots,Bn\} \\ X \models \exists.(S \wedge u1 = t1 \wedge \ldots \wedge um = tm \wedge c1 \wedge \ldots \wedge ck)\end{array}}{< (G \cup \{p(t1,\ldots,tm)\}), A, S > \mapsto < (G \cup \{B1,\ldots Bn\}, A, (S \cup \{u1 = t1,\ldots,um = tm, c1,\ldots ck\}) >}$$

We will, however, use the individual transactions in the completeness proof in section 3.3.2 below.

New GP(X) State Transitions The difference from $CLP(X)$ lies in the handling of propagation constraints. When a propagation constraint *constraint* A_i is selected, the atom A_i is added to the set of propagation agents. The transition is as follows:

$$\frac{}{< (G \cup \{\text{constraint } A_i\}), A, S > \mapsto < G, (A \cup \{A_i\}), S >}$$

An example is:
$< \{\text{constraint } p1(X,Y), p2(X,Y)\}, \emptyset, \emptyset > \mapsto < \{p2(X,Y)\}, \{p1(X,Y)\}, \emptyset >$.

The propagation agents spontaneously and repeatedly cause further state transitions in which new approximation constraints are added, if consistent, to the constraint store. In section 3.4.2 below, we shall formalise an operator $prop(A_i, S_{old})$ that extracts from a constraint store S_{old} and a propagation agent A_i an approximation constraint. The extracted constraint is satisfied by all solutions to the propagation agent with the input constraint store in the following sense. For a $GP(X)$ program P, if θ is any solution to $\leftarrow A_i$ under store S_{old}, then $X \models prop(A_i, S_{old})\theta$.

It is the spontaneous production of new information, in the form of approximation constraints, that we call generalised propagation. Generalised propagation can be seen as an example of the *relaxed tell* operation of [HD91] which is discussed in more detail in section 5.2, below.

For any state $< G, A, S >$ in which A_i is a propagation agent ($A_i \in A$), there is a possible state transition corresponding to single propagation steps on an agent A_i in each subset S_{old} of the constraint store S. However if $prop(A_i, S_{old})$ is already implied by S then no transition takes place (since the resulting state would admit all the same transitions as the original state). Otherwise, the transition *tell*'s AC_i to the constraint store S. The transition is as follows:

$$\frac{\begin{array}{l} S_{old} \subseteq S \\ A_i \in A \\ X \models \neg\forall.(S \rightarrow prop(A_i, S_{old})) \end{array}}{< G, A, S > \mapsto < G, A, tell(prop(A_i, S_{old}), S) >}$$

In our example program $prop(p1(X,Y), \emptyset) = (1 \leq X \leq 3 \wedge 0 \leq Y \leq 3) = AC1$.
Thus there is a transition:
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, \emptyset > \mapsto < \emptyset, \{p1(X,Y), p2(X,Y)\}, AC1 >$.

In a sequential implementation, the constraint store S_{old} used for propagation is the current constraint store S (i.e. $S = S_{old}$). Conversely, suppose the calculation of $prop(A_i, S_{old})$ takes place in parallel with some state transitions. In this case, at the time $prop(A_i, S_{old})$ is *told* back to the constraint store S, the store may include new constraints (i.e. no longer is $S = S_{old}$). Hence the condition $S_{old} \subseteq S$. We shall give an example of this in section 3.5.2 below.

The final transition returns a propagation agent from the set of agents to the current goal. This transition enables the propagation constraints eventually to be unfolded like ordinary user atoms. The unfolding is necessary to ensure the soundness of $GP(X)$ computed answers.

The transition is as follows:

$$\frac{}{< G, (A \cup \{A_i\}), S > \mapsto < (G \cup \{A_i\}), A, S >}$$

An example is the transition
$< \{p2(X,Y\}, \{p1(X,Y)\}, \emptyset > \mapsto < \{p1(X,Y), p2(X,Y)\}, \emptyset, \emptyset >$.

GP(X) Computed Answers We now define the computed answer returned by a P-refutation.

Definition 2 *For a program P, a P-computed answer, to a subquery $\leftarrow G$ with constraint store S_0 is $\exists_{\backslash G} S$ where S is the final constraint store in any P-refutation $< G, \emptyset, S_0 > \Longrightarrow < \emptyset, \emptyset, S >$.*

Formally, no propagation agents can appear in either the initial or the final state. However, as noted in section 3.2.3above, propagation agents may be present, as long as no propagation on the agents is performed.

3.3 Soundness and Completeness

3.3.1 Soundness of GP(X)

Firstly note that constraints are only added to the constraint store using our *tell* operator. This ensures that if the constraint store in any state is not consistent it is $false$, the state is a $failure$ state, and, by definition, no further transitions are possible.

For soundness we require that all the computed answers represent correct solutions.

Definition 3 *For a $GP(X)$ program P, a computed answer Ans to a query $\leftarrow G$ with constraint store S_0 is sound if every valuation θ such that $X \models Ans\theta$ is a solution to $\leftarrow G$ under constraint $\exists_{\backslash G} S_0$.*

The following lemma follows immediately:

Lemma 1 *For a $GP(X)$ program P, a computed answer Ans to a query $\leftarrow G$ under constraint S_0 is sound if and only if $X \models \forall.(Ans \rightarrow \exists_{\backslash G} S_0)$ and $P \models_X \forall.(Ans \rightarrow G)$*

The result we shall prove is that for each P-derivation, $< G_1, A_1, S_1 > \Longrightarrow < G_2, A_2, S_2 >$ the final state logically implies the initial state.

Lemma 2 *For any P-derivation $< G_1, A_1, S_1 > \Longrightarrow < G_2, A_2, S_2 >$, it is the case that $P \models_X \forall.(G_2 \wedge A_2 \wedge S_2) \rightarrow (G_1 \wedge A_1 \wedge S_1)$.*

Proof

By examining each allowed transition in turn, it is clear that the result holds for P-derivations of length one. Inductively the result follows for derivations of any finite length.

A particular case of this result is when the derivations are in fact complete refutations. In case $\leftarrow G_1$ is a query, and $\exists_{\backslash G1} S_1$ is a computed answer, there is a P-refutation $< G_1, \emptyset, S_0 > \Longrightarrow < \emptyset, \emptyset, S_1 >$. Since $S_1 \supseteq S_0$, the first requirement for soundness $X \models \forall.(\exists_{\backslash G1} S_1 \rightarrow \exists_{\backslash G1} S_0)$ is satisfied. Since the P-refutation is sound by the above lemma, $P \models_X \forall.(S_1 \rightarrow G_1)$, which satisfies the second requirement for soundness. We have therefore established the following theorem.

Theorem 3 *For every $GP(X)$ program P, every P-computed answer to any subquery $\leftarrow G$ with any constraint store S_0 is sound.*

3.3.2 Completeness of GP(X)

In this section we shall not only prove that every correct solution is found by some refutation, but we shall also show that completeness is retained even if the system commits to certain transitions without exploring any alternatives. In particular the order of selection of goals is immaterial, and the order, "timing" and number of propagation steps makes no difference to the set of reachable *success* states.

Our approach is based on that of Jaffar and Lassez [JL86] where the computation domain is a predefined structure. Later papers, after [Mah87], specify the domain as a theory and thus obtain a stronger completeness result. However standard domains, such as the Herbrand domain, cannot be defined precisely enough for our needs by a theory, so we have returned to the earlier formalisation.

Our completeness requirement is expressed as follows (see [Smo91]):

Definition 4 *Over the computation domain X, a set of computed answers R represents a set of solutions Θ, if, for every solution $\theta \in \Theta$, there is a computed answer $r \in R$ such that $X \models r\theta$.*

Theorem 4 *For any $CLP(X)$ program P, the set of P-computed answers to any query $\leftarrow G$ under with any constraint store S_0 represents the set of solutions to $\leftarrow G$ under $\exists_{\backslash G} S_0$.*

Proof

For unconstrained queries, the proof is in [JL86], and sketched as part of the proof of theorem **1** in [Mah87]. The presence of constraints S_0 in the initial store only cuts off derivations

which yield a computed answer inconsistent with S_0 (since in the CLP transitions defined above only the *tell* operation is affected by the current constraint store). Solutions θ which satisfy such computed answers do not satisfy S_0, and therefore they are not solutions to $\leftarrow G$ under constraint $\exists_{\setminus G} S_0$. Consequently the remaining computed answers indeed represent all the solutions to $\leftarrow G$ under $\exists_{\setminus G} S_0$.

As we pointed out in section 3.2.2 above, since the logical denotation of *constraint* A_i is defined to be the denotation of A_i, the declarative semantics of the program P and goal G are precisely the declarative semantics of the $CLP(X)$ program $clp(P)$ and goal $clp(G)$ respectively. Consequently the answers to $\leftarrow clp(G)$ against the program $clp(P)$ are precisely the answers to $\leftarrow G$ against P.

We now use the completeness of $CLP(X)$ to prove that $GP(X)$ is also complete.

Theorem 5 *For any $GP(X)$ program P, the set of P-computed answers to any query $\leftarrow G$ with constraint store S_0 represents the set of solutions to $\leftarrow G$ under $\exists_{\setminus G} S_0$.*

Proof

For any $GP(X)$ program P, if θ is a solution to $\leftarrow G$ under constraint S_0, then for the $CLP(X)$ program $clp(P)$, θ is a solution to $\leftarrow clp(G)$ under S_0. By completeness of $CLP(X)$, the $clp(P)$-computed answers to $\leftarrow clp(G)$ represent all the solutions. However every $clp(P)$-refutation of $\leftarrow clp(G)$ can be mapped to a P-refutation of $\leftarrow G$ with the same computed answer by replacing user atoms with propagation constraints where appropriate, and, wherever those user goals are selected in the $CLP(X)$ refutation, adding two extra transitions which add the atom to the set of propagation agents, and then return it to the user goal. Thus the P-computed answers to $\leftarrow G$ also represent all the solutions.

This result is rather trivial. The more interesting question is what happens if the $GP(X)$ evaluations *commits* to transitions involving propagation. We must firstly show that completeness is not lost if we only admit derivations in which propagation constraints are unfolded last. We must secondly show that completeness is not lost if we only admit derivations in which propagation steps actually take place. We must accordingly show that, by postponing the return of propagation agents to the current goal for unfolding until it is empty, computed answers are not lost. We must then show that no computed answers are lost as a result of propagation.

The first requirement can be met at once. By modifying the switching lemma of Lloyd [Llo84] to admit constraints on any computation domain X, we conclude that the order in which goals are unfolded cannot change a $CLP(X)$ refutation into a failed derivation. Moreover the computed answer returned by the changed refutation is logically equivalent to the original computed answer. The modified $CLP(X)$ refutation maps to a $GP(X)$ refutation, where the propagation constraints are unfolded last.

We now establish two theorems showing that the insertion of extra propagation steps into a $GP(X)$ refutation cannot change its result. The first theorem states that the constraint store which includes approximation constraints added by propagation steps remains logically equivalent to the unexpanded constraint store. The second theorem states that the number of propagation steps is guaranteed to be finite. We start by establishing three lemmas.

Lemma 6 *At any transition in the refutation extended with propagation steps, the constraint store is $S \cup AC$ where S was the store at this transition in the original refutation, and AC is the set of approximation constraints added by propagation steps.*

This lemma is easily proved by induction on the transitions in the refutation.

Lemma 7 *If the original P-refutation was $< G, \emptyset, \emptyset > \Longrightarrow < \emptyset, \emptyset, S_{fin} >$. then for each atom A_i that appears in any goal in any intermediate state in this P-refutation, $P \models_X \forall.(S_{fin} \rightarrow A_i)$.*

This lemma is a simple consequence of lemma 2 above. (In particular it means that each propagation constraint is a logical consequence of the final store.)

Lemma 8 *If $< G, \emptyset, \emptyset > \Longrightarrow < \emptyset, \emptyset, S_{fin} >$ is a successful $GP(X)$ refutation without propagation steps; and if $< G1, \emptyset, \emptyset > \Longrightarrow < G2, A, S >$ is a subderivation of it; and if $< G1, \emptyset, \emptyset > \Longrightarrow < G2, A, (S \cup AC) >$ is the $GP(X)$ subderivation which results from inserting a number of propagation steps into that; then $X \models \exists.(S \wedge AC)$*

Proof

Each approximation constraint $prop(A_i, S_{old})$ added during the refutation satisfies $X \models prop(A_i, S_{old})\theta$, for every solution θ to A_i with store $S_{old} \subseteq S$. By lemma 7 above, $P \models_X \forall.(S_{fin} \rightarrow A_i)$, and $S_{fin} \supseteq S_{old}$, therefore $X \models S_{fin}\theta$ implies that θ is a solution to $\leftarrow A_i$ under S_{old}. Thus for every valuation θ such that $X \models S_{fin}\theta$, also $X \models prop(A_i, S_{old})\theta$. We conclude that
$X \models \forall(S_{fin} \rightarrow prop(A_i, S_{old}))$.

Since this holds for every approximation constraint $prop(A_i, S_{old})$ it also holds for AC which is a conjunction of such approximation constraints. Also $S \subset S_{fin}$, and so $X \models \forall.(S_{fin} \rightarrow (S \wedge AC))$. The consistency of $S \cup AC$ is now an immediate consequence of the consistency of S_{fin}.

This lemma shows that the additional approximation constraints cannot give rise to a $failure$ state in the extended derivation. Thus the addition of a finite set of propagation steps to a successful refutation yields a new successful refutation. The final result says the resulting refutation yields the same computed answer up to logical equivalence.

Theorem 9 *If* $< G, \emptyset, \emptyset > \Longrightarrow < \emptyset, \emptyset, S_{fin} >$ *is a* $GP(X)$ *refutation without propagation, and* $< G, \emptyset, \emptyset > \Longrightarrow < \emptyset, \emptyset, (S_{fin} \cup \{AC_1, \ldots, AC_n\}) >$ *is a* $GP(X)$ *refutation which differs from the first only by including a number of propagation steps, then* $X \models S_{fin} \equiv (S_{fin} \wedge \bigwedge_{i=1}^{n} AC_i)$.

Proof

Clearly $\models S_{fin}(S_{fin} \wedge \bigwedge_{i=1}^{n} AC_i)$. Taking $S = S_{fin}$ in the proof of lemma 8 above, we obtain the reverse implication: $X \models S_{fin} \rightarrow (S_{fin} \wedge \bigwedge_{i=1}^{n} AC_i)$.

If an infinite number of transitions were inserted into a refutation, the result would no longer be a refutation. In this way the completeness of $GP(X)$ could be threatened if the evaluation "committed" to each propagation step. However the second theorem states that there can never be an infinite number of propagation steps.

Theorem 10 *If* $< G, \emptyset, \emptyset > \Longrightarrow < \emptyset, \emptyset, S_{fin} >$ *is a successful* $GP(X)$ *refutation without propagation steps, the number of propagation steps that could be added to produce extra transitions is finite.*

This is guaranteed by a condition on approximation constraints, introduced in the next section, which ensures that any infinite sequence of propagations would produce an inconsistent constraint store. In other words an infinite sequence of propagations could indeed occur in derivations which would have failed anyway, but not by inserting propagations into a successful refutation.

3.4 A Specification of Generalised Propagation

3.4.1 Some Conditions on Approximation Constraints

The information extracted from a single propagation constraint is informally the best approximation to all its answers. To make this notion formal we first introduce a partial ordering on interpreted constraints by logical implication; that is if A implies B we write $A \sqsubseteq B$. Thus logically stronger constraints are below logically weaker constraints in our ordering. Notice that this is an ordering on the logical denotations of the formulae, not the formulae themselves, thus all logically equivalent constraints are equal. Since approximation constraints are a subclass of the interpreted constraints, this ordering defines a subordering on approximation constraints.

We shall now impose a few conditions on the approximation constraints.

- They should include $true$ and $false$.
- Over the domain X, every consistent strictly decreasing sequence of approximation constraints whose free variables belong to a fixed finite set, should be finite.

The first condition merely ensures that every set of interpreted constraints has at least one upper bound ($true$). The least upper bound can be used to approximate the sets of solutions to a propagation constraint. Moreover every unsatisfiable propagation constraint can be revealed to be so (since approximated by $false$).

The second condition ensures that successful propagation sequences terminate. If $AC_1, AC_2, \ldots$ are the approximation constraints added by a sequence of propagation steps, then by our definition of a transition, no AC_k is logically implied by $\bigwedge_{i=1}^{k-1} AC_i$. Since constraints are closed under conjunction, each such conjunction is itself an approximation constraint, and the sequence of conjunctions is decreasing under our ordering. The second condition ensures that this sequence stabilises, if it is consistent.

In fact every countable set of approximation constraints can be mapped to a decreasing sequence in the same way. Consequently any such set is either inconsistent, in which case its greatest lower bound is $false$, or else it is consistent in which case the sequence stabilises and we have a greatest lower bound which is itself an approximation constraint.

Recall that the underlying domain is not necessarily defined by a theory. For example the Herbrand domain of logic programming is defined by an algebra. Consequently the compactness theorem does not apply: there are indeed infinite sequences which are inconsistent with the Herbrand domain (eg. $\exists Y.X = f(Y)$, $\exists Y.X = f(f(Y))$, ...), for which every finite subsequence is consistent.

Similarly over the integers, $X \geq 1, X \geq 2, \ldots$ is an infinite sequence which is inconsistent, though every finite subsequence is consistent. In fact there is no consistent strictly decreasing infinite sequence $T1_i \leq T2_i : i \in I$ or $T1_j \geq T2_j : j \in J$ where the $T1_i$,$T2_i$,$T1_j$ and $T2_j$ are integers or variables from a fixed set of free variables. Thus the class of approximation constraints $T_1 \leq T_2$ and $T_1 \geq T_2$ over the integers does satisfy our conditions on approximation constraints, since also $true \equiv (1 \geq 1)$ and $false \equiv (1 \geq 2)$.

3.4.2 Information Extracted by a Single Propagation Step

We are now in a position to specify precisely the result of a single propagation step on a constraint.

Definition 5 *For a $GP(X)$ program P, the constraint $prop(A_i, S_{old})$ extracted by a single propagation step on a propagation agent A_i with constraint store S_{old} is the smallest approximation constraint AC_i, whose free variables are also free in A_i, and which is satisfied by all the solutions to A_i under S_{old}.*

The intuition behind this definition is to extract as much information as possible from the propagation constraint without excluding any solutions. The restriction that the free variables in the approximation constraint are also free in the propagation agent is, necessary to avoid potentially infinite approximation sequences involving more and more variables. Notice that this definition depends only on the declarative semantics of the program and the agent. The result of propagation is independent of the precise program definition for the predicate of A_i.

For example, consider the result of propagation on the constraint $p1(X, Y)$ defined as above
$p1(3, 0) \leftarrow$
$p1(1, 1) \leftarrow$
$p1(2, 3) \leftarrow$
with constraint store $\{Y \geq 1\}$. Propagation on $p1(X, Y)$ yields the tightest approximation constraint which is implied by both $X = 1 \wedge Y = 1$ and $X = 2 \wedge Y = 3$. This is $1 \leq X \leq 2 \wedge 1 \leq Y \leq 3 \wedge X \leq Y$. This is a simple example showing the difference between approximation constraints and "labels" as described by Davis [Dav87]. The last atom $X \leq Y$ cannot be expressed by any label on individual variables.

We now show that propagation on a given agent is monotonic in the sense that if there is more information in the constraint store then more information will be extracted by propagation.

Lemma 11 *Let A_i be a propagation agent, and S_1 and S_2 be constraint stores. If $S1 \sqsubseteq S2$ (i.e. $S1$ is more constrained than $S2$), then $prop(A_i, S_1) \sqsubseteq prop(A_i, S_2)$.*

Proof

The condition $S1 \sqsubseteq S2$ implies that $\models S1\theta \rightarrow S2\theta$ for any valuation θ. If θ is a solution to A_i under constraint S_1, then $P \models_X A_i\theta$ and $X \models S_1\theta$. However we immediately conclude that $X \models S_2\theta$, and so θ is also a solution to A_i under S_2. Therefore, by definition, $X \models$

$prop(A_i, S_2)\theta$. However $prop(A_i, S_1)$ is the *least* approximation constraint satisfied by every solution θ to A_i under S_1, and we can conclude that $prop(A_i, S_1) \sqsubseteq prop(A_i, S_2)$.

As an example of this property consider $prop(p1(X,Y), Y \geq 0)$ and $prop(p1(X,Y), Y \geq 1)$, where $p1$ is defined as above. $Y \geq 1$ is more restrictive that $Y \geq 0$, so $(Y \geq 1) \sqsubset (Y \geq 0)$.
$prop(p1(X,Y), Y \geq 0) = (1 \leq X \leq 3 \wedge 0 \leq Y \leq 3) = AC1$, and
$prop(p1(X,Y), Y \geq 1) = (1 \leq X \leq 2 \wedge 1 \leq Y \leq 3 \wedge X \leq Y) = AC2$.
Clearly the tighter constraint store excludes more solutions and allows a tighter approximation constraint to be extracted, and indeed $AC2 \sqsubset AC1$.

A propagation agent A_i is *idle* in a state $< G, A, S >$ with $A_i \in A$, if no transitions are possible by propagating on A_i. This can be formalised as:

Definition 6 *A propagation agent $A_i \in A$ is idle in $< G, A, S >$ if for all subsets $S_{old} \subseteq S$,*
$X \models S \rightarrow prop(A_i, S_{old})$

The previous lemma says that A_i is idle in $< G, A, S >$ if (and only if) $X \models S \rightarrow prop(A_i, S)$.

3.4.3 Propagation Sequences

In this subsection we shall take an initial state $< G, A, S_0 >$ and we shall consider what can result from a sequence of propagation steps, assuming no other transitions take place. Thus each state that we shall consider has the same goal G and the same set of propagation agents A.

If every agent $A_i \in A$ is idle, then no further propagation steps can take place. In our framework a *propagation sequence* is a derivation $< G, A, S_0 > \Longrightarrow < G, A, S_{fin} >$ comprising solely propagation steps and in whose final state all the propagation agents are idle. In this section we shall show that for any initial state $< G, A, S_0 >$ all propagation sequences yield, up to logical equivalence, the same final state $< G, A, S_{fin} >$.

To this purpose we first define an operator fix which, for any given propagation agent A_i maps constraint stores to constraint stores. $fix(A_i, S_0)$ is the final store which results from propagating on A_i until it is idle.

In fact it can be shown that A_i is idle in $S_0 \wedge prop(A_i, S_0)$:

Lemma 12 *For any constraint store S_0 and propagation agent A_i,*
if $S = (S_0 \cup prop(A_i, S_0))$ then $X \models S \rightarrow prop(A_i, S)$

Proof

Let $AC_i = prop(A_i, S_0)$. AC_i is satisfied by every solution θ to A_i under S_0. Therefore, for every such θ, $X \models S_0\theta \wedge AC_i\theta$. Thus θ is also a solution to A_i under $(S_0 \wedge AC_i) = S$. But, as shown above, every solution to A_i under S is a solution to A_i under $S_0 \subseteq S$. Therefore the result of propagation on A_i with S remains $AC_i \subseteq S$.

By monotonicity it then follows that no sequence of propagations on A_i can produce more information than $prop(A_i, S_0)$. Thus we can define $fix(A_i, S_0)$ very simply as $S_0 \wedge prop(A_i, S_0)$.

We can now establish three properties of the operator fix for any given agent A_i.

Theorem 13 *For any propagation agent A_i, $fix(A_i)$ is a monotonic, decreasing and idempotent operator on constraint stores.*

Using these properties of fix it is simple to show that for any propagation agents A_i and A_j,
$X \models fix(A_i, fix(A_j, S_0)) \equiv fix(A_j, fix(A_i, S_0))$.

Since every propagation sequence is finite, as shown above, we are sure to reach a state where all the constraints are idle. The above result shows that sequences of propagations on each agent can be reordered at will without changing the final result. Moreover by monotonicity it follows that the same final result is still obtained however the individual propagation steps on the different agents are interleaved.

Theorem 14 *For any given initial state* $< G, A, S >$*, every propagation sequence produces the same final state* $< G, A, S_{fin} >$ *up to logical equivalence.*

An example propagation sequence, using predicates $p1$ and $p2$ defined as before is:
$prop(p1(X,Y), \emptyset) = (1 \leq X \leq 3 \wedge 0 \leq Y \leq 3) = AC1$.
Moreover $prop(p2(X,Y), AC1) = (1 \leq X \leq 3 \wedge 1 \leq Y \leq 2 \wedge Y \leq X) = AC2$.
Finally $prop(p1(X,Y), AC2) = (1 \leq X \leq 1 \wedge 1 \leq Y \leq 1) = AC3$.
These propagations produce the following propagation sequence:
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, \emptyset > \mapsto$
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, AC1 > \mapsto$
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, AC1 \cup AC2 > \mapsto$
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, AC1 \cup AC2 \cup AC3 >$.
It is interesting to follow different propagation sequences that lead to the same derivation
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, \emptyset > \Longrightarrow < \emptyset, \{p1(X,Y), p2(X,Y)\}, \{1 \leq X \leq 1, 1 \leq Y \leq 1\} >$.

3.5 Aspects of Programming in GP(X)

3.5.1 Unfolding Propagation Constraints

It was stated in section 3.1 above that logically a propagation agent A_i is equivalent to the user atom A_i. However neither a single propagation step nor a whole propagation sequence is guaranteed to extract approximation constraints logically equivalent to A_i.

As much information is extracted as can be expressed using approximation constraints, but in general there may remain further information not expressible as approximation constraints. In particular if a pair of goals are inconsistent, independent propagation will not necessarily reveal this. Over the integers, approximated as before by $\leq$ and $\geq$, consider the propagation agent
$r(X,Y)$ defined by the facts
$r(1,2) \leftarrow$
$r(2,3) \leftarrow$
$r(3,1) \leftarrow$
The information extracted in $fix(r(X,Y), \emptyset)$ is that X and Y lie between 1 and 3. Now if r is defined as above and s is defined by the clauses
$s(1,3) \leftarrow$
$s(3,2) \leftarrow$
$s(2,1) \leftarrow$
then propagation on the two agents $r(X,Y)$, and $s(X,Y)$ will produce no more than $1 \leq X \leq 3, 1 \leq Y \leq 3$ in the final constraint store. However the result of the query $\leftarrow r(X,Y), s(X,Y)$ is, of course, failure.

This example shows that for soundness of $GP(X)$ it is necessary that evaluation should not terminate until the constraints in the constraint store imply the truth of the propagation agents. This is enforced in our operational semantics by defining a success state to be one in which the set of propagation agents is empty, so that the agents are guaranteed to be returned to the goals and unfolded.

In traditional constraint propagation systems, the propagation is complemented by search routines which non-deterministically instantiate problem variables to values in their domains. This "labelling" enables further propagation to take place, and eventually ensures that the propagation constraints are satisfied. The use of propagation agents additionally as goals, treated by unfolding, has the effect of adding to the constraint store the appropriate basic constraints in the domain of computation to satisfy the propagation constraint. Thus it is an appropriate (and automatic) generalisation of labelling in finite domains. In the crossword program, for example, the labelling is done solely by unfolding the propagation constraints each time that no further propagation is possible. Nevertheless in general the programmer is also free

to write his own labelling procedures, and they will be treated before any unfolding of the propagation constraints is allowed to begin.

3.5.2 Parallel Evaluation of Propagation Steps

Theorem 14 above frees the user from all concerns about the scheduling of propagation steps. Propagation may be sequential, in which case each propagation step uses the latest constraint store, or the extraction of information from propagation agents and constraint stores may be performed in parallel. For example propagation on two agents $p1(X,Y)$ and $p2(X,Y)$ in the empty store performed sequentially ($p1(X,Y)$ then $p2(X,Y)$) yields the following two transitions,
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, \emptyset >\mapsto$
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, (1 \leq X \leq 3 \wedge 0 \leq Y \leq 3) >\mapsto$
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, (1 \leq X \leq 3 \wedge 1 \leq Y \leq 2 \wedge Y \leq X) >$.

However if the calculation of $prop(p1(X,Y), \emptyset)$ and $prop(p2(X,Y), \emptyset)$ are performed in parallel, the following transitions might take place:
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, \emptyset >\mapsto$
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, (1 \leq X \leq 3 \wedge 0 \leq Y \leq 3) >\mapsto$
$< \emptyset, \{p1(X,Y), p2(X,Y)\}, (1 \leq X \leq 3 \wedge 1 \leq Y \leq 3) >$.
The example shows that concurrent propagation may converge more slowly on the final fixpoint. However, it is ultimately guaranteed to converge to the same fixpoint as sequential propagation.

If other transitions take place before all the agents are idle, the computed answers remain correct, as shown above, but the search tree may be greater than necessary. (After the two steps above there remain two clauses for both $p1$ and $p2$ that are consistent with the constraint store.) Notice that "concurrent propagation" can still be taking place even while unfolding transitions are made.

3.6 Termination in $GP(X)$

Termination of the search for answers to a propagation constraint is not guaranteed. Non-termination due to unfolding is inherited from $CLP(X)$: in practice the programmer is responsible for ensuring that unfolding should terminate. Just as any user goal in $CLP(X)$, a propagation constraint in $GP(X)$ can only be evaluated after the clause in whose body it appears has been unfolded. In this sense $GP(X)$ is no different from $CLP(X)$.

There are two differences. Firstly all answers to a propagation constraint are generally required instead of just one as in $CLP(X)$. Of course backtracking will generally imply that many answers to a goal must be found in $CLP(X)$ as well. The theoretical problem remains that in $CLP(X)$ every answer lies at the end of a terminating success branch, whilst the requirement during propagation for *all* answers to a propagation constraint implies that *any* infinite branch in the search tree can cause non-termination of a propagation step.[3]

Secondly, a propagation constraint may be evaluated and re-evaluated many times in $GP(X)$. Luckily this does not alter the termination behaviour of the program. The reason is that on later evaluations the constraint store is logically at least as strong as before. Consequently the later evaluations may benefit from extra pruning of some branches, but no new infinite branches can arise.

Implementations, like ours, that postpone unfolding until all propagation agents are idle, may therefore sacrifice completeness waiting for a propagation step to terminate. However our framework admits propagation taking place in parallel with unfolding, and in this case completeness is preserved at the risk of certain branches in the search tree being "cut off" later than necessary.

[3] But it frequently does not, as we show below in section 3.7.3.

3.7 Topological Branch and Bound

We use the name "topological branch and bound" as a description of our technique for extracting approximation constraints from a propagation agent. The technique is based on a form of branch and bound search through the answers to the propagation agent, where the bound is just a lower bound in our ordering on approximation constraints.

3.7.1 Evaluating Propagation Constraints

Conceptually, the calculation of the information $prop(A_i, S)$ extracted in a propagation step requires

- finding all the computed answers to the goal A_i with store S
- finding the smallest approximation constraint which is an upper bound for the set of computed answers

Lemma 15 *An approximation constraint AC_i is an upper bound on the set of computed answer to a query $\leftarrow G$ with constraint store S if and only if AC_i is satisfied by all solutions to $\leftarrow G$ under S.*

Proof

> If AC_i is satisfied by all solutions to $\leftarrow G$ under S, then by soundness it is an upper bound on the computed answers. If AC_i is not satisfied by some solution θ, then by completeness there is a computed answer Ans such that $X \models Ans\theta$ and it is not the case that $Ans \sqsubseteq AC_i$. Therefore AC_i is not an upper bound on the computed answers.

Using this result, when calculating $prop(A_i, S)$, the system can use the set of computed answers to A_i with S. Notice, though, that computed answers are defined only for states in which the set of propagation agents is empty. As noted in section 3.2.3 above, we can allow propagations to be present as long as they are not used to perform propagation steps. Therefore, when computing answers to a propagation agent, the remaining propagation agents are temporarily "suspended".

3.7.2 An Example

To illustrate the topological branch and bound algorithm we shall use as computation domain the Herbrand universe. The basic constraints are equations $T1 = T2$, and we shall also use equations as approximation constraints. As an example, using equations as approximation constraints, the best approximation for the two answers $X = a \wedge Y = a$ and $X = b \wedge Y = b$ is $X = Y$.

We now describe the calculation of $prop(t(X, Y, Z), X = a)$, using the following program definition.
$t(b, c, d) \leftarrow$
$t(a, b, b) \leftarrow$
$t(a, c, c) \leftarrow$
$t(a, W, W) \leftarrow tt(W)$
$t(a, b, c) \leftarrow$
$t(a, c, d) \leftarrow$
We assume the predicate tt also has a program definition, but we will not need it to perform propagation on $t(X, Y, Z)$!

The initial approximation constraint AC_0 is set to $false$.

After each answer Ans to the goal $\leftarrow t(X, Y, Z)$ is retrieved it is first checked for consistency with the constraint store $X = a$. If $Ans \wedge X = a$ is unsatisfiable, then the answer is thrown away. The first answer is $X = b \wedge Y = c \wedge Z = d$. This is indeed inconsistent with $X = a$ and the answer is thrown away.

If no consistent answers are found, then constraint propagation has detected an inconsistency, and the propagation sequence terminates producing the approximation constraint $false$. In our example, however, there are further answers which are consistent with $X = a$.

When a consistent answer Ans_i is found it is added to the current best approximation, and the pair $\{AC_i, A_i\}$ is approximated yielding a new approximation constraint AC_{i+1}. The next consistent answer to $t(X, Y)$ is $X = a \wedge Y = b \wedge Z = b$, and this is also the next approximation. Call it AC_1. The following (consistent) answer to $t(X, Y)$ is $X = a \wedge Y = c \wedge Z = c$. The best approximation to $X = a \wedge Y = b \wedge Z = b$ and $X = a \wedge Y = c \wedge Z = c$ is $X = a \wedge Y = Z$. Call it AC_2.

During the search for an answer, basic constraints are added to a local constraint store LS. If at any stage $LS_{i+1} \rightarrow AC_i$, then the local search is abandoned. Search for new answers continues by choosing other clauses to unfold. After unfolding the next clause, the local constraint store LS_3 contains $\{X = a, Y = W, Z = W\}$. Although the refutation is not yet complete, and in fact there may be no consistent answers to $tt(W)$, $X \models \forall.(LS_3 \rightarrow AC_2)$. Consequently it is unnecessary to search further: any answer Ans obtained via this clause will be logically tighter than the current approximation AC_2, and therefore AC_2 will remain the tightest approximation to AC_2 and Ans.

Propagation terminates as soon as the approximation constraint AC_i is implied by the constraint store, $X = a \rightarrow AC_i$. In this case no new information could be extracted, and so $prop(t(X, Y, Z), X = a) = true$. The next clause is $t(a, b, c)$, and yields answer $X = a \wedge Y = b \wedge Z = c$. The best approximation to this answer and $X = a \wedge Y = Z$ is simply $X = a$. Call it AC_3. This is no stronger than the original constraint store, and therefore no new information has been extracted by propagation on $t(X, Y, Z)$. Although there are further clauses defining t, there is no need to search further, and the calculation of $prop(t(X, Y, Z), X = a)$ terminates producing the approximation constraint $true$.

Otherwise propagation terminates when there are no further alternative clauses to unfold. Then the current approximation constraint is added to the constraint store. Thus $prop(t(X, Y, Z), X = b)$ produces $X = b \wedge Y = c \wedge Z = d$.

3.7.3 Decision Procedures

Thus for $GP(X)$ three decision procedures are required.

- For checking consistency, the system must support an effective decision procedure for interpreted constraints over X (the same procedure is required for $CLP(X)$). This requires a decision procedure to establish a proof of $X \models \forall.(\bigwedge_{i=1}^{n} Ci \rightarrow false)$ where the C_i are atomic interpreted constraints.
- For extracting approximations, the system must additionally support an effective procedure for producing the smallest approximation constraint which is an upper bound for an answer and a current approximation. The approximation AC for an agent A_i must satisfy $X \models \forall.(\exists_{\setminus Ai} \bigwedge_{i=1}^{n} C_i) \rightarrow AC$, where the C_i are interpreted constraints.
- In section 3.2.3 above another effective decision procedure was mentioned, to determine if an approximation constraint is a logical consequence of the current store. This is needed again here to test if the current approximation constraint AC is already implied by the local constraint store collected on a certain branch of the search tree. The formula to be proved is $X \models \forall.(\bigwedge_{i=1}^{n} C_i \rightarrow AC)$, where the C_i are interpreted constraints.

3.7.4 Interleaving Answering and Approximation

In practice the evaluation of propagation constraints interleaves the finding of individual answers and their generalisation. To make this possible we assume that our procedure for extracting approximations can return the smallest approximation constraint which is an upper bound for an answer and an approximation constraint. We now prove that to approximate a finite set of computed answers it is possible perform the approximations pairwise.

Lemma 16 *If $A2$ is the best approximation of $\{D1, D2\}$ and $A3$ is the best approximation of $\{A2, D3\}$, then $A3$ is the best approximation of $\{D1, D2, D3\}$.*

Proof

> Call AC the best approximation for $D1, D2, D3$. Clearly $A3$ approximates $D1$ and $D2$ and $D3$, therefore $AC \leq A3$. Moreover AC approximates $D1, D2$, so $A2 \leq AC$. Consequently AC approximates $A2$ and $D3$, so $A3 \leq AC$. Therefore $AC \equiv A3$.

This lemma generalises to finite sets of answers by induction.

Recall that in calculating $prop(t(X, Y, Z), X = a)$ we used pairwise approximations to extract $X = a$ as the best approximation for the three answers $X = a \wedge Y = b \wedge Z = b$ and $X = a \wedge Y = c \wedge Z = c$ and $X = a \wedge Y = b \wedge Z = c$.

3.7.5 Cutting All Remaining Branches

We now describe two optimisations which fit naturally into the evaluation of propagation constraints. Both optimisations depend upon the interleaving of answering and approximation. At any point in the evaluation of a propagation constraint the system has available

- the constraint store S
- the current approximation constraint AC which is the smallest approximation constraint which is an upper bound for the answers found so far

The current approximation constraint can be used just like the current best cost in a branch and bound search. However it can also be used, in a way not available in branch and bound, to prune off all the remaining branches of the search tree.

Using the procedure which decides if an approximation constraint is implied by the constraint store S, it is possible to prune the evaluation of a propagation constraint by interleaving the finding of answers and generating new approximation constraints AC; and terminating the computation as soon as $X \models S \rightarrow AC$.

Recall that we used the interleaving to extract the intermediate approximation $AC_3 = X = a$ for $prop(t(X, Y, Z), X = a)$, At this point the current best approximation was already as strong as the original constraint store $X = a$, and therefore the search for further answers stopped.

This optimisation is very important for propagation constraints defined by large numbers of clauses. For such constraints it is often easy to find a few solutions, but very expensive to find them all. Its significance is illustrated by the crossword compilation application described below 4.1.1.

3.7.6 Cutting off the Current Branch

When exploring a single branch the system collects locally a set of basic constraints extracted during the unfolding of clauses. The conjunction of all the basic constraints extracted along a branch goes to make up a single answer to the propagation constraint. If this answer is logically stronger than the current approximation constraint (which approximates all the answers found so far), then it cannot affect the final result.

Branch and bound search benefits from the observation that there is no need to explore to the end a branch that is already more expensive than the current best branch. In evaluating a propagation constraint the same observation applies: there is no need to explore further if the local constraints gathered on a branch are already logically stronger than the current approximation constraint.

The required decision procedure is the same as that for determining if a propagation agent is idle. We need to determine if the current approximation constraint is implied by a set of interpreted constraints.

Recall that in evaluating $prop(t(X,Y,Z), X = a)$, the current best approximation AC_2, at the time when the clause $t(a,W,W) \leftarrow tt(W)$ was unfolded, was $X = a \wedge Y = Z$. After unfolding the clause the local constraint store LS contained $X = a \wedge Y = W \wedge Z = W$. Since $X \models \forall.LS \rightarrow AC_2$, any further answer Ans along this branch was bound to satisfy $Ans \rightarrow LS \rightarrow AC_2$. Consequently AC_2 was also the best approximation for AC_2 and Ans for each such Ans. Since pairwise approximation is equivalent to approximating all the answers at once, AC_2 was guaranteed to remain the next best approximation after adding all the answers (whether there are any or not!) using this clause.

This optimisation proves to be very valuable for propagation constraints defined by recursive clauses. This will be illustrated using the *member* predicate in section 4.2 below.

4 Some Instances of GP(X)

Two implementations of generalised propagation over the Herbrand universe have been completed. One implementation is in the Elipsys system [DSVX91] which runs on a parallel machine. Using finite domains as the approximation language, it has achieved good speedups on disjunctive scheduling programs [PV92] and for temporal reasoning [Lev91].

In the paper we describe the other implementation which is embedded in a sequential prolog compiler system. It is an implementation of generalised propagation over the Herbrand universe $GP(HU)$, and it is called *Propia*. Propia extracts information about equalities from propagation constraints, and it offers a number of approximation languages some of which will be described below. Propia is implemented in Sepia [MAC+89] with the help of some special added built-ins.

An important requirement for the efficient implementation of generalised propagation is a sophisticated coroutining facility. Sepia has a special built-in delay condition which enables a delayed goal to be woken as soon as any of its variables are "touched" during unification: this includes the unification of two variables in the clause as well as further instantiation. The delayed clause can then be redelayed again on the same condition. Such a facility provides the ideal support for propagation agents which need be checked if and only if any of their variables are "touched" in this way.

During the calculation of a single propagation step it is necessary to suspend other propagation agents and to collect new constraints into a local constraint store. Both these requirements are satisfied in Propia by simply renaming the variables in the propagation agent A_i to new variables in a copy AA_i of the agent. Thus no agents are woken when AA_i is evaluated, and local answers are expressed as bindings on the new variables in AA_i.

Of special interest is the implementation of the topological branch and bound. If the propagation constraint is $p(X,Y,Z)$ the current best approximation is held as a term. Thus the approximation $X = a \wedge Y = Z$ is held as the term $p(a,W0,W0)$ (where $W0$ is a new variable). Similarly answers are represented by terms. Thus, for example, $p(a,b,c)$ might represent an answer to the goal $\leftarrow p(X,Y,Z)$. After retrieving an answer the new approximation is obtained by anti-unifying the answer with the previous approximation. The result of anti-unifying $p(a,b,c)$ with $p(a,W0,W0)$, for example, is $p(a,W1,W2)$.

Another built-in predicate is used to prune the search as soon as the answer is more constrained than the current best approximation. This built-in checks for inequality between two terms. Specifically it proves $\forall_{\setminus T1}(\neg T1 = T2)$. Thus if $p(X,Y,Z)$ is the agent being used for propagation, and if $p(a,W,W)$ is the current best approximation, it checks that $\forall W.\neg p(X,Y,Z) = p(a,W,W)$. This built-in delays, and redelays, on the free variables until the goal is a consequence of the current constraints, or contradicts them. Thus the goal $\leftarrow \forall W.\neg p(X,Y,Z) = p(a,W,W)$ delays. However if X,Y and Z become instantiated it is woken. The instantiated goal $\forall W.\neg p(a,b,c) = p(a,W,W)$ succeeds, but the instantiated goal $\forall W.\neg p(a,b,b) = p(a,W,W)$ fails (since indeed the terms are equal for $W = b$).

Now each time an answer to AA_i is found, and a new best approximation is extracted and it is encoded as a term AC_i. If the term AC_i is a variant of the original propagation agent A_i, the search terminates producing no new information. Otherwise the disequality $\forall_{\setminus AA_i}(\neg AA_i = AC_i)$ is added as a (delayed) goal, and the search restarts. If AC_i approximates all the answers to AA_i, then the search will fail, since all answers will be pruned by the disequality. In this case AC_i is indeed the result of propagation on A_i.

4.0.7 Evaluating Propagation Sequences

In the case of finite domain propagation, the procedure for performing propagation on a single constraint is called $REVISE$ [MF85]. Essentially the evaluation of a propagation sequence for generalised propagation can be obtained from the AC-3 algorithm of Mackworth [Mac77] by replacing $REVISE$ with topological branch and bound.

A feature of AC-3 is that after propagating on a constraint C, C is removed from the list of constraints to be dealt with in the current propagation sequence. C is only added to the list again if some of its variables are affected by propagation on other constraints. For the correctness of AC-3 it is therefore necessary that propagation on a single constraint is itself a fixpoint operation, and as we showed above in section 3.4.3 above, $prop(A_i, S) \cup S = fix(A_i, S)$ is already a fixpoint. This condition is not satisfied by *relaxed tell* [HD91], which is an abstraction of generalised propagation (see below 5.2).

4.0.8 Propagation as Consistency Checking

Various alternative approximation languages are available in Propia. The more expressive the approximation language the more information is extracted, but the costlier the propagation.

One very simple approximation language has just two approximation constraints: $true$ and $false$. We call this the *consistency* approximation language. With this language the result of propagation on a constraint is either nothing (in case an answer was found) or failure (in case none could be found). The behaviour of the crossword program with this language is to use each constraint as a continual check on the choices made so far. This ensures that no inconsistent choices are made, but that no "active" constraint propagation is done.

The advantage of using such a trivial approximation language is that in this case topological branch and bound is very effective in optimising the evaluation of propagation constraints. Suppose a certain constraint is being evaluated for propagation. As soon as a single answer is found, the current approximation constraint (approximating the answers found so far) becomes $true$. Since $true$ is implied by the current constraint store (since it is implied by any constraint store) propagation terminates immediately.

Clearly with the trivial approximation language generalised propagation is efficient and economic in its basic operations. Assuming the propagation agents are all defined by flat relations, as is the normal assumption for constraint propagation problems, then when a set of n variables become instantiated during search it suffices to check once each of the agents involving an affected variable. In this context the unification problem reduces to a matching problem which has constant cost for each tuple checked. Thus the worst case complexity for consistency checking is $e * d$ where e is the number of problem constraints and d the number of tuples defining the largest constraint.

However Propia only checks constraints woken by the newly instantiated variables. In the crossword application (described in the next section), for example, only two constraints are ever woken by the instantiation of a single letter, however big the crossword. Sepia offers indexing on all arguments, orders the indices by their effectiveness, and in checking a partially instantiated query it uses the most effective index amongst those argument that are instantiated; and, of course, the uninstantiated arguments cannot cause consistency to be violated. Consequently, for example, an average propagation step in the crossword application with a 25000 word lexicon takes only ten milliseconds on a Sun4.

Consistency checking offers an alternative to intelligent backtracking, in this sense. If every user goal is annotated as a propagation constraint, then the propagation prevents any further (irrelevant) choices being made if any other goal is already unsatisfiable. This is because the failure is detected immediately when attempting propagation.

4.1 GP(Datalog)

Datalog is logic programming without functions. The basic constraints in Datalog are equalities, $X = c$ or $X = Y$ where c is a constant and X and Y are variables. There is no termination problem for Datalog queries, and thus propagation steps can always be made to terminate. Moreover for a propagation

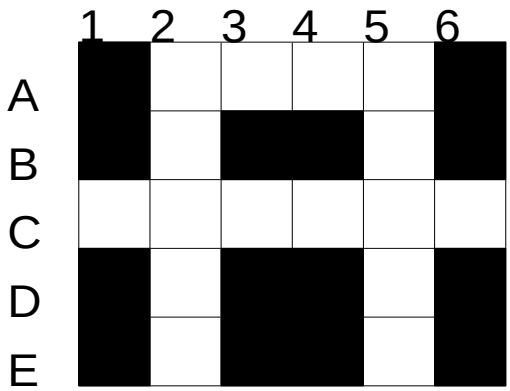


Figure 4.1: A toy crossword example

agent with n variables, each propagation step reduces the number of distinct variables by one (either a variable is instantiated to a constant, or two variables are unified). Consequently there is a maximum of n propagation steps on each propagation agent. A consequence is that far less propagation steps are performed by $GP(Datalog)$ than would be necessary to enforce arc consistency over a suitable domain. Crossword compilation provides evidence for this.

4.1.1 Crossword Compilation

Crossword compilation is an application of $GP(Datalog)$. We now give a toy crossword compilation to illustrate a $GP(Datalog)$ evaluation. The crossword to be filled is:

The dictionary is encoded as a set of facts:

$w5(b,r,a,k,e) \leftarrow$ $w4(b,u,m,p) \leftarrow$ $w6(b,e,t,t,e,r) \leftarrow$
$w5(b,l,o,k,e) \leftarrow$ $w4(p,l,a,y) \leftarrow$ $w6(c,a,n,n,o,n) \leftarrow$
$w5(s,t,e,a,m) \leftarrow$ $w4(f,r,e,e) \leftarrow$ $w6(w,e,a,l,t,h) \leftarrow$
$w5(c,r,e,a,m) \leftarrow$ $w4(s,t,o,p) \leftarrow$ $w6(d,e,a,r,t,h) \leftarrow$
$w5(p,a,t,c,h) \leftarrow$
$w5(p,i,t,c,h) \leftarrow$

The problem is posed as the following query

$\leftarrow$ constraint $w4(A2,A3,A4,A5)$,
constraint $w6(C1,C2,C3,C4,C5,C6)$,
constraint $w5(A2,B2,C2,D2,E2)$,
constraint $w5(A5,B5,C5,D5,E5)$

Recall that a $GP(X)$ refutation is a sequence is state transitions, where each state is a triple $< G,A,S >$ comprising the current goal G, set of propagation agents A and constraint store S. For the above query, the refutation starts with an empty set of propagation agents and an empty constraint store. First all the propagation constraints are moved into the set of propagation agents yielding the new state:

$<$ $\emptyset$,
$\{$ $w4(A2, A3, A4, A5)$,
$w6(C1, C2, C3, C4, C5, C6)$,
$w5(A2, B2, C2, D2, E2)$,
$w5(A5, B5, C5, D5, E5)$ $\}$,
$\emptyset$ $>$

which has an empty goal, four propagation agents, and an empty constraint store.

Next propagation is attempted on all the agents (i.e. the blank words). However no new information is elicited. The first agent, corresponding to the blank word $w4(A2, A3, A4, A5)$, is then returned to the goal. This time it is no longer a propagation constraint, but a user atom. The resulting state is:

$<$ $\{w4(A2, A3, A4, A5)\}$,
$\{w6(C1, C2, C3, C4, C5, C6), w5(A2, B2, C2, D2, E2), w5(A5, B5, C5, D5, E5)\}$,
$\emptyset$ $>$

The atom in the goal is unfolded, using the first clause in its program definition, yielding:

$<$ $\emptyset$,
$\{w6(C1, C2, C3, C4, C5, C6), w5(A2, B2, C2, D2, E2), w5(A5, B5, C5, D5, E5)\}$,
$\{A2 = b, A3 = u, A4 = m, A5 = p\}$ $>$

Now propagation is attempted again on all the agents. Using $A5 = p$, propagation on $w5(p, B5, C5, D5, E5)$ yields $C5 = t \wedge D5 = c \wedge E5 = h$. Using $C5 = t$, propagation on $w6(C1, C2, C3, C4, t)$ yields $C2 = e \wedge C3 = a \wedge C6 = h$. Now propagation on $w5(b, B2, e, D2, E2)$ yields $false$, and the system backtracks to the unfolding of $w4(A2, A3, A4, A5)$.

The choices "play" and "free" are similarly proved inconsistent by propagation, so finally the evaluation reaches the state:

$<$ $\emptyset$,
$\{w6(C1, C2, C3, C4, C5, C6), w5(A2, B2, C2, D2, E2), w5(A5, B5, C5, D5, E5)\}$,
$\{A2 = s, A3 = t, A4 = o, A5 = p\}$ $>$

Propagation yields, as before, $C5 = t \wedge D5 = c \wedge E5 = h \wedge C2 = e \wedge C3 = a \wedge C6 = h$. Now propagation on $w5(s, B2, e, D2, E2)$ yields $B2 = t \wedge D2 = a \wedge E2 = m$. The crossword is completely filled except for three letters. The state is as follows:

$<$ $\emptyset$,
$\{w6(C1, C2, C3, C4, C5, C6), w5(A2, B2, C2, D2, E2), w5(A5, B5, C5, D5, E5)\}$,
$\{A2 = s, A3 = t, A4 = o, A5 = p, B2 = t, C2 = e, D2 = a, E2 = m,$
$C5 = t, D5 = c, E5 = h, C3 = a, C6 = h\}$ $>$

The propagation agents are now, one by one, returned to the goal and unfolded. The constraint store precludes all choices except ones that lead to a solution. Thus the four solutions are found without further backtracking.

In real crossword grids, with real dictionaries, very little propagation is possible until the system starts to guess words that instantiate the second or third letter in an intersecting word. In these early stages the calculation of propagation steps quickly terminates because the tightest approximation soon becomes $true$.

As the crossword fills up, the propagation begins to produce information which ensures no bad choices can be made later. At this point propagation sequences begin to grow in length, as information extracted from one constraint enables further information to be extracted from others.

To sum up, little work is invested in generalised propagation by the system until it actually starts to be useful. Evidence for the naturally good behaviour of generalised propagation on crossword compilation is this. The crossword program sketched above is perfectly naive. In fact a meta-program has been written which takes any crossword drawn as a grid and generates such a program automatically. Yet generalised propagation applied to the resulting program happens to yield a crossword compilation algorithm very similar to one developed specially for crosswords and described in [Ber87]. On a Sun4 workstation, with a 25000 word lexicon, a crossword grid from the International Herald Tribune can be compiled by Propia in 90 seconds.

4.1.2 Equalities Between Variables

For the crossword application above, the only useful information concerns values for variables (expressed as an equality between a variable and a constant). In this section we shortly demonstrate the usefulness of extracting information about equalities between variables. Applications where such information is important include those involving boolean variables, such as circuit design, analysis and testing, and propositional satisfiability problems.

Such applications involve complex boolean functions describing the behaviour of, for example, circuit components which are already analysed. Each such function can be used immediately as a propagation constraint. Let us choose the very simple "and-gate", which appeared in section 1.4 above, to illustrate the following discussion. Its behaviour can be described using four clauses:
$and(true, true, true) \leftarrow$
$and(true, false, false) \leftarrow$
$and(false, true, false) \leftarrow$
$and(false, false, false) \leftarrow$
The approximation language admits any equality as an atomic approximation constraint. In a program where *constraint and(X, Y,Z)* appears as a goal, the following information can be extracted:

Constraint store	Information extracted
$Empty$	$Nothing$
$X = false$	$Z = false$
$X = true$	$Z = Y$
$Y = false$	$Z = false$
$Y = true$	$Z = X$
$Z = true$	$X = true \wedge Y = true$
$X = Y$	$Z = X$

Even though boolean variables have finite (2-element) domains, finite domain propagation cannot elicit any information in case, for example, the constraint store has $X = true$. In this case both Y and Z could take either value $true$ or $false$. For real problems in the applications listed above, the extraction of information of the form $Z = Y$ is essential for performance reasons.

To obtain such a behaviour on these applications in CHIP [SD90, SD87b, SD87a, Sim88, SP89] it was necessary to use a form of guarded clause called "demons". The demon clauses defining the *and* predicate explicitly use the constraints in the "Constraint Store" column above as guards. Each demon remains idle, until the current constraint store logically implies its guard. At this point the clause is immediately selected and unfolded. However no choice point appears in the evaluation tree: the system commits to the selected demon clause and the other clauses are excluded. Expressed using an extended clause syntax, with a vertical bar to separate the guard from the clause body, the *and* demons are:
$and(X, Y, Z) \leftarrow X = false | Z = false$
$and(X, Y, Z) \leftarrow X = true | Z = Y$
...
Whilst the demons for *and* are built-in in CHIP, for complex boolean functions the CHIP programmer is required to generate a set of demons for himself. To encode a set of demons for a propagation constraint the programmer must consider all cases and generate each demon body by, effectively, performing the propagation in their head. Propagation constraints like *and* can often be encoded into demons. However, experiments have shown that the number of distinct demons required for even moderately complex boolean functions can often be over ten thousand.

The relationship between generalised propagation and committed choice languages will be discussed in more detail below.

4.2 GP(HU)

In the last section we examined applications run using Propia which did not use functors.

In fact all practical Propia programs use functors. We first consider some programs which use functors,

but whose propagation steps yield only bindings between variables and other variables or constants.

A propositional satisfiability problem is often expressed as a set of clauses,
$(X \vee Y \vee \neg Z) \wedge (\neg X \vee \neg Y) \wedge (X \vee Z) \wedge \ldots$.
The idea is to obtain an assignment of t or f to all the propositional variables so as to satisfy every clause. Such a problem can be expressed in logic programming as a query:
$\leftarrow pclause([+X, +Y, -Z]), pclause([-X, -Y]), pclause([+X, +Z]), ...$
with the following definition of $pclause$:
$pclause([+t|\_]) \leftarrow$
$pclause([-f|\_]) \leftarrow$
$pclause([\_|T]) \leftarrow pclause(T)$
There is no restriction on the size of a clause, so the list in the argument of $pclause$ may be arbitrarily long. However atoms of the form $pclause(List)$ can perfectly well be used as propagation constraints, and used for pruning the search for a solution to satisfiablity problems. For example propagation on the goal $pclause([+X])$ immediately produces the answer $X = t$, and similarly $pclause([-X])$ produces $X = f$. Thus generalised propagation immediately assigns values to variables appearing in singleton clauses, which is a technique used by specialised programs for solving propositional satisfiability problems.

Given a fixed set of clauses, with a fixed bound on the number of variables in a clause, it is possible to use CHIP's demons to perform similar propagation. Though in this case the "calculation of the best approximation" for each agent and constraint store has already been done by the programmer, it is interesting to record that Propia when applied to a benchmark of propositional satisfiability problems [MR91], had execution times on the same hardware similar to that obtained using CHIP's demons. This reflects the performance of the Sepia engine and the efficiency of the topological branch and bound algorithm.

We now consider what happens when functors appear in the approximation constraints. The information extracted remains information about equalities between terms. However the answers to a query may now contain local variables. For example the first answer to the query $\leftarrow member(1, Y)$ given the usual definition of $member$
$member(X, [X|T]) \leftarrow$
$member(X, [H|T]) \leftarrow member(X, T)$
is $\exists T.Y = [1|T]$. Theoretically we can eliminate such local variables in approximation constraints by admitting functions $functor(Atom)$, $arity(Atom)$ and $arg(Position, Atom)$ as approximation constraints, where $functor(f(X, Y)) = f$, $arity(f(X, Y)) = 2$ and $arg(1, f(X, Y)) = X$. The above answer could now be expressed as $functor(Y) = dot \wedge arity(Y) = 2 \wedge arg(1, Y) = 1$. However we use a shorthand which is to admit the _ symbol in answers. Thus we write $Y = [1|\_]$.

Using these approximation constraints we wish to show that infinite decreasing consistent sequence are finite. For a given depth of function embedding there are only finitely many equalities in a given fixed set of free variables which can be used to approximate a propagation agent. Subsequently, to achieve a tighter approximation it is necessary to use a deeper embedding of functions. An infinite sequence of approximations therefore will include terms of greater and greater depth. Since terms of infinite depth do not denote elements of HU, such infinite sequences cannot be consistent (see also section 3.4.1 above).

In many applications it is of interest to detect the success or failure of membership as soon as possible, instead of just using member as a check. Yet even this is a serious problem (see for example [Nai86]). For example even if the tail of the list is known most control regimes require the check to delay until the head of the list either equals or fails to unify with the first argument.

Generalised propagation can be applied to any $member$ propagation constraint without fear of non-termination. The information extracted from "constraint $member(M, [E1, \ldots, En|Tail])$" can be summarised as follows.

- If $Tail$ is empty, then
  - M becomes equal to the most specific generalisation of $M1, \ldots, Mn$ where Mi is the most general unifier of M and Ei. If none of the Ei unify with M, the result is $false$.
  - Ei becomes equal to the most general unifier of Ei and M if Ei is the only element that unifies with M. Otherwise there are no resulting constraints on Ei.

- If $Tail$ is a variable, then
  - There are no resulting constraints on M
  - There are no resulting constraints on any Ei
  - If none of the Ei unify with M, then $Tail = [\_|\_]$

The effect of the topological branch and bound in pruning the search for the infinite set of answers which return bindings for the tail is essential to ensure termination.

It is very instructive to try and construct ways of expressing the same propagation using guarded clauses!

Our experience shows that generalised propagation can safely be used for Horn clause programs with function symbols. Moreover we have been experimenting with generalised propagation in a database context, with favourable early results [BPM92]. The application to $GP(HU)$ means that generalised propagation can also be applied to database relations with compound attribute values.

5 Generalised Propagation and Other Approaches

There are many overlaps with other work and in this paper it is not possible to include a full comparison in every case. We have tried to consider more closely related research which is particularly interesting and influential. However even in the short list considered here, there are many points on which our comparison could be greatly expanded.

5.1 Most Specific Logic Programs

The instance $GP(HU)$ of generalised propagation extracts information from propagation constraints which is precisely the most specific generalisation described in [MNL88]. In this earlier work, the most specific generalisation of a set of possible solutions was calculated in advance of execution, so as to transform a program statically into one which was more efficient and had other better properties. Various algorithms have been proposed for calculating most specific logic programs, some using bottom-up evaluation and others breadth-first.

By contrast generalised propagation is performed at runtime. This presents new challenges since it must be efficiently implemented, and there may be different tradeoffs between the precision of the approximation language (i.e. the amount of information extracted) and the cost of propagation. The topological branch and bound procedure, based on a pruned top-down evaluation, can be efficiently implemented and makes practicable the extraction of most specific generalisations, or other approximations, at runtime.

The most specific generalisation of a program captures as much information as can be captured once, at compile time, and then nothing more is possible. Generalised propagation also offers more different possibilities for optimisation since the flow of information through the program may depend on data supplied at runtime. For example given the propagation agents $and(X, Y1, Z1), or(X, Y2, Z2)$, if X is instantiated to t, then propagation yields $Z2 = t$, however if X is instantiated to f then propagation yields information about $Z1$ instead! Finally generalised propagation offers the possibility to interleave propagation and search, and a propagation agent may be involved in many different propagation sequences during a single derivation.

5.2 Relaxed Tell

In [HD91] an operational semantics for constraint logic programming is introduced which offers an operation called *relaxed tell*. The relaxed tell operation extracts from a non-basic constraint an approximation. The operation requires two functions, a *relaxation* function and an approximation function which depends on the relaxation function.

A relaxation function r maps the constraint store S to an approximation $r(S)$ satisfying $\models S \rightarrow r(S)$. CHIP uses such a relaxation function in its treatment of arithmetic constraints over finite domains. A finite domain for a variable V, such as $\{1,2,4\}$ can be approximated by its end points, $1 \leq V \leq 4$.

An approximation function ap (given a relaxation function r) maps a non-basic constraint C and a store S to an approximation $ap(S,C)$ satisfying $(r(S) \wedge C) \rightarrow ap(S,C)$. CHIP also uses approximation functions in its treatment of arithmetic constraints over finite domains. For example the linear constraint $1+V1 = V2$ is handled by using the equations to reduce the upper bounds and increase the lower bounds of the variable domains so that the equation is satisfied by the new bounds. Thus if $V1 \in \{1,3\}$ and $V2 \in \{2,3\}$, the result of approximation on the above equation is $1 \leq V1 \leq 2$ and $2 \leq V2 \leq 3$.

The requirement on the approximation function in the relaxed tell framework is that it must approximate the constraint C, whereas in the framework of generalised propagation the result approximates all the answers to the constraint. This difference arises because relaxed tell is designed for non-basic *built-in* constraints such as arithmetic ones. For generalised propagation *any user goal* can be annotated as a constraint. In this case there is a clear definition of an answer to the constraint, but the logical semantics of the constraint itself is more difficult to pin down. The logical semantics for program clauses does not license any negative consequences. However in this case no pruning information could be extracted from propagation constraints! For our purposes it would therefore be necessary to use some form of minimal model semantics for constraint logic programs, with all the restrictions this entails [JL87].

Apart from the restriction to built-in constraints, relaxed tell is an abstraction of generalised propagation. The inclusion of a relaxation function makes it strictly more powerful than generalised propagation, whose "relaxation function" is just the identity function. The disadvantage of using a relaxation function is that propagation on a single constraint cannot be guaranteed to yield a fixpoint. In fact the example of approximation above has this property. If the result of propagation is added to the constraint store the resulting store now has a different relaxation $1 \leq V1 \leq 1$, which enables further useful propagation to be performed *on the same constraint*. This means that the efficient AC-3 algorithm no longer produces complete propagation sequences.[1]

5.3 Guarded Clauses and Concurrent Constraint Logic Programming

It is not possible in this paper to make a comparison of generalised propagation with the different languages in these frameworks. At an abstract level propagation constraints can be seen as deterministic processing agents which communicate with the constraint store using *relaxed tell*. More concretely it is interesting to specify precisely what communications take place in terms of *ask* and *tell*, and how this behaviour reflects the declarative semantics of the constraint.

We can therefore attempt to encode the behaviour of a propagation constraint as a set of definitions using committed choice, guarded clauses. Let us take finite domain propagation as an example and use $ask\ X \in \{C_1,\ldots,C_n\}$ to ask if the current constraint store implies that $(X = C_1) \vee \ldots \vee (X = C_n)$, and $tell\ X \in \{C_1,\ldots,C_n\}$ to tell this formula to the constraint store. For *constraint* $p(X,Y)$, where p is defined as
$p(1,2) \leftarrow$
$p(2,1) \leftarrow$
$p(3,1) \leftarrow$
we could express finite domain propagation thus:

$$
\begin{array}{lll}
constraint\ p(X,Y) & \leftarrow true & \mid tell\ X \in \{1,2,3\}, tell\ Y \in \{1,2\}, constraint\ p1(X,Y) \\
constraint\ p1(X,Y) & \leftarrow ask\ X \in \{2,3\} & \mid tell\ Y = 1 \\
constraint\ p1(X,Y) & \leftarrow ask\ X = 1 & \mid tell\ Y = 2 \\
constraint\ p1(X,Y) & \leftarrow ask\ X = 2 & \mid tell\ Y = 1 \\
constraint\ p1(X,Y) & \leftarrow ask\ X = 3 & \mid tell\ Y = 1 \\
constraint\ p1(X,Y) & \leftarrow ask\ Y = 1 & \mid tell\ X \in \{2,3\} \\
constraint\ p1(X,Y) & \leftarrow ask\ Y = 2 & \mid tell\ X = 1
\end{array}
$$

This encoding is similar to that used for the *and* demons (see section 4.1.2 above).

[1] In CHIP, which uses AC-3, it is therefore sometimes necessary to state constraints twice!

The main drawback of using such an encoding is the huge number of clauses necessary to capture each interesting propagation. We hypothesise that if conjunctions of basic constraints are admitted in the guard, the number of guarded clauses can rise exponentially with the number of clauses needed to express the propagation constraint.

A second drawback of guarded clauses is, paradoxically, their great expressive power. For example it is possible to express the *merge* operation using guarded clauses, although this operation has no logical semantics. In general it is not possible to give a declarative semantics for a set of guarded clauses, and thus it is not possible to state the effect of a program except in terms of the operational behaviour of its clauses.

There is a "logical subset" of guarded clause programs that have a logical semantics. It is possible to state when a set of "logical" guarded clauses is sound with respect to a logic program specification as in [Smo91]. However even for such logically sound guarded clauses there remains the question of completeness. There seems no simple way to determine when the behaviour of a set of clauses is equivalent to the behaviour of generalised propagation. For example it is only possible to confirm that the encoding of *constraint* $p(X, Y)$ using guarded clauses above really does extract all possible propagations in all possible constraint stores by performing an exhaustive analysis on constraint stores. The set of interesting constraint stores to be analysed soon grows prohibitively large for non-trivial constraints (see also above section 4.1.2).

A form of guarded rules with multiple heads is being investigated at ECRC [Fru92], which provides a language for expressing constraint simplification. The rules are called *simplification rules*. In many cases it would be practical to express certain interesting propagations as simplification rules. The integration of these *simplification rules* into our framework would make it possible to encode the results of static analysis and partial evaluation of generalised propagation. Consequently the whole range of possibilities on the continuum between compilation and interpretation of generalised propagation would be available in one system.

5.4 Andorra

A relationship has been often pointed out between David Warren's Andorra principle [War88] and the preference for deterministic computation which underlies constraint propagation. Based on Warren's extended Andorra model [War90], the language AKL has been defined [HJ90]. In this section we compare generalised propagation with AKL.

Andorra promotes deterministic computations. The control of how hard to work to find subcomputations that yield deterministic results has reached a considerable degree of sophistication. However the basic idea is to perform parts of the computation locally and if the result is deterministic to make it available globally, adding the resulting constraints to the constraint store. This is similar to extracting results from propagation constraints.

In a local computation in Andorra, nothing is thrown away. This is quite different from constraint propagation which finds many answers, extracts "common" information from them all, and then throws the answers away again. This can in practice make constraint propagation more expensive than Andorra's deterministic promotion, but it also makes it possible to extract more information deterministically than can be done in Andorra. For example generalised propagation extracts $X = f(\_)$ from the propagation agent $p(X)$ defined by
$p(f(a)) \leftarrow$
$p(f(b)) \leftarrow$
However the evaluation of $p(X)$ is not deterministic so no information can be extracted in Andorra.

A second difference has to do with the dependence of information extracted on the precise syntax of the program. In Andorra the information that can be extracted from a local computation depends on the precise clausal definitions of the goal predicates involved. For example we could recode $p(X)$ above as
$p(f(Y)) \leftarrow q(Y)$
$q(a) \leftarrow$
$q(b) \leftarrow$
to get more information extracted by Andorra from the goal $p(X)$. In constraint propagation the information extracted is independent of the program syntax. It depends only on the logical semantics of the

program. Therefore constraint propagation has a more abstract behavioural semantics than deterministic promotion in Andorra.

6 Conclusion

The same word "constraint" has been used to describe two rather different extensions of logic programming. In one extension ($CLP(X)$) "constraints" involve interpreted predicates whose interpretation on the underlying domain is predefined. In the other extension (based on CSP) "constraints" are goals which are used not for search but for deterministic reduction of the search space. This paper has extracted a more abstract concept which includes both uses of the word *constraint*.

The abstract concept is useful for clarifying our understanding of CLP, but this paper has shown that it also yields immediate practical benefits. A generalisation of propagation has been introduced which integrates the constraint behaviour of both extensions. This enables techniques of local consistency enforcement from CSP to be applied to arbitrary goals in arbitrary $CLP(X)$ programs. The result is called $GP(X)$, for "generalised propagation parameterised on the computation domain X".

Propagation on a goal G in $GP(X)$ requires that the system extracts a constraint approximating all the answers to G. The paper has introduced a generic algorithm for generalised propagation which avoids enumerating all the answers to a propagation constraint. Instead the retrieval of answers is interleaved with approximation steps, so that an approximation to the answers found so far is always maintained. This approximation is used to cut branches in the search for answer, in a way similar to branch and bound. Additionally it is used to cut all the remaining branches in the search tree, when the approximation becomes too general to be useful. The algorithm has been called topological branch and bound, in section 3.7 above.

Generalised propagation offers very flexible control via the choice of approximation constraints. If only a coarse approximation is offered the topological branch and bound drastically prunes the search tree, thus making generalised propagation relatively cheap. If a finer approximation is offered, more information is extracted from each propagation constraint, enabling the global search to be more reduced.

An implementation (called *Propia*) of generalised propagation over the Herbrand universe has been described. Experiments with Propia have shown that generalised propagation enables problems to be simply stated and efficiently solved in a way not possible using either $CLP(X)$ or propagation based on CSP. It has been very rewarding to take pure logic programs as specifications and, by simply annotating certain goals as propagation constraints, to achieve an efficient implementation. A very important feature of the resulting programs is their guaranteed correctness with respect to their specification. This can be contrasted with the encoding of the same problems using demons (a special form of guarded clause), which cannot be validated against the specification since they have no declarative semantics.

As to the future, further implementations of generalised propagation are being developed for new computation domains, thus expanding the range of problems that can be naturally expressed as $GP(X)$ programs. We are also investigating the notion of propagation constraints as concurrent processing agents. In this view generalised propagation is an interesting special case of concurrent constraint logic programming, in which the operational semantics can be dramatically simplified (and for which there is always an equivalent declarative semantics). Finally, partial evaluation of $GP(HU)$ is already under investigation at ECRC, with the results expressed in the form of demons. With the integration of simplification rules into our system (see section 5.3 above), the potential for optimisation of $GP(X)$ programs can be fully explored.

7 Acknowledgements

This paper has benefited from discussions with many researchers both inside and outside of ECRC. We particularly wish to thank André Veron, for his extensive implementation work and experimentation on generalised propagation in the Elipsys system. We also thank the reviewers for perceptive and helpful comments. Andrei Voronkov gave important feedback on the final draft. Our collaborators in the CHIC

Esprit project (Nr. 5291) have also helped sharpen our ideas. Finally thanks to all the CORE team at ECRC and to Alexander Herold, for reading many drafts of papers on generalised propagation and helping to bring out the important issues.

Bibliography

[Ber87] H. Berghel. Crossword compilation with Horn clauses. *The Computer Journal*, 30(2):183–188, 1987.

[BPM92] S. Bressan, T. Le Provost, and O. Monteil. Experiments with set-oriented propagation. Experiments performed at ECRC: report in preparation, 1992.

[Cla79] K.L. Clark. Predicate logic as a computational formalism. Technical Report 79/59, Imperial College, London, 1979.

[Col85] A. Colmerauer. *Theoretical Model of Prolog II*, pages 3–31. Ablex Publishing Corporation, 1985.

[Dav87] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[DSV90] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:74–94, 1990.

[DSVX91] M. Dorochevsky, K. Schuermann, A. Véron, and J. Xu. Constraints Handling, Garbage Collection and Execution Model Issues in ElipSys. In A. Beaumont and G. Gupta, editors, *Proceedings of the ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, Paris, June 1991. LNCS 569.

[DVS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, November 1988.

[Fik70] R.E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[Fru92] T. Fruehwirth. Constraint simplification rules. Technical Report ECRC-92-18, ECRC, July 1992.

[Gal85] H. Gallaire. Logic programming: further developments. In *IEEE Symposium on Logic Programming*, pages 88–99, Boston, July 1985. Invited paper.

[GB65] S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.

[HD91] P. Van Hentenryck and Y. Deville. Operational semantics of constraint logic programming over finite domains. In *Proc. PLILP'91*, Passau, Germany, Aug 1991.

[HE80] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, October 1980.

[HJ90] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In *Proc. of the 7^{th} Int. Conf. on Logic Programming* [ICL90], pages 31–46.

[ICL87] *Proceedings of the 4^{th} International Conference on Logic Programming*, Melbourne, 1987. MIT Press.

[ICL88] *Proceedings of the 5^{th} International Conference and Symposium on Logic Programming*, Seattle, 1988. MIT Press.

[ICL90] *Proceedings of the 7^{th} International Conference on Logic Programming*, Jerusalem, Israel, 1990. MIT Press.

[JL86] J. Jaffar and J-L. Lassez. Constraint logic programming. Draft Technical Report, Monash University, June 1986.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages (POPL'87)*, Munich, FRG, January 1987.

[Kow79] R. A. Kowalski. *Logic for Problem Solving*, chapter 8. North-Holland, 1979.

[Lev91] J. Lever. Temporal reasoning - a progress report. Presented at the CHIC workshop, Imperial College, 1991.

[Llo84] J.W. Lloyd. *Foundations Of Logic Programming*. Springer-Verlag, 1984.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[MAC+89] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. De Villeneuve, A. Herold, P. Kay, B. Perez, E.Van Rossum, and J. Schimpf. Sepia - an extendible prolog system. In G. X. Ritter, editor, *Information Processing 89*, San Francisco, September 1989. Elsevier Science Publisher B.V.

[Mah87] M. J. Maher. Logic semantics for a class of committed-choice programs. In *Proc. of the* 4^{th} *Int. Conf. on Logic Programming* [ICL87], pages 858–876.

[MF85] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[MNL88] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proc. of the* 5^{th} *Int. Conf. and Symp. on Logic Programming* [ICL88], pages 909–923.

[Mon74] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[MR91] I. Mitterreiter and F. J. Radermacher. Experiments on the running time behaviour of some algorithms solving propositional calculus problems. Technical Report Draft, FAW, Ulm, 1991.

[NAC90] *Proceedings of the 1990 North American Conference on Logic Programming*. MIT Press, 1990.

[Nai86] L. Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer, 1986. PhD. Thesis, Melbourne Univ.

[PV92] T. Le Provost and A. Veron. Boosting an application via constraints prototyping and or-parallelism. Forthcoming ECRC report, 1992.

[RHZ75] A. Rosenfeld, A. Hummel, and S.W. Zucker. Scene labelling by relaxation operations. Technical Report TR-379, Computer Science Department, University of Maryland, 1975.

[Sar89] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa, January 1989.

[SD87a] H. Simonis and M. Dincbas. Using an extended prolog for digital circuit design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, Munich, W.Germany, October 1987.

[SD87b] H. Simonis and M. Dincbas. Using logic programming for fault diagnosis in digital circuits. In *German Workshop on Artificial Intelligence (GWAI-87)*, pages 139–148, Geseke, W. Germany, September 1987.

[SD90] H. Simonis and M. Dincbas. Propositional calculus problems in CHIP. In H. Kirchner, editor, *Proceedings of the 2nd International Conf on Algebraic and Logic Programming*, Nancy, France, October 1990. CRIN and INRIA-Lorraine, Springer Verlag. (to appear).

[Sim88] H. Simonis. Test pattern generation with logic programming. In *New Aspects of Research for Testing of VLSI Circuits*, Ising, W. Germany, March 1988.

[Smo91] G. Smolka. Residuation and guarded rules for constraint logic programming. Technical Report 12, Digital PRL, June 1991.

[SP89] H. Simonis and T. Le Provost. Circuit verification in CHIP: Benchmark results. In L.J.M. Claesen, editor, *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, November 1989. IFIP, North Holland, Elsevier Science Publishers.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. The MIT Press, 1989.

[VD86] P. Van Hentenryck and M. Dincbas. Domains in logic programming. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86)*, Philadelphia, PA, August 1986.

[War88] D.H.D. Warren. The Andorra Model. Presented at the Gigalips Workshop, Univ. of Manchester, 1988.

[War90] D.H.D. Warren. The Extended Andorra Model with implicit control. Presented at the ICLP'90 workshop on Parallel Logic Programming, Isreal, June 1990.

Other Reports Available from ECRC

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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