Domain Independent Propagation

Thierry Le Provost and Mark Wallace

Abstract

Recent years have seen the emergence of two main approaches to integrating constraints into logic programming. The *CLP Scheme* introduces constraints as basic statements over built-in computation domains. On the other hand, systems such as CHIP have introduced new inference rules, which enable certain predicates to be used for propagation thereby pruning the search tree. Unfortunately these two complementary approaches were up to now incompatible, since propagation techniques appeared intimately tied to the notion of finite domains. This paper introduces a generalisation of propagation that is applicable to any CLP computation domain, thereby restoring orthogonality and bridging the gap between two important constraint logic programming paradigms. The practical interest of this new notion of "domain independent" propagation is demonstrated by applying a prototype system for solving some hard search problems.

1 Introduction

There are two main approaches for integrating constraints into logic programming. The first approach, formalised as $CLP(X)$ [JL87], is to replace the usual domain of computation with a new domain X. The computation domain X specifies a universe of values; a set of predefined functions and relations on this universe; and a class of *basic constraints*, which are formulae built from predefined predicate and function symbols, and logical connectives. The CLP scheme requires that an effective procedure decide on the satisfiability of the basic constraints. The facility to define new predicates as facts or rules, possibly involving the built-in's, is carried over from logic programming. The evaluation of queries involving such user-defined predicates is performed using an extension of resolution, where syntactic unification is replaced with deciding the satisfiability of basic constraints (constraint solving). As with standard logic programming the default search method for evaluating program-defined predicates is depth-first, based on the ordering of program clauses and goals.

The second main approach to integrating constraints in logic programming uses the standard, syntactic, domain of computation, except that the variables may be restricted to explicitly range over finite subsets of the universe of values (*finite domain variables*) [VD86]. In this approach, inaugurated by *CHIP* [DVS$^+$88], it is the proof system that is extended. The new type of controlled inference is termed *constraint propagation* or consistency techniques [Van89]. These techniques combine solution-preserving simplification rules and tree search, and were originally introduced for solving constraint satisfaction problems [Mon74, Mac77].

Informally constraint propagation aims at exploiting program-defined predicates as constraints. It operates by looking ahead at yet unsolved goals to see what locally consistent valuations there remain for individual problem variables. Such constraint techniques can have a dramatic effect in cutting down the size of the search space [DSV90].

To date the technique of propagation has only been defined for search involving finite domain variables. 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 $CLP(X)$ approach. In addition propagation, as currently defined, 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 have had to remain quite separate. Even in the CHIP system 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.

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. Furthermore its basic concepts, theoretical foundations, and abstract operational semantics can be defined independently of the computation domain. This allows programmers to reason about the efficiency of CLP programs involving propagation 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 programs is preserved.

The main idea behind generalised propagation is to use whatever basic constraints are available in a $CLP(X)$ language to express restrictions on problem variables. Goals designated as propagation constraints are repeatedly approximated to the finest basic constraint preserving their solutions. When no further refinement of the current resolvent's basic constraint is feasible, a resolution step is performed and propagation starts again.

The practical relevance of generalised propagation has been tested by implementing it in the computation domain of Prolog. 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. The performance results have been very encouraging.

In the next section we recall the interest of integrating propagation over finite domains into logic programming. We then present a logical basis for propagation that will provide the basis for generalisation. The following section introduces generalised propagation, and sketches its theoretical basis. The fourth section introduces our prototype system on top of Prolog, and discusses some of the examples that we tackled with it. In conclusion we identify the directions that this work is now taking.

2 Propagation over Finite Domains

2.1 Propagation in Constraint Satisfaction Problems

The study of constraint satisfaction problems has a long history, and we mention here just a few important references. The concept of arc consistency was introduced in [Mac77]; its combination with backtrack search was described in [HE80]; the notion of value propagation is due to [SS80]; the application of constraint methods to real arithmetic was surveyed in [Dav87]; finally [Van89] extensively motivates and describes in detail the integration of finite-domain propagation methods into logic programming.

A constraint satisfaction problem (CSP) can be represented as

- a set of variables, $\{X1, \ldots, Xn\}$, each Xi ranging over a finite domain Di;
- a set of constraints $C1, \ldots, Cm$ on these variables, where each constraint Ci is an atomic goal $p_i(Xi_1, ...Xi_k)$ defined by a k-ary predicate p_i.

A solution to the problem is an assignment of values from the domains to the variables (a *labelling*) such that all the constraints are satisfied. We now briefly recall the main approaches to solving CSP's in a logic programming setting, using the following toy example. The problem has four variables $X1, X2, X3, X4$, each with domain $\{a, b, c\}$. There are four constraints, each involving the same binary predicate p:
$p(X3, X1) \wedge p(X2, X3) \wedge p(X2, X4) \wedge p(X3, X4)$
The relation denoted by p has three tuples: $< a, b >$, $< a, c >$, $< b, c >$.

Generate and Test This approach enumerates labellings in a systematic way until one is found that satisfies all the constraints. It is hopelessly inefficient for all but the smallest problem instances. In our example the system will go through all 27 labellings which begin with an a, before discovering that $X1$ cannot take this value due to the first constraint $p(X3, X1)$. In general reordering the constraint goals may only bring minor improvements. Analysing the cause for the failure of goals so as to avoid irrelevant backtrack steps (*selective backtracking*) makes the runtime structures more complex and is insufficient for complex problems (see for instance [Wol89]).[1]

Backtrack Search A first improvement on pure generate-and-test is to check each constraint goal as soon as all its variables have received values [GB65]. Backtrack search thus performs an implicit enumeration over the space of possible labellings, discarding partial labellings as soon as they can be proved *locally inconsistent* with respect to some constraint goal. Backtrack search demonstrates considerable gains over generate-and-test (the inconsistent assignment $X1 = a$ is detected at once). However this procedure still suffers from "maladies" [Mac77], the worst being its repeated discovery of local inconsistencies. For instance it is obvious from $p(X3, X1) \wedge p(X2, X3)$ alone that $X1$ cannot take the value b. Backtrack search will nonetheless consider all 9 combinations of values for $X2$ and $X3$ before rescinding $X1 = b$.

Local Propagation The idea behind local propagation methods for CSP's is to work on each 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 [Mon74]. Consistency can be applied as a preliminary to the search steps or interleaved with them [HE80]. The application of these techniques in the constraint logic programming language CHIP 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 *lookahead inference rules* to reduce finite domains in a deterministic way

The effect of applying lookahead 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 goal solutions. The solutions can be determined by simply calling the goal repeatedly. Application of the lookahead rule is repeated on all constraint goals until no more domain reductions are possible, forming a *propagation sequence*. Constraint goals that are satisfied by any combination of values in the domains of their arguments can now be dropped.

Our example problem can be encoded in a CHIP-like syntax as follows:

```
csp(X1,X2,X3,X4) :-
    lookahead p(X3,X1),  /* [1] */
    lookahead p(X2,X3),  /* [2] */
    lookahead p(X2,X4),  /* [3] */
```

[1]Selective, or intelligent, backtracking [CS90] addresses the symptom of too many choice points. Propagation addresses the cause, by reducing the number of choice points in advance.

```
    lookahead p(X3,X4),  /* [4] */
    dom(X1),dom(X2),dom(X3),dom(X4).
```

The `lookahead` 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; domain goals merely check each of the variable bindings already produced. A possible computation sequence is as follows (though the ordering is immaterial for the final result):

```
lookahead on:          produces:
p(X3,X1)       [1]  X3::{a,b}, X1::{b,c}
p(X2,X3::{a,b}) [2]     X2=a, X3=b
p(a,X4)         [3]     X4::{b,c}
p(b,X1::{b,c})  [4]     X4=c
p(b,X1::{b,c})  [1]     X1=c
p(a,b)          [2]  succeeds
p(a,c)          [3]  succeeds
p(b,c)          [4]  succeeds
```

Note that the 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.

From this brief summary of consistency techniques for CSP's and their integration into logic programming, it may appear that finite domain variables form the cornerstone of propagation. The purpose of this paper is to show that this is not the case, and that propagation has a very general, natural and useful counterpart in constraint logic programming languages that do not feature finite domains.

2.2 A Logical Basis for Propagation

The effect of (finite domain) propagation on a constraint is to reduce the domains associated with the variables appearing in the constraint. The resulting domains capture as precisely as possible the meaning of the constraint. The aim of this section is to say in what sense the meaning of a constraint is captured by a set of domains, and to give a formal characterisation of the qualification "as precisely as possible".

A constraint $C(X_1, \ldots, X_n)$ is to be understood as a logical formula with free variables $X_1, \ldots, X_n$. A *constraint formula* has the syntactic form:
$(X_1 = a_{11} \wedge \ldots \wedge X_n = a_{1n}) \vee \ldots \vee (X_1 = a_{k1} \wedge \ldots \wedge X_n = a_{kn})$.
A *domain formula* $Dom(X)$ is a disjunction of equalities involving a single variable X:
$X = a_1 \vee X = a_2 \vee \ldots \vee X = a_n$.
Generally many variables are involved in a problem, and we therefore introduce a syntactic class of formulae representing the conjunction of their domains. These are the *basic formulae.* Thus a basic formula $D(X_1, \ldots, X_n)$ has the form:
$Dom_1(X_1) \wedge \ldots \wedge Dom_n(X_n)$.

The reduced domains, resulting from propagation on a constraint, approximate the constraint formula as closely as is possible using only a basic formula. Propagation is "precise" if this basic formula is logically equivalent to the constraint formula. The problem is that basic formulae have a limited expressive power, and it is not in general possible to find one logically equivalent to a given constraint formula.

For example the constraint formula $C(X_1, X_2)$,
$(X_1 = a \wedge X_2 = b) \vee (X_1 = a \wedge X_2 = c) \vee (X_1 = b \wedge X_2 = c)$,
is best approximated by the basic formula
$(X_1 = a \vee X_1 = b) \wedge (X_2 = b \vee X_2 = c)$.
However there is no basic formula logically equivalent to $C(X_1, X_2)$.

Definition 1 *A propagation step takes a constraint formula C and a basic formula D and yields a "least" basic formula D' which satisfies $(C \wedge D) \rightarrow D'$. D' is the least such formula in the sense that for any other basic formula D'' satisfying $(C \wedge D) \rightarrow D''$ it is also true that $D' \rightarrow D''$.*

This definition will be illustrated using the constraint $C(X_1, X_2)$:
$(X_1 = a \wedge X_2 = b) \vee (X_1 = b \wedge X_2 = c) \vee (X_1 = c \wedge X_2 = a)$.
The input basic formula $D(X_1, X_2)$ is:
$(X_1 = a \vee X_1 = b) \wedge (X_2 = a \vee X_2 = b \vee X_2 = c)$.

Propagation on a constraint involves two steps: the *simplification* of the constraint and the *reduction* of domains associated with its variables.

The simplification of the constraint $C(X_1, X_2)$, with respect to the basic constraint $D(X_1, X_2)$ is just the calculation of a simplified constraint logically equivalent to $C(X_1, X_2) \wedge D(X_1, X_2)$. The result of simplifying is $C'(X_1, X_2) \equiv (C(X_1, X_2) \wedge D(X_1, X_2)) \equiv$
$(X_1 = a \wedge X_2 = b) \vee (X_1 = b \wedge X_2 = c)$.

The reduction of the domains is the calculation of a new basic formula which approximates as closely as possible the simplified constraint. The result of reducing is $D'(X_1, X_2) \equiv$
$(X_1 = a \vee X_1 = b) \wedge (X_2 = b \vee X_2 = c)$.
For this example there is no basic constraint logically equivalent to $C'(X_1, X_2)$. However $D'(X_1, X_2)$ is the least basic formula implied by $C'(X_1, X_2)$ since the domain of X_1 must include at least a and b, and the domain of X_2 must include at least b and c.

Definition 2 *Propagation is the result of applying a propagation sequence, which is the repeated application of propagation steps on every constraint until no more domain reductions are possible.*

This definition does not mention the order in which propagation steps are done. In fact the result of performing propagation on a set of constraints is independent of the order. We prove this as follows.

Lemma 1 *If basic formulae are ordered by logical entailment, propagation steps are increasing and monotonic on basic formulae.*

This is easily deduced from the definition of a propagation step.

Lemma 2 *Each (ordered) propagation sequence yields a fixpoint.*

This follows from the fact that there are only finitely many basic formulae greater than a given basic formula under the logical entailment ordering, and propagation steps are increasing.

Theorem 1 *The result of a propagation sequence is independent of the order of the steps.*

Suppose $fix1$ and $fix2$ were distinct fixpoints of a propagation sequence, resulting from an initial basic formula $s0$. Since propagation is increasing, $fix1 > s0$. $fix2$ results from applying a particular ordered sequence of propagations on $s0$. By monotonicity this same sequence applied to $fix1$ yields a result $fix3 > fix2$. However since $fix1$ is a fixpoint of the propagation sequence, $fix3 = fix1$. We conclude that $fix1 > fix2$. Symmetrically we can conclude that $fix2 > fix1$, and therefore $fix1 = fix2$.

It is also possible to show that propagation can be performed in parallel, and still yield the same fixpoint. These and other results fall out very naturally when lattice theory is used to describe the constraints. The lattice theoretic formalisation of generalised propagation is described in another paper [LW92].

3 Generalised Propagation

For finite domain propagation, the basic formulae express domains associated with the problem variables, and the constraint formulae express membership of tuples in relations. Each class of formulae has a certain limited expressive power. However the definition of a propagation step and a propagation sequence do not depend on the particular syntactic classes chosen for basic formulae or constraint formulae. In this section we will explore the consequences of admitting different classes of formulae. We shall propose a notion of *generalised propagation* parameterised on the classes of formulae.

In the CLP(X) approach a class of basic constraints is identified for each domain X. Generalised propagation on a domain X is the result of admitting the *basic constraints* on X as *basic formulae* as described in the last section. The class of constraint formulae is the class of goals expressible in CLP(X).

The basic formulae used for finite domain propagation involve only the equality predicate and no function symbols. For generalised propagation over a domain X the basic formulae may include other predicates, such as $<$ and $>$, and function symbols such as $+$ and $*$. However the purpose and effects of propagation remain the same. To detect inconsistencies early and to extract as much information as possible from a set of goals deterministically before making any choices. The information extracted is expressed as a basic formula, which is added to the current constraint set, either yielding inconsistency immediately, or else helping to prune the remaining search.

As a simple example of generalised propagation, consider $CLP(\mathcal{Q})$ with atomic constraints $Var \geq num$ and $Var \leq num$, where num is any rational number. Let us define a predicate p on which we shall perform generalised propagation.

```
p(X) :- X >= 3.4, X =< 4.6
p(X) :- X >= 2.8, X =< 3.9
```

Assume the current constraints include $X \leq 4.0$, and $p(X)$ is a goal. The CLP(X) approach requires us to treat user-defined predicates such as p a la Prolog. One clause in the definition of p is selected, and if that yields an inconsistency the other is tried on backtracking.

Generalised propagation on the predicate p, treated this time as a constraint, deterministically derives the tightest basic constraint $C(X)$ satisfying $(p(X) \wedge X \leq 4.0) \rightarrow C(X)$, and adds $C(X)$ to its current set of constraints. In this case $C(X) \equiv (X \geq 2.8 \wedge X \leq 4.0)$, which can be used to prune the remaining search tree.

The case of finite domains can be viewed as an instance, CLP(FD), of the constraint logic programming scheme, where the basic constraints are the basic formulae as defined in section 2.2. Propagation on finite domains can now be seen as an instance of generalised propagation, just as CLP(FD) is an instance of CLP(X). Notice that the expressive power of CLP(FD) is weaker than that of standard logic programming, since it is impossible using domains to state that two variables are equal, until their domains are reduced to one value. This is indeed a weakness of propagation over finite domains, and in the next section we shall present an implementation of generalised propagation that overcomes it.

Unfortunately it is not the case that generalised propagation can be automatically derived for any computation domain X. There is a practical requirement to constructively define a propagation step. Specifically, for propagating on a goal the system requires an efficient way to extract a basic formula which generalises all the answers to the goal.

More fundamentally a theoretical problem arises when we move from finite domain constraints to arbitrary basic constraints. There are only finitely many finite domain constraints tighter than a given constraint. This fact ensures that propagation is bound to reach a fixpoint. However for many sets of basic constraints, such as inequalities over the rationals as exampled above, there is no similar guarantee of termination. This problem has been addressed by introducing a notion of *approximate generalised propagation* in [LW92].

4 Propia: An Implementation of Generalised Propagation

4.1 An Overview of the Implementation

The behaviour of generalised propagation in practise has proved to be more than satisfactory. An implementation of generalised propagation has been completed based on ECRC's Sepia prolog system. We call it *Propia*. The underlying domain is the Herbrand domain of standard logic programming. The built-in relation on this domain is '=', and basic constraints are conjunctions of equalities (or equivalently substitutions).

A simple example of generalised propagation over this domain, is propagation on the predicate p defined as follows:

```
p(g(1),a,b).
p(f(a),a,a).
p(g(2),b,a).
p(f(b),b,b).
```

The result of generalised propagation on the goal `p(A,X,X)`, is the deterministic addition of a new equation, $A = f(X)$. Although there are two different possible values for A, they both have the form $f(X)$, where X is the *same* variable occurring as the second and third arguments in the goal. Using finite domains (even if structured terms were admitted) it would only be possible to infer that the domain of X was $\{a, b\}$ and the domain of A was $\{f(a), f(b)\}$, but not that $A = f(X)$. This is the weakness of finite domain pointed out on page 8 above.

Implementationally constraint simplification with respect to this goal amounts to selecting those clauses in the definition which unify with the goal, as done by Prolog. The reduction step, given a set of answers, finds the set of equations which best approximates them. The best approximation is, in fact, their most specific generalisation.

Computations interleave the making of choices and propagation. When a propagation sequence terminates, goals are called a la Prolog until a new binding, or set of bindings, occur thereby conjoining new equations $X = T$ to the current basic constraint. At this point propagation restarts. When a fixpoint is reached, the propagation sequence is complete and further goals are called a la Prolog.

It would be prohibitively expensive to attempt propagation on all the constraints at each choice. In practise the system determines on which variables new equalities have been added and only propagates on constraints involving those variables. When further equalities are added during a propagation sequence, then propagation is also attempted on constraints involving these variables.

The purpose of propagation is to extract as much information as possible deterministically before making any choices. The *Andorra principle* [War88] has a similar intent: it states that deterministic goals should be executed before other goals. The goal $p(A, X, X)$ in the previous example is clearly not deterministic, yet deterministic information can be extracted from it. Lee Naish coined the term *data determinacy* for the determinism detected and used by generalised propagation, as opposed to Andorra's weaker *control determinacy*.

4.2 An Example of Propagation

The behaviour of generalised propagation in the syntactic equality theory can be illustrated using a simple example. We shall investigate what propagation is possible for various calls on the 'and' predicate defined as follows:

```
and(true,true,true).
and(true,false,false).
and(false,true,false).
and(false,false,false).
```

We treat the goal as a propagation constraint by making the call `?- propagate and(_,_,_)`. Note that finite domain variables are not part of our chosen propagation language.

For "most specific generalisation" we shall use the abbreviation *msg*. First if the call is fully uninstantiated `?- propagate and(X,Y,Z)` the system finds the first two answers and forms the msg $and(true, Y, Z)$. After the third answer the msg becomes $and(X, Y, Z)$, which is as little instantiated as the query, and propagation stops.

Second if the call has its first argument instantiated to false `?- propagate and(false,Y,Z)` there are two answers whose msg is $and(false, \_, false)$. Thus the equality $Z = false$ is returned.

Third if the call has its first argument instantiated to true `?- propagate and(true,Y,Z)` there are again two answers, $and(true, false, false)$ and $and(true, true, true)$. Our generalisation procedure is able to derive the equality of the last two arguments and the final msg is $and(true, Y, Y)$. Thus the equality $Y = Z$ is returned.

We note that the behaviour is very similar to that obtained by encoding and using "cut guards" in Andorra, GHC rules, or "demons" in CHIP. For example in CHIP we would write:

```
?- demon and/3.
and(false,Y,Z) :- Z=false.
and(true,Y,Z)  :- Z=Y.
and(X,false,Z) :- Z=false.
and(X,true,Z)  :- Z=X.
and(X,Y,true)  :- X=true, Y = true.
and(X,X,Z)     :- Z=X
```

The difference is that the use of `propagate` enables us to separate the specification of the predicate, from its control. When using guards or demons we are forced to mix them together. Indeed generalised propagation allows declarative specifications to be directly used as constraints!

We used Propia for a benchmark set of propositional satisfiability problems distributed by the *FAW* research institute [MR91]. Its behaviour was in general quite comparable to that of CHIP's demons or built-in constraints.

Another application we examined was that of crossword puzzle compilation. The problem is to fill up an empty crossword grid using words from a given (possibly large) lexicon. The propagation constraints enforce membership of words in the given lexicon. Intersections are expressed through shared variables.

The statement of the problem is as follows:

```
/* some lexicon of available words */
word(a).
word(a,b,a,c,k).
...

prog :-
propagate word(A,B,C,D),
/* Note the shared letter B  */
propagate word(E,F,B,H),
...,
```

The program just comprises a set of propagation constraints. (There is no need for a labelling since Propia itself selects a propagation constraint for resolution when the propagation sequence terminates.) Immediately certain letters are instantiated by the original propagation. Subsequently, each time some letters are instantiated after selecting a *word* goal for resolution, the affected propagation constraints are re-executed in the hope of instantiating further letters.

The crossword compilation problem has also been addressed using CLP by Van Hentenryck [Van89]. Generalised propagation yields a performance improvement of about 15 times on Van Hentenryck's example. However much more significant is the power of generalised propagation for solving large problems. Van Hentenryck's example uses a lexicon which contained precisely the 150 words needed to compile the crossword. With generalised propagation it is possible to compile crosswords from a 25000 word lexicon. It is interesting to note that generalised propagation automatically yields a similar algorithm for generating crosswords as that developed for specialised crossword puzzle generating programs [Ber87].

A further way to control the evaluation of the crossword puzzle example is to divide the *word* goals into clusters, reflecting connected subareas of the crossword grid. A predicate *cluster* can be defined which combines all the words in a cluster:

```
cluster(A,B,C,D,E,F) :-
    word(A,B,C),word(A,D,F),word(C,E,F).
```

Generalised propagation can then be applied to the whole cluster:

```
propagate cluster(A,B,C,D,E,F)
```

In general propagation on *cluster* yields strictly more information than propagation on each of the *word* goals individually. However the amount of computing required to perform the propagation on *cluster* is also likely to be greater than propagating on the *word* goals individually.

If propagation is applied to larger subproblems, then we term it more "global". Global propagation is more expensive than local propagation but the amount of pruning of the search tree that results can be very significant.

4.3 Topological Branch and Bound

Generalised propagation is based on the idea of finding all answers to a query and eliciting the most specific generalisation. However it much more efficient to alternate the finding of answers and calculating the most specific generalisation. We call this "topological branch and bound".

For example after finding two words which satisfy a *word* goal in the crossword example, the system immediately attempts to "generalise" by finding common letters within and between words. If there are no common letters, the propagation process ceases immediately. Only if there are common letters does the system now search for a third word. As a result, the system very rarely needs to find more than a few answers to any *word* goal during propagation. This is the reason that the program has such an excellent runtime, even with a dictionary of 25000 words compiling real crosswords in a minute. It also accounts for Propia's good performance on the propositional satisfiability benchmarks despite its recalculating at runtime propagation information which in the CHIP program was hard coded by the programmer using demons.

Further optimisations can be applied if the predicate being used for propagation is defined by rules instead of facts. The exploration of a new branch in the search tree incrementally builds a new set of equalities. If, when exploring a branch, the partial set of equalities becomes larger than the current most specific generalisation, then the search on this branch can be stopped. This means that propagation can terminate even when the actual search tree is infinite. For example given the definition

```
p(s(0)).
p(s(X)) :- p(X).
```

propagation on $p(X)$ terminates after finding two solutions yielding the constraint $X = s(\_)$.

5 Conclusion

Constraint logic programming systems offer a range of tools for writing simple and efficient programs over various computation domains. Unfortunately it is not always possible to use different tools together. For example classical propagation cannot be used in programs working on domains such as Prolog III's trees.

A second drawback is that the logic of the program, when efficiency considerations are taken into account, has to be transformed extensively, or parts of it replaced altogether with rules expressed in some reactive language such as demons. The result for non-toy programs is a loss of clarity and, possibly, efficiency. If the programmer is not extremely competent these problems compound themselves, too often yielding a result which is not only inefficient but incorrect.

Generalised propagation makes a contribution to both problems. Firstly propagation can be used for arbitrary domains of computation, thereby improving orthogonality. Secondly the propagation annotations keep the control very simple and quite separate from the program logic, thereby preserving clarity and correctness.

Current experiments show generalised propagation to be a powerful and flexible tool for expressing control. More global propagation is more costly but it can bring a drastic reduction of the search tree. Local propagation is a cheap solution which is much easier to program and debug than guarded clauses or demons.

We are continuing to investigate the effectiveness of generalised propagation on a range of applications, studying its practical applicability to other computation domains, and following up the study of its lattice theoretic basis.

6 Acknowledgements

This work was funded by the Esprit 2 project, no.5291 *CHIC*. Thanks also to Bull, ICL and Siemens for providing a wonderful working environment at ECRC.

References

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

[CS90] P. Codognet and T. Sola. Extending the WAM for intelligent backtracking. In *Proc. 8th International Conference on Logic Programming*. MIT Pres, 1990.

[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.

[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.

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

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

[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.

[LW92] T. Le Provost and M. Wallace. Generalised propagation. Technical Report ECRC-92-1, ECRC, Munich, 1992.

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

[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.

[SS80] G.J. Sussman and G.L. Steele. CONSTRAINTS: A language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, January 1980.

[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.

[Wol89] D.A. Wolfram. Forward checking and intelligent backtracking. *Information Processing Letters*, 32(2):85–87, July 1989.