Introduction to CLP(BNR)

William J. Older

Notes from a course given at Carleton University,
Winter Term, 1995

Author’s Note

In addition to presenting this material at a course given at Carleton University in the Winter Term of 1995, I also gave a two-day intense version at the University of Montreal in 1995. Some of the individual chapters and problems had been previously presented in summary at the University of Marseille in 1993 and at the University of Orlean in 1995. I don’t recall which ones went where. The ODE paper was never properly published but shared with an associate in Germany who was attempting to diagnose the problems with the first orbiting X-ray telescope launched from ESA, which failed to maintain orientation because of some mechanical glitch. It was also shared with the people at Brandeis and Brown Universities who did a lot of work on it.

William Older
July 9, 2019

Preface

Since its beginnings in the 1980's the field of constraint logic programming has developed into a practical industrial tool, which can be used to solve some difficult combinatorial problems quite elegantly with very modest programming effort.

The newer interval-based constraint systems, such as CLP(BNR) provide constraints on real-valued variables as well as on integer-valued and boolean-valued variables. While still somewhat less mature, this technology is potentially more revolutionary in its consequences. At least in principle, it brings many of the classical problems treated in numerical analysis into the realm of logic programming and theorem proving. i.e. "black" magic gets replaced by "white" magic.

Interval constraint technology is now becoming commercially available in a number of systems (Prolog IV, ILOG, ALS/CLP(BNR)) with reasonable performance, and is no longer merely an academic curiosity, but has been used to solve real, hard problems.

This new technology does require that we learn new ways to think about problems, or rather, to *unlearn* many of the presumptions that we have been taught by exposure to conventional numerical practice. I hope that many of you will find this as liberating and exciting as I have.

Please excuse the many inconsistencies (including fonts and styles), not to mention downright errors, in the following (draft) chapters.

William Older

CLP = Constraints + Logic Programming

The word "constraint" is used in a variety of contexts from everyday life to physics, mathematics, and other sciences with much the same intuitive meaning, even when the formalization is different. A bead on a wire, a weight on a string, a beaker open to the atmosphere, a space mission limited by onboard fuel, a desert ecology limited by the lack of water are all examples of constrained systems.

CLP refers to the combination of constraints and logic programming. A constraint is declaratively just a relation--usually a mathematical relation-- but it differs operationally from other ways of treating relations. As relations, constraints fit very well into logic programming framework which is also based on relations.

CLP systems began appearing about 10 years ago and span by now a considerable variety in system and underlying technology as well as a variety of applications.

I. Historical Introduction to CLP

A. Forerunners

Prolog II, introduced in the early 1980's by A. Colmerauer, the founder of Prolog a decade earlier, is a forerunner of a CLP system in two quite different ways.

Prolog II departed from previous Prolog practice by altering the interpretation of the fundamental unification algorithm of Prolog. This was done by introducing the notion of rational trees to describe the space of terms. Conventionally, Prolog terms were deemed to be finite trees: unifications of a variable with a term containing that variable (eg. X= f(X)) should result in a failure, detected by an "occurs check". As a matter of efficiency, however, the occurs check code was usually incorrectly omitted. By permitting in theory the use of infinite trees, e.g. f(f(f(f(....)))), the omission of the occurs check could be formally justified at the price of weakening the conventional connection with classical predicate calculus. This so-called "rational tree" unification algorithm could be thought of as a system of equations to be solved in a certain formal structure (the rational trees), for which the axioms of equality provide a basis for the unification algorithm considered as rewrite rules. The change in viewpoint- which both freed unification from its traditional logical interpretations and provided a new interpretation in terms of equation solving– was later developed in Prolog III into a comprehensive CLP system.

The second relevant feature of note in Prolg II was the introduction of the freeze/2 primitive:

freeze(V, P)

has the operational effect of delaying the execution of goal P until the variable V becomes instantiated. This allows one to escape from the rigid sequential structure of Prolog and encourages the use of sets of asynchronous predicates which trigger one another through variable bindings, "instantiation propagators". This mechanism could be used effectively and efficiently to solve various puzzles concerning finite domains and to improve upon the traditional Prolog approach to arithmetic, and this stimulated interest in what later became various constraint technologies.

B. CLP Systems of the 1980's

In the middle of the 1980's several such systems were developed. The three best known are:

-Prolog III (Colmerauer, U. of Marseille and Prologia) expanded on Prolog II by adding rational term disequality, boolean variables subject to boolean relations, concatenation monoid on strings, and rational linear arithmetic (using the simplex method of linear programming). Nonlinear relations are delayed using something similar to freeze. Unification was generalized to mean the addition of new constraints to a constraint store and testing for satisfiability at each procedure call.

-CLP(R) (Laissez, Jaffar - a research product from IBM's T.J. Watson Laboratory) provided for solution of linear systems of equations and inequalities and also, like Prolog III, delayed nonlinear equations. Theoretical work done in connection with CLP(R) provided a general framework for discussing idealized semantics of constraint systems which closely parallels the formal semantics of Prolog. The actual implementation of CLP(R) was based on floating point computation, which gave it better performance than Prolog III but is not sound.

-CHIP (developed at ECRC and commercialized in Charme and later by ILOG) focussed primarily on specialized propagation techniques for finite domains (and boolean domains) which were based on algorithms devised in the AI community. These systems have been successfully applied to OR problems, particularly scheduling and discrete resource allocation.

C. CLP(Intervals)

The first interval based CLP system was incorporated in BNR Prolog (1988) based on ideas of Cleary (1986). The original version was based on intervals over the reals only, but could handle simple non-linear constraints. A similar, independently developed, system was described by Hyvoenen (1990) Another similar system (based on BNR Prolog) was sold as Interlog by Dassault Electronique in the early '90's.

In 1992 BNR Prolog's CLP system was extended to CLP(BNR) which provides a uniform treatment of boolean, integer, and real valued variables using interval propagation. Similar capabilities were announced in 1994 for the new version of IBM Prolog, just shortly before that system was discontinued by IBM. Prolog IV, the next generation system from U. of Marseille expected to be released in 1995, is also interval based. In 1994 commercial distribution rights to CLP(BNR) were granted to Applied Logic Systems of Newton, Mass., with shipment to commence in 1995.

Interval based CLP has a somewhat different character from previous approaches for continuous domains. Previous approaches were based on traditional technology (such as linear programming) and thus represent only a marginal change in approach and scope. The use of intervals, however, permits one to deal with uncertain or unknown data as well as non-linear models, and thus represents a significant enlargement of the space of problems that can be tackled. In addition, the interval approach can be applied also to discrete domain problems and there provides solutions similar to traditional finite-domain methods. Finally, because of the uniformity of the approach, problems which involve closely interacting mixtures of discrete and continuous domains can now be solved in ways not previously available.

In this course we will be using CLP(BNR) as working language. Most of the techniques discussed here for discrete domains will be available (in some form) in other CLP languages for finite domains. The continuous domain techniques will be for the most part specific to interval-based systems.

D. Other Systems

The system CAL developed at ICOT is a CLP system based on symbolic solution of polynomial equations over the complex field. The essential limitation to complex numbers and its symbolic basis gives it a quite different character from other constraint systems, more akin to symbolic systems like Mathematica or Maple. This and similar systems are very powerful but only in a somewhat limited range of problems.

Systems like ILOG Solver, which is sold as a C++ library, are at the fringe of the CLP realm because they are only rudimentary Logic Programming systems. As such they provide the bare mechanism of the constraint engine but without the conceptual framework of Logic Programming. In such a system one writes mainly calls to built-in C++ functions and the rudimentary logic programming environment is de-emphasized as much as possible. ILOG Solver appears to be at present commercially successful (partly because of the C++ connection), and has been used for large applications, but they appear to almost exclusively in the scheduling and resource allocation area pioneered by CHIP.

A number of systems (CHIP, Eclipse) provide a constraint framework in which one has the freedom (and the responsibility) to write customized solvers or value propagators or "demons". This is consider a feature, but little can be said in general about such systems unless the full set of demons and the details of their use is known. Arbitrary "heuristic" propagators can of course be made to perform certain computations, but do not necessarily mean anything, and there is a serious problem whenever the order of firing needs to be controlled.

II. Characteristics of CLP systems

CLP systems have either been modified Prolog systems (like Prolog III) or embedded in Prolog (like CLP(BNR)); the difference can be subtle. The relation with Prolog has several aspects which become design issues for the CLP system and which can also have profound effects on the use of the system.

A. Bactracking compliant

Prolog is a backtracking (or in some cases OR-parallel) system and the constraint system must be designed with this in mind. Backtracking is necessary since it provides the non-deterministic search needed for completeness even in the discrete case.
There are two major issues:

(1) when a constraint (the imposition of the constraint) is backtracked across, the state of the entire system (including the constraint store and any consequences deduced from it) must be restored completely to something equivalent to that which existed just prior to establishing the constraint. This can sometimes be done relatively easily through extensions of the Prolog trailing mechanism, but may result in restrictions on the data structures employed. This can be thought of succinctly as a "a Prolog failure must undo constraints".

(2). Conversely, a detected non-satisfiable (inconsistent) constraint store must cause a Prolog failure. Because of (1) this has the effect of removing some inconsistent constraints from the constraint store.

B. Incrementality

This is sometimes taken as a general term to include bactracking compliance as well. Here it means that in the Prolog setting problems are assembled a bit at a time as constraints are encountered in Prolog code instead of all at once as in e.g. classical linear programming. This requires that classical algorithms (if used) must be reformulated to work on partial problems.

C. Satisfaction test completeness

Incrementality raises a difficult design issue: should the consistency test used on each update of the constraint store be complete or not. If not complete, it can result in large amounts of unnecessary work at the Prolog level. On the other hand, a complete test is generally much more expensive than something which is less than complete, so it results in the constraint engine doing more work, much of which might become unnecessary as soon as the next constraint is seen. Prolog III has opted (where possible) for complete consistency since the variety of specialized algorithms it uses can provide such a test, and by creating specific algorithms tailored for the special situation the costs can still be kept reasonable. CLP(BNR) as a rule does not use complete tests (they would be prohibitively expensive in the general continuous non-linear and general discrete realms) but does only as much as is convenient for the underlying mechanisms. Thus the completeness issue depends on the constraint domains handled and the machinery employed, and in turn it determines the sort of theoretical and implementation issues that need to be addressed. Curiously, in places where direct comparisons between complete and incomplete systems are possible, the overall performance appears to be comparable, although some specific problems will do better under one regime or the other.

D. Instantiation Issues

In general, instantiation should propagate both ways between the constraint solvers and the Prolog system. Prolog instantiations which affect constraint variables need to be processed (usually immediately) by the constraint solver. Likewise, variables which can be fixed by a constraint solver need to be reflected at the Prolog level, especially if freeze is supported. If a variable can be involved in more then one constraint subsystem, this permits an interaction in which one solver fixes its value and propagates the binding to the Prolog level,which in turn propagates it to the other solver. If such a variable is bound by Prolog, there is a practical question of which solver to give it to first, as well as a theoretical question to ensure that in principle it doesn't matter which sees it first. In multi-solver paradigms such as Prolog III, however, we note that the solvers in fact respond to different and disjoint types of variables so these complications do not arise.

In CLP(BNR) there is only one solver for all types of constraint variables, and (except in rare circumstances) the solver takes precedence over Prolog.

E. Equality versus unification

The issue here is whether a term which describes a constraint is regarded as a term (i.e. an uninterpreted syntactic structure) or as a constraint (interpreted structure) when passed as a parameter or used in whatever plays the role of unification in the Prolog part of the system. For example, as terms 'X + Y = 2' is clearly false (since the left side is a functor + of arity 2 and the right is a constant of arity 0) while as an interpreted constraint it might be true. In CLP(R) for example such expressions are always interpreted (according to the syntactic rules of the language) as constraints. It follows then that the corresponding syntactic expression 'X + Y = 2' cannot be formulated in this languages unless some special new notation (which functions as 'quote') is introduced. CLP(BNR), on the other hand, distinguishes between term unification ('=') and arithmetic equality ('=='), and hence arithmetic and other constraint expressions are just Prolog terms until they are explicitly processed as constraints via a primitive call (the outfix operator '{ .... }'). This means that Prolog plays the role of a meta-language with respect to the constraint sub-language; more importantly, it means that symbolic processing of constraint expressions can be easily done in Prolog and that the use of constraints is always explicit in the language.

F. Delay

Most constraint systems permit constraints they cannot actually deal with; typically these are the non-linear constraints on continuous variables. There is then a decision of how to treat such a constraint: failure, error, or delay. Most have chosen to delay such constraints (using a mechanism like freeze) until such time that instantiations of some of there variables changes the constraint into a form which can be dealt with by the system. Interval based techniques do not need to do this explicitly, since they can 'handle' non-linear constraints, although the handling may often be in fact indistinguishable from delay.

G. Muliple solvers vs. single solver

Prior to the development of the interval based methods the direction appeared to be towards having a separate solver for each separate domain over which constraints were defined. This creates a number of problems in terms of potentially conflicting data structures, inter-solver protocols, and correctness issues. The interval based systems at present use a single-solver approach and thus avoid this particular set of problems. In future, however, we expect to see, e.g. symbolic solvers, which are complementary to the interval approach, integrated into CLP systems and some of these problems will again become relevant.

H. Output Considerations

Systems which involve significant symbolic processing (e.g. linear programming) or delay have a problem with regard to output, especially final output. Systems like Prolog III and CLP(R) have attempted to display whatever instantiations which may have been computed as well as a simplified set of remaining constraints, those whose "force" is unspent. "Simplified" here means usually "in terms of the original variables introduced by the user". However, the simplification problem (a form of variable elimination problem) is very difficult (NP hard or worse) even in the simple case of linear inequalities. This form of output seemed like a very good idea when people began experimenting with small problems, since the simplified symbolic output could often be used by the user to finish the solution of a problem outside the system's competence. But with problems involving hundreds (let alone tens of thousands) of user variables and many more intermediate variables the user can no longer make use of such answers and the system often cannot find sufficient memory or time to generate them.

With interval based methods, output takes the form of either an instantiation of a variable or a narrowing of its range of possible values, which can be expressed as a pair of floating point numbers. Such a representation loses information (specifically: all the constraints) ,of course, but is cheap to compute, easy to understand, and often useful (especially when the range is narrow).

Early Applications

Since CLP(BNR) is a way of solving concrete mathematical problems, it may be of use in any subject which such problems arise. Since the interval approach can work with incomplete information (a range of possible values instead of a single value), it may be more useful than traditional approaches in fields where precise information is difficult to obtain. But this field is so new and access to implementations has been so limited, that only a small range of applications has been explored.

The first useable prototype of CLP(BNR) was constructed at the Computing Research Lab of BNR in 1ate 1992. In the short span of six months following this, a number of successful applications of this technology had been explored, and it was felt that it would be useful to hold a workshop for exchanging results. This workshop-- ARIA '94-- was held in August of 1993 at BNR and brought together most of the people who had used CLP(BNR) up to that time. A brief description of some of the applications presented indicates both the range of application areas and the variety of mathematical problems treated.

1. Dr. Majumdar explained the use of CLP(BNR) and the earlier BNR Prolog system for the computation of performance bounds of tasking systems. This was a problem of bounding possible solutions to large systems of non-linear equations arising from performance modeling.

2. Dr. B. Nadel of Wayne State University described the application of BNR Prolog interval arithmetic to the synthesis of designs of automobile transmissions. This problem was subsequently reformulated in a much more satisfactory way in CLP(BNR).

3. Andre Vellino of BNR described CLP(BNR) solutions to bin packing problems of the sort that arise in system configuration.

4. Angelo Bean of BNR described the use of boolean equation solving in CLP(BNR) for detecting siphons and traps in Petri net models of software systems.

5. Mike Kelly of BNR discussed its use in the 2-dimensional layout of functional blocks in Field Programmable Gate Arrays, a discrete resource allocation problem.

6. Tammer Kamel of BNR described how CLP(BNR) could be used to analyze timing requirements in digital circuits and detect possible timing violations.

7. Bill Older of BNR described a CLP(BNR) approach to solving the Traveling Salesman Problem in 2-dimensions for small networks and the formulation of classical constrained optimization problems.

Since then further work has been done to develop techniques in several broad areas:
-continuous time scheduling problems (Older & Van Emden),
-continuous resource allocation scheduling (Older)
-ordinary differential equations (Older, Skuppin, & Hickey),
-integral and functional equations(Hickey).

These exploratory investigations have widened the scope of CLP(BNR) applications and now appear to be ready for exploitation, although they may be at the practical limits of the current technology.

IV. Concerning CLP(BNR)

At this point no one knows what the limits of CLP(Interval) technology may be in the near future- even the limits of the current technology are extremely hazy. In general we cannot at present predict whether a given problem can be usefully formulated in it, nor can we give anything like a recipe for such formulations, nor can we generally predict performance. Understandably, these uncertainties make it risky for anyone to commit themselves to the use of this technology, especially in the risk-averse contexts now so prevalent in industry. So it is natural to ask: why the intense interest in this technology? There are several answers:

A. First, it is a new paradigm for doing applied numerical mathematics-- indeed, probably the first new paradigm since the invention of digital computers. Some of the things which have been done with it are very radical --even bizarre--by all conventional standards. Part of the difference is that the CLP technology both *uses* different information and *produces* different information than does conventional practice. Learning to use it, it seems, is mostly a process of *unlearning* conventional ways of thinking.

B. It is mathematically rigorous, at least in principle. Furthermore, it makes it feasible to put applications also on a completely rigorous footing, since the code is usually quite small and extremely close to the mathematical statement of the problem.

C. Some of the practical applications which have already been done successfully (and easily) have *never* been done successfully using conventional techniques.

D. In NP-hard combinatorial problems where direct comparisons between CLP(BNR) approaches and the most sophisticated conventional algorithms are possible, the performance is very roughly comparable.

E. The field is changing rapidly, both in technique and technology; many of the successful applications mentioned above could not be done just months earlier because the right approach (or something) wasn't known at the time.

V. About the Course

The first half of the course is devoted to basic principles and their use in examples, with the aim of producing a good intuitive working knowledge of the technology and techniques that exploit it. In the second half of the course there will be more emphasis (though informal) on the theoretical aspects and the advanced topics which require it.

Each student is expected to do a substantial project --typically a prototype application -- as part of the course requirements. A sheet of project guidelines and timetable will be provided.

Review of Prolog

This session is devoted to a short review of the basic ideas of Prolog. The intention is to briefly cover aspects which will be relevant for CLP(BNR). In particular, the more theoretical aspects of Prolog and its connection with logic will be largely ignored.

Backtracking

Recall that computer science in the late 1960's and early 1970's was very much concerned with the design and formal specification of computer languages, or at least, of their syntax. Many of the languages still popular (e.g. Pascal) were devised at this time. One of the principle tools in widespread use for specifying syntax was --and still is-- Backus-Naur Form or BNF. Such a specification is familiar:

```
program :-   decl_part, body.
decl_part :-   'var' , decl_list.
decl_list   :- .
decl_list   :-   declaration, ';', decl_list.
declaration :-   identifier, ':', type.
body :- 'begin'  , statement_list, 'end'.
statement_list  :-  .
statement_list  :-  statement,  ';'  ,  statement_list.
```

Declaratively, you can read ':-' as 'consists of' and ',' as 'followed by'. As a specification it has the virtue of being both formal and very succint. Furthermore it could be converted manually into an implementation of the recursive descent type, either by direct coding, or by writing BNF interpreters, both of which were popular implementation techniques.

Consider what such a BNF interpreter would have to do: it has a stack to manage recursion, there is a stream of characters to be parsed, and some store (possibly a heap) in which to construct parse trees. The stack is initialized with `program`. The interpreter pops the top item (a 'non-terminal') from the stack, and tries to find a definition for it, i.e. a rule where `program` appears on the left of ':-'. If it does find such a rule, it can copy the body of the (first) definition to the stack, i.e. so `decl_part` is now the top. If the top is a "terminal" (in quotes), it is matched against the input string, and, if successful, pops the terminal and continues. In some cases there may be more than one rule for a non-terminal, as in the case of `decl_list` above. In this case, it is necessary to store the current file index in the stack, so if the first definition doesn't work out, it is possible to restore the file pointer and try an alternate definition. The failure of a string match triggers this "backtracking" process. In this context the need for backtracking or "non-determinism" is obvious.

[Sometimes it is necessary or desirable to terminate backtracking prematurely, for example when it suffices to know that there is at least one answer to a question. A special symbol '!' can be introduced with the operational meaning that it commits the computation to the current branch, by removing backtracking information generated since the current goal (the one matching the current clause) was called.]

An abstract version of this, showing the original goal explicitly is:

```
  :-  p.    % goal
p :-  l, b.
l :-  v, dl.
dl:-  .
dl:-  d, sc, dl.
d :-  i, c, t.
b :-  be, sl, e.
sl:-  .
sl:- s, sc , sl.
```

In the early 1970's Alain Colmerauer noticed that if one interpreted the 'p :-q ' as 'p is implied by q' and ',' as 'and' in symbolic logic then the algorithm given above provides a theorem prover for the theorem 'p' . The basic step, which is to infer p:-r,x from p:-q,r and q:-x, is called a resolution step (a combination of transitivity and weakening). (From the earlier theoretical work of Robinson it was known that this sort of elementary inference was the only kind needed.) So we have an analogy between a theorem proving technique (applied to problems of a special kind) and recursive descent parsing, and more generally, to any program which resembles recursive descent parsing.

Logic Variables and Unification

Of course, to build a compiler it is not enough to have a parser which simply tells you whether something is syntactically correct. One needs to build some sort of data structure (parse tree) and decorate it with useful bits of information and sometimes transform it a bit. So every BNF interpreter would add some notion of variable and some semantic primitives that manipulate these variables. Of course, if the parse failed down a particular branch, then some of these things would have to be undone to maintain consistency, and this aspect was seldom done quite right.

Now the concept of resolution had already been extended to theorem proving in first-order predicate calculus, which as a language already has a notion of variable. So we might ask: can these be used to provide the kind of variable we need to extend our language interpreter? A logic variable X is a pure "placeholder" which notionally represents *any* individual object in the domain of discourse, which in practice means any term we can construct in the syntax of the language. Multiple occurrences of the same variable in the same sentence of course must represent the same individual. (Different variables may refer to different individuals but they need not be distinct.) In the *clausal form* of predicate calculus all the variables in a single clause (written as a disjunction) are universally quantified; such a clause can serve as a rule (in the sense used earlier) in that it can be applied (not neccessarily successfully) for every possible substitution of its variables and can be used an arbitrary number of times. Clauses with at most one positive literal (so-called Horn clauses) , such as:

(X)(Y)(Z) { p(X,Z) \/ ~q(X,Y) \/ ~r(Y,Z) }

which can be transformed (using the logical definition p<-q = p \/ ~q and the law ~p\/~q= ~(p & q)) to:

(X)(Y)(Z) p(X,Z) <-q(X,Y) & r(Y,Z)

which puts it into the right form for the algorithm above. (By convention capitalized names refer to variables.)

(Note: If we move the universal quantifier (Y) into its natural place first:

(X)(Z) p(X,Z) \/ (Y) {~q(X,Y) \/ ~r(Y,Z) }

and then transform (using the law ~p\/~q= ~(p & q)) to

(X)(Z) p(X,Z) \/ (Y) ~ (q(X,Y) & r(Y,Z))

and then (using using (Y) ~ = ~EY and the logical definition p<-q = p \/ ~q) to get:

(X)(Z) p(X,Z) <- EY (q(X,Y) & r(Y,Z)).
Thus, even though all variables in a clause are formally universally quantified when the quantifiers are all scoped over the whole clause, variables appearing only in the body are in fact existensionally quantified when quantifiers are put into their natural position.)

In order to resolve the goal p(X..) (with whatever actual arguments it has) with the clause p(Y..) :- ..., we must first specialize both the goal and the clause so its head is equivalent to the goal; that is, its arguments are exactly the same up to a consistent renaming of variables. (Of course, we must not overspecialize or we might miss a possibility!) This process is called unification (denoted by '=' because it makes terms "equal") and is traditionally described in terms of finding substitutions which make the terms (the goal and the head) match exactly. There are four essential cases to deal with:
(1) a variable and a non-variable - just substitute the non-variable for the variable everywhere. For example, unifying the goal p(fred,2,U) with the head p(X,Y, g) sets X=fred and Y=2 (so these act like input variables) and U=g , so U acts like an output variable. Thus unification generates (in the simplest cases) a variable passing mechanism. (This is the place where an "occurs check" would be made.);
(2) an atom and an atom - succeeds if they are the same atom and fails otherwise (thus triggering backtracking). This is analogous to a terminal matching the input stream in the BNF interpreter;
(3) a structure and a structure - e.g. f(X,2)=f(Y,Z) for the structures to be identical they must have the same label (or "functor") and the same components (i.e. X=Y & 2=Z) ; and finally
(4) a variable and a variable - e.g. X=Y. This case is actually the trickiest. If X and Y are the same variable, this is trivially true, corresponding to the reflexive law of equivalence. Otherwise we must process this in a way which respects the other other laws, transitivity and symmetry. In Prolog this is usually done very efficiently by making the newer variable "point to" the older one, i.e. the oldest instance of each equivalence class is taken as the canonical representative of the class.

Thus the predicate calculus connection gives us a notion of variable (the logic variable) and unification gives us automatically a parameter passing mechanism and a basic equality relation on atoms and constructed terms. With these additions our interpreter begins to look at least a bit like a general programming language. However, unification also has a number of other properties which "come with the package" which many people find a bit strange at first. For example:

f(g(X,Y),h(Z)) = f(g(2,3),h(4)) -> {X=2, Y=3, Z=4}

f(g(2,Y),h(W))= f(g(A,3),h(A)) -> { A=2, Y=3, W=2 }

f(Z, h(4)) = f(g(A,3), h(A)) -> { Z=g(4,3), A=4 }.

Unification can be used to match patterns, extract subterms, exchange data between terms or subterms, and construct or replicate terms, often all at the same time, so it is a sort of general purpose data shuffler. It takes a while to get used to it, but once you do the conventional alternatives seem very clumsy.

Basically, Prolog consists of just the three things we have discussed: resolution, backtracking, and unification. This isn't quite enough to do everything you might want to do, of course, like actually adding 1 and 1 to get 2 or printing something or reading a file or opening a window .... So, a number of built-in primitives are added to the basic system to do really basic things outside of the Prolog universe, Usually the syntax for calling such primitives looks just like Prolog goals, but they may behave differently since they are playing by different rules.

Thus we see that Prolog represents a sort of compromise or hybrid between theorem proving technology and a programming language- not quite a theorem prover (by the standards of logicians) and not quite a programming language (by the expectations of programmers). The particular compromise is also a little surprising (for one can imagine taking a slightly different mix of features) and also quite delicate. Over the years there has been at work a dialectic process which sometimes pulls towards formal logic (as the name Logic Programming suggests) and sometimes in the programming direction (as seen in Prolog compiler technology and the extensions in commercial systems), but (fortunately I think) neither side has prevailed.

The fact that Prolog is associated with just a subset (the Horn clauses) of standard logic, although once regarded (and still so regarded by some, no doubt) as a deficiency to be overcome, appears now to be an advantage, for this subset of logic can be regarded as a logic (or proto-logic, sometimes called "positive logic") in its own right, which is in essence the common part of many of the competing systems of formal logic. The things omitted (from standard boolean logic) are boolean negation and the possibility of proving a disjunction p\/q without actually proving either p or proving q. From this we can see that it is very closely related to intuitionist logic. Research has also shown because of these omisions that it possesses certain useful formal properties such as the minimal model property.

Finally, we should note that Prolog has been very fecund in the sense that most of the formal structure can be retained even when it has been utterly transformed to become a parallel programming language, a concurrent programming language, a process algebra language, or a constraint programming language, all of which have occurred in the 1980's. Each of these developments reinterprets the basic symbols ':-' and ',' and the logic variable in quite different (and often incompatible) ways, yet retains much of the flavour of Prolog.

BNR Prolog Terms

BNR Prolog has extended the syntax (and semantics) of terms a little bit to make some things easier to do. We won't as a general rule be using these features very much in the course if they are avoidable, but I will describe them anyway.

Most Prolog systems have an idea of a list (of terms) and a special syntax that goes with it, consisting of brackets and comma separators, e.g.

```
[]
[a,b,c]
[X,b,D,3,4]
```

are three lists of definite lengths 0, 3 and 5 respectively. There is a special notation for representing indefinite lists (with no specific length),e.g.

```
[a,b,c|X]
```

is a list with 3 or more elements. (Usually funny things happen if the thing after '|' is anything other than a single variable.) These uses (which I will call "tail variables") are useful for expressing list recursions:

```
f([]).
f([X|Xs]):-  p(X),  f(Xs).
```

BNR Prolog has an alternative notation for indefinite lists: the previous example could also be written in BNR Prolog as:

```
        [a,b,c,X..]  .
```
This notation (but not '|') can be used to express any list ('[X..]') (which of course is not the same as any term.) It can also be used to express structures of indefinite arity, e.g.:
```
        f(2,3,4,X..),
        { P,Q,  Rest..}.
```
The 'X..' construct can only appear just before ')' , ']', or '}' which are the bracket constructs in the language. If 'X..' is used anywhere in a context, 'X' (X without the dots) represents [X..], so a list recursion could also be written as:
```
        f([]).
        f([X,Xs..]):-    p(X),  f(Xs).
```

BNR Prolog also permits the functor of a structure to be a variable, e.g `F(U,V,W,X..)`
represents any structure with arity at least 3.

Finally, BNR Prolog supports cyclic structures (infinite trees or rational trees) so
```
        f(X)=X        -->   X=f(f(f(f(f(...))))),
        [2,X..]=X     -->   X=[2,2,2,2,2,2,2,...]
```
both "infinite" structures. This is easy, since one need only omit the occurs check as previously discussed; the hard part is to efficiently guarantee termination on the *next* unification or primitive call made on the result. Infinite trees have a natural interpretation as generalized state machines (and hence as formal languages) and rational trees as finite state machines (regular languages), and unification generalizes the notion of machine equivalence. More generally, for many applications the natural data structure is some sort of graph, and cyclic structures makes it possible to construct graphs.

Narrowing concepts in Prolog

In this chapter we are going to look at Prolog from a more formal algebraic point of view as an example of something called a "narrowing algebra." Later on this will be generalized a bit to cover interval based constraint systems, but it is useful to discuss it first for Prolog.

In the last chapter we discussed the subtle historical and technical relations between Prolog and symbolic logic. A lot of theoretically inclined courses on Logic Programming spend a lot of time on this issue, but as a practical matter in Prolog programming it seems to be seldom of any use. The sorts of general problems which arise in practical Prolog programming are generally not addressed by that sort of theory. Most of the problems, it turns out, arise from the use of primitives (sometimes called "non-logical" primitives) which is not covered at all by the Logic models. Something else is needed.

For this analysis it is best to forget about backtracking for a while and concentrate entirely on what I call "forward computation", i.e. along any one branch. Once we have understood what can happen on any branch, the total behaviour is just the disjoint union of the behaviour on all the branches. Also, since in BNR Prolog, we notionally allow infinite trees, we will here ignore termination issues: conceptually speaking, non-terminating computations generate answers also.

Unification and Narrowing

If we fix the set of symbols (including operators) to some definite set (possibly even all the symbols permitted), we can then imagine the (infinite) space of all the terms that can be constructed from them according to the syntactic rules. Terms that contain no variables are called *ground* terms, which will be denoted by G and unification is just equality on G. G of course has a rich "algebraic" structure--sufficiently rich that it can model any (first-order) theory whatsoever. But for our purposes here we will just think of it as an unstructured set.

Now for every term t (possibly with variables) there is a *set* of ground terms **t** which is the set of ground terms which successfully unify with t:

$$\mathbf{t} := \{\ t' \in G \mid t=t'\}.$$

If two terms s,t are equivalent (by a relabeling of variables), then **s**=**t** (set equality of course). So from now on, we embed the space of all terms into the powerset on G, and think of a general term as representing the ground terms which are its instances. We call this space **T**. Note that not all elements of the power set were in the range of this map, so only very special sets of terms are in T.

Having made this identification, the space of terms now has some structure, since it is partially ordered set inclusion. The term consisting of a single variable, since it can represent any term, is a maximum element (or "top") in the partial order. (There is no term corresponding to the empty set.) Now consider what happens when we unify two terms, say s=t. After the unification (if successful) the two terms have become equivalent, call the common result r. By intention r is a specialization of both s and t and is the most general common specialization. As sets **s** and **t** have an intersection **s**$\cap$**t**, and if this intersection is in **T** , then it must be **r**. By going through the details of performing such an intersection, one can derive the unification rules presented previously, and this shows that unification corresponds to intersection, with the empty set $\emptyset$ corresponding to failure. In order to get closure under unification, we adjoin $\emptyset$ to **T** . (even if it is not associated with a term- think of it as the "empty" term). With this slight change **T** becomes a semi-lattice (and in fact a lattice, although we will have no use for the "join" of two terms).

We have already talked about unification as formally representing an equivalence relation between terms, that is a reflexive, symmetric, transitive relation. There is a correspondence between these properties and those of intersection:

x=x	**x**=**x**$\cap$**x**
x=y <-> y=x	**x**$\cap$**y**=**y**$\cap$**x**
x=y & y=z-> x=z	**x**$\cap$**z** $\supset$ (**x**$\cap$**y**)$\cap$**z**

One could call this the "narrowing interpretation of equality"- when things are represented by sets of possibilities, and equal things have equal sets of possibilities, then knowledge of equality entails that only the common possibilities survive.

Operators

Consider the function (operator) $\boldsymbol{t}$ defined on **T** by

$$\boldsymbol{t} : \mathbf{x} \dashrightarrow \mathbf{t} \cap \mathbf{x} .$$

Such operators have several main, easily proved, properties:

(contraction)	$\mathbf{x} \supset \boldsymbol{t}(\mathbf{x})$
(monotonicity)	$\mathbf{x} \supset \mathbf{y} \rightarrow \boldsymbol{t}(\mathbf{x}) \supset \boldsymbol{t}(\mathbf{y})$
(persistence)	$\boldsymbol{t}(\mathbf{y}) \supset \mathbf{x} \rightarrow \boldsymbol{t}(\mathbf{x}) = \mathbf{x}$.

Operators can be combined directly by function composition. These unification operators then also commute, since:

$$\boldsymbol{s}(\boldsymbol{t}(\mathbf{x})) = \mathbf{s} \cap (\mathbf{t} \cap \mathbf{x}) = \mathbf{t} \cap (\mathbf{s} \cap \mathbf{x}) = \boldsymbol{t}(\boldsymbol{s}(\mathbf{x})).$$

Pure Prolog (the subset intuitively defined by clause resolution and unification with no primitive calls) is a sequence of such operators and therefore satisfies these properties.

Predicates as operators

Now we want to consider an arbitrary Prolog goal $p(t_1,t_2,...)$ as an operator (named by p) applied to its argument list regarded as a term, i.e. as if it were written as a function $p([t_1,t_2,...])$, its output being the resulting term $[t_1',t_2',...]$ after the call. The operator p can be a pure Prolog operator (written entirely in basic Prolog) or it can be a built-in primitive or any combination theoreof, but note that we do not include any side effects (output routines, or operating system calls) in the semantics; we are looking only at the effect on the Prolog arguments. Consider whether such an operator is contracting, monotone, or persistent:

Contracting: Normally every such operator will be contracting because of the "write once" or immutable character of logic variables. The only exception would be a primitive which changes the value of a symbol (or other instantiated type) to a variable (shudder!) or to a different symbol (or other instantiated type). (This is possible, of course, but hopefully very rare.)

Persistent: Normally this is the case as well, especially if primitives use an internal unification routine to update output arguments. An exception would be any primitive that has some internal state (not passed as an input argument), such as a random number generaotr with an implicit seed. In particular, most type tests like `integer(X)` or `symbol(X)` are persistent. An exception is the var test `var(X)`. A special case of interest is the standard "is" primitive: in

```
V is Expression
```

normally V is a variable and gets bound to the value of the Expression during the call. Consider a sequence in which this goal is repeated:

```
V is Expression,...,V is Expression
```

On the second occurrence V is already instantiated, so for "is" to be persistent it is necessary that the evaluation of the Expression yield the exact same result (which of course it should) and the primitive must then do a numerical equality test instead of a bind.

Note that a persistent operator need only ever be called once with a particular argument.

Monotone: This is the most likely to fail: primitives that expect instantiated input arguments and fail when the input arguments are not instantiated will not be monotone. (If the call can ever succeed with some instantiated argument, then monotonicity implies that it must succeed when that argument is a variable, all other arguments being unchanged.) Note that for this reason `var(X)` is monotone, but standard Prolog arithmetic is not.

For primitives one needs to check their specifications and usually do some testing to determine which properties hold and for which arguments.

Usually in a clause execution, unification is used to pull inputs apart into separate components, operators (calls) are applied to the components, and then the results glued back together using unification, but all this could be written in purely functional terms using function composition, although it often would be comparatively clumsy. (As an exercise, take a small Prolog predicate and convert it to such a functional form.)

What makes the properties above interesting is first of all that:
(1) the composition of contractions is a contraction
(2) the composiiton of monotone operators is monotone

(3) the composition of persistent operators is persistent.
(Exercise: prove these staements from the definitions.)
From this it follows that these properties {abreviated CMP} of an operator are unaffected by composition with unification, since unifications possess all three. And knowing, for example, that a clause uses var(X) tells us that the predicate it is part of is in danger of being non-persistent (unless there is another clause that repairs the damage) and that a clause which uses standard arithmetic is in danger of being non-monotone. This is useful both diagnostically and normatively.

From the above closure properties we see that if we start with a set S of operators satisfying all three properties {CMP} as generators, and form S*, the set of all the operators we can get from them by composition, they all have these properties. The identity function, written 1, is included in S* (formally: it is the composition of the empty set of primitives), so S* is a monoid (semigroup with identity) under composition as product, what we will call a CMP-monoid. (This monoid is also a partially ordered set and has extra structure as well, but we won't be discussing it now.)

Now we get a very strong result:

Theorem: Let p belong to the CMP monoid S*. Then there exists a term $\mathbf{t} \in \mathbf{T}$ depending on p such that for all $\mathbf{x} \in \mathbf{T}$, $p(\mathbf{x}) = t(\mathbf{x}) = \mathbf{t} \cap \mathbf{x}$.

Proof: By hypothesis p is a persistent monotone contraction. Let $\mathbf{u}$=mga(p) be the most general argument admissible for p. (Standard Prolog predicates have a definite arity n, so the mga is a list of n variables; for BNR Prolog predicates which can be of indefinite arity the mga is therefore [X..].) For p to succeed we must have $\mathbf{u} \supset \mathbf{x}$, otherwise (p always fails) we can take $\mathbf{t} = \varnothing$ Then since p is contracting, $\mathbf{x} \supset p(\mathbf{x})$, and since p is monotone, $p(\mathbf{u}) \supset p(\mathbf{x})$, so $\mathbf{x} \supset \mathbf{x} \cap p(\mathbf{u}) \supset p(\mathbf{x})$. Then since p is monotone, we have $p(\mathbf{x}) \supset p(\mathbf{x} \cap p(\mathbf{u})) \supset p(p(\mathbf{x})) = p(\mathbf{x})$ (since p is persistent and hence idempotent), so $p(\mathbf{x}) = p(\mathbf{x} \cap p(\mathbf{u}))$. But since $p(\mathbf{u})) \supset \mathbf{x} \cap p(\mathbf{u})$ and p is persistent, $p(\mathbf{x} \cap p(\mathbf{u})) = \mathbf{x} \cap p(\mathbf{u}) = p(\mathbf{u}) \cap \mathbf{x}$, and we take $\mathbf{t} = p(\mathbf{u})$.

One can interpret each $p(\mathbf{u})$ as the "local universal answer" to p, for each branch of the computation. The (very likely infinite) collection of all the $p(\mathbf{u})$'s from all branches (as a formal disjunction) is then the global universal answer, and a similar theorem could be formulated for it by distributing intersection over disjunction.

Corollary: The operators of S* commute.

Proof: immediate.

Exercise: Prove the corollary directly from the properties CMP.

Messy exercise: From the commutativity of the operator composiiton monoid, show that the ',' operator appearing in clauses also commutes.

This gives us an alternate, rather abstract, way of characterizing the notion of a "pure Prolog" system, one which doesn't depend on how it is implemented. It has been summarized in the phrase: "the question, when properly narrowed, is the answer." We will return to this framework later and generalize it to handle constraints as well as Prolog.

Standard Prolog Arithmetic

The conventional standard arithmetic which has been added to Prolog uses the primitives "is" and the various relational arithmetic operators if the forms:

```
Var is Expression
Expression  Op  Expression
```

where Expression denotes a valid and *fully instantiated* arithmetic expression. These forms are therefore non-monotonic, although both are persistent contractions, provided `is` defaults to equality when its left argument is instantiated. It follows that, even with integer arithmetic, the use of these primitives gives rise to impure (sometimes called non-logical) behaviour. Since most large applications need to do some arithmetic, this is one of the main reasons why large Prolog applications do not enjoy the same properties of the small teaching examples.

When this form of arithmetic was pragmatically extended to handle floating point numbers, an additional problem was introduced: the results are no longer correct. In systems where exact arithmetic equality ('==') (i.e. exact match of binary representations) is used, for example, one gets anomalies such as the unexpected failure of 1.21 == 1.1 * 1.1. This occurs because 0.1 is an infinite decimal in binary representation, which gets truncated in practice, and as a result the two sides differ by a rounding bit. More complex cases occur whenever arithmetic is done. Fuzzing equality (as was done in APL for example) may mask some but not all such problems, and in addition makes equality non transitive (and possibly even non-symmetric, depending on how it is done).

Thus the problem of constructing a Prolog compatible arithmetic can be viewed as twofold: restoring correctness and restoring monotonicity, without sacrificing the other two properties. One way of doing this is to use infinite precision rational arithmetic (to restore correctness) and delay to formally restore monotonicity, as in Prolog III. Rational arithmetic is, however, insufficient since it does not supply answers to e.g. "X is sqrt(2)", and one is forced to extend the number system to the computable reals. Delay is also counter-productive, since at the end of the computation one two often has most of one's arithmetic still in the deferred state, and it no longer functions as a control on Prolog execution (which is one of its major roles). ←

One way around this impass is to move to an interval representation: each arithmetic variable is associated with an interval bounding its range of possible values, and arithmetic operations result in a narrowing of ranges. Done properly, this restores both monotonocity and correctness, while permitting the use of ordinary floating point arithmetic (with careful control of rounding direction). The next section describes the result of this approach in detail.

Syntax and Semantics of CLP(BNR)

CLP(BNR) is Prolog extended with an interval based constraint system which supports booleans(B), integers(N), and reals(R). It consists fundamentally of three predicates: one to create numeric variables, one (':') to to query the current value of its range (domain), and one to establish constraints ('{...}'). There are also some specialized utilities which will be covered in later sections.

The explicit creation of numeric variables is done through one of the following "type declarations":

```
X:real,
X:real(L,U)
X:integer,
X:integer(L,U)
X:boolean.
```

Here X is (normally) a variable or a list of variables, and L and U are either instantiated arithmetic expressions with L=<U or variables (conceptually representing -/+ infinity). If X is instantiated already, then X:real is equivalent to float(X), X:integer to integer(X), and X:boolean is true if X=0 or X=1. Such declarations restrict the possible subsequent instantiations of X, just as "X = F(_,_,_)" limits the possible values of X to be an arity 3 structure in Prolog.

For such a typed variable X

```
domain(X,  Type)
```

returns the type and bounds (for integer and reals only) in the same form as used by ':'; it fails unless X is a typed variable. The bounds returned are the "current" bounds and reflect whatever narrowing has occurred because of established constraints.

Since domain fails whenever X is instantiated, there is for convenience also a predicate range which returns the bounds as a list, but which also works on instantiated numerics. For many purposes (e.g. display) one is more interested in the bounds than the type, and range is therefore preferable, especially for integer quantities which tend to become instantiated. As a matter of terminology, it is useful to speak of "numeric quantities" consisting of either "numeric constants" or "numeric variables".

In terms of the partial order (subsumption order) of terms, any typed variable is a subtype of an untyped variable. For two typed variables X,Y of the same type, X is a subtype of Y if and only if its bounding range is smaller, and the unification of X and Y has as bounding range the intersection of the two ranges (when not empty) or failure (when the intersection is empty). For two type variables of different kinds, one regards boolean as a subtype of integer and integer as a subtype of real (via the natural embeddings) and this is considered as well as bounding interval inclusion. With these rules, the partial order on terms has been extended to cover typed variables wherever they may occur. The fact that booleans are represented as 0 or 1 (rather than true and false) and can be treated as integers and reals means that boolean variables can appear in arithmetic expressions, e.g.

```
        { Z is  B*X + (~B)*Y}
```

which plays the role of a conditional assignment in conventional languages.

All constraints are established by executing *constraint goals* of the form:

```
                { A } or
                { A1,A2,  ....}.
```

where the A's are arithmetic relations. Arithmetic relations are written in the usual way, using 'is', '==', '=<', etc. and the usual arithmetic expressions. The form "{A1,A2 }" is equivalent to "{A1},{A2}", although the former is often more convenient to write, but the latter is sometimes easier to debug if there is a failure.

With respect to the extended subsumption order, constraint goals are contracting, monotone and persistent and hence commute with each other and with other such goals, e.g. unifications.

Therefore the order in which constraints are imposed will not affect the results. However, the order of ':' and domain calls relative to each other and to constraint goals does matter.

Constraint goals affect the system state in two distinct ways: one is by generating implicit declarations (see next paragraph), and the other by narrowing the ranges. When processing constraint goals all intermediate variables corresponding to subexpressions and any untyped uninstantiated variables are assigned default types according to a set of type inferencing rules. In most cases these types will be appropriate, and explicit declarations will therefore not be required. (The default types assigned in any particular situation can be determined experimentally by making the intermediate variables explicit ; by explicitly typing such variables one can then override the defaults). In the present version of the system, once a type has been assigned to a variable, it cannot be changed by a subsequent declaration. As a result, declarations may not commute with constraint goals which infer a different type for that variable. Declarations of the same type but different ranges do commute.

Ideally, there should be only three outcomes to a constraint goal: (1) it adds the new constraint to the system and succeeds, (2) the new constraint is provably inconsistent with the exisitng constraint system and it fails, or (3) there is something syntactically wrong with the constraint and an exception is generated. (Note: in the present version, some types of syntactic errors may result in failure rather than an exception condition.)

CLP(BNR) Syntax and Semantics

In the last section we presented a conceptual model of CLP(BNR) as an extension of Prolog, regarded as a narrowing algebra. This conceptual model will be the basis for everything we will do in this course.

Some Examples

(The following examples were generated using version 4.3 of BNR Prolog.)

Consider the following query:

```
?-  [M,N]:integer(0,8), { M == 3*N}.
```

with response

```
        ?- [[_H583, _H588] : integer(0, 8), {_H583 == 3 * _H588}]
           where [_H583 : integer(0, 6), _H588 : integer(0, 2)].
```

The query is echoed, although variable names have been replaced by system-generated global names (which may differ from these appearing here). To avoid problems with the system names being different each time and to improve readability, I will henceforth substitute the original names, prefixed with '_', in these examples, e.g.

```
   ?- [[_M, _N] : integer(0, 8), {_M == 3 * _N}]
        where [_M : integer(0, 6), _N : integer(0, 2)].
```

The post-query domain information is expressed in a "where" clause which will be automatically supplied whenever a term is output using the predicate `print`. Note also that *both* variables have had their ranges narrowed by the constraint.

Now if we add an additional constraint, `M > 3` :

```
?-  [M,N]:integer(0,8), { M == 3*N}, { M > 3}.
     ?- [[6, 2] : integer(0, 8), {6 == 3 * 2}, {6 > 3}].
```

There is now a *unique* and *exact* answer, which has been instantiated.

Things are a little different when dealing with real variables. Consider the following query:

```
?-X::real(1,3),   {Y**2==X}.
```

Note that Y will by default be considered as if it had been declared as `Y:real`. With Prolog we are use to answers being instantiations ("substitution instances") of the query, but here there is not enough information in X to uniquely determine Y (or X). So what do we get? The response to the above query is:

```
    ?- [_H527 :: real(1, 3), {_H541 ** 2 == _H527}]
        where [_H527 : real(1.0, 3.0),
                   _H541 : real(-1.73205080756888, 1.73205080756888)].
```

Note that since the sign of Y is unspecified, so the answer permits negative Y values. Note also that the interval for Y contains points (from -1.0 to 1.0) which are *not* possible solutions. But Y is the smallest *interval* that contains all the solutions and allows for worst-case rounding error.

Now try either:

```
?-X::real(1,3),   Y:real(0,_),   {Y**2==X}.
```

or

```
?-X::real(1,3),   {Y>=0},   {Y**2==X}.
```

or

```
?-X::real(1,3),   {Y**2==X},{Y>=0}.
```

and get

```
    ?- ...
        where [_X : real(1.0, 3.0),
                  _Y : real(1.0, 1.73205080756888)].
```

All are equivalent.

Now try

```
?-  {Y>=0,Y**2==X},   {X==3}.
```

and get

```
    ?- [{_Y >= 0, 3.0 == _Y ** 2}, {3.0 == 3}]
          where [_Y: real(1.73205080756888, 1.73205080756888)].
```

Note that X has been *coerced* to a float and instantiated to a point value, but Y is still an interval, though an exceedingly small one. This is because the rounding error in taking the square root makes the value of Y slightly uncertain; indeed no floating point number and no

rational number is the square root of 3. You can't actually see the difference in the bounds in this particular case because the printing resolution isn't high enough, but you can test this with:
```
?-  {Y>=0,X==Y**2},  {X==3},  range(Y,[L,U]),  L=U.
```
produces 'NO' while
```
?-  {Y>=0,X==Y**2},  {X==3},  range(Y,[L,U]),  L<U.
```
produces 'YES'. You must avoid or be very careful when copying such answers for entry later (especially when the bounds appear to be identical), or you may introduce numerical errors.

Any floating point constant, which is entered as a (terminating) decimal, gets "fuzzed" slightly when put into a real variable:

```
?-  X:real,  {X==1.73205080756888}.
     ?- [_X : real, {_H442 == 1.73205080756888}]
          where [_X : real(1.73205080756888, 1.73205080756888)].
```

WARNING: This fuzzing helps compensate for the conversion from decimal to binary, but is not necessarily enough to preserve exact properties:
```
?-  X:real,  {X**2==3.0},{X==1.73205080756888}.
NO
```
To avoid this, numerical data should normally be entered as a declaration, with the bounds reflecting the actual effective precision.

Here is a more complex example, involving two linear equations:
```
?- {1==X + 2*Y, Y - 3*X==0}.
    ?- {1 == _X + 2 * _Y, _Y - 3 * _X == 0}
          where [_Y : real(0.428571428571429, 0.428571428571429),
                 _X : real(0.142857142857143, 0.142857142857143)].
```

And here is one involving a pair of non-linear equations (one involving a transcendental function):
```
?- [X,Y]:real(0,1), {tan(X)==Y, X**2 + Y**2 == 1 }.
     ?- [[_X, _H782] : real(0, 1),
         {tan(_X) == _Y, _X ** 2 + _Y ** 2 == 1}]
          where [_X : real(0.649888946665696, 0.649888946665697),
                 _Y : real(0.760029181677751, 0.760029181677751)].
```

In general, sets of equations (even when the number of equations equals the number of unknowns) will have multiple solutions; there is a non-deterministic predicate `solve` which usually splits the solutions:

```
?- X:real(0,1), {0==35*X**256 -14*X**17 + X}, solve(X).
     ?- [0.0 : real(0, 1),
         {0 == (35 * 0.0 ** 256 - 14 * 0.0 ** 17) + 0.0},
         solve(0.0)].
     ?- [_X : real(0, 1),
         {0 == (35 * _X ** 256 - 14 * _X ** 17) + _X},
         solve(_X)]
          where [_X : real(0.847943660827315, 0.847943660827315)].
     ?- [_X : real(0, 1),
         {0 == (35 * _H634 ** 256 - 14 * _H634 ** 17) + _H634},
         solve(_X)]
          where [_X : real(0.995842494200498, 0.995842494200498)].
YES
```

Note that the first solution (0.0) was actually instantiated.

Some Caveats

The current implementation of CLP(BNR) (V4.3) has - partially for historical reasons and partly because of implementation difficulties - several anomalies that do not quite fit the conceptual model discussed in the last section:

(1) Historically, Prolog has made an absolute distinction between integer and floating point constants. As a result, when X is declared as a real variable, it is only allowed to take floating point values, and when declared an integer it is only allowed integer values (which is ok). The former violates the idea that integers should be regarded as a subset of reals. This has some annoying consequences when writing code, as we will see below.

(2) Related to this, the system does not currently allow one to change the type of a variable by a subsequent declaration. Since such changes are often the result of a programming error, this is apparently a mixed blessing, but does formally violate the model above.

(3) Also related to this: an arithmetic equality between arithmetic variables of different types does not merge the two variables into a single variable of the intersection type, although they will in fact be constrained to be equal. For example,

```
X:real, N:integer, {X==N}, N=1
```

results in X being equal to 1.0 and N to 1.

Also note that in general declarations and `domain` calls do not commute:

```
        X:real,   domain(X,T)
```

works, but

```
        domain(X,T),   X:real
```

fails (assuming X was initially a plain variable), and in general neither declarations nor domain calls commute with constraint goals.

Programming Considerations

The interaction between arithmetic constraints and Prolog introduces new algorithmic possibilities as well as some new problems, which sometimes have non-obvious solutions. As a simple example, consider a predicate to sum a list of numeric quantities. In normal Prolog, the simplest solution (for a list of numeric constants) is:

```
sum( [], 0).
sum( [X|Xs], Sum):- sum( Xs, S), Sum is X + S.
```

This version has the drawback that it is not tail-recursive, so requires more stack storage. A tail-recursive version introduces an auxiliary predicate and an accumulator variable:

```
sum( List, Sum):- $sum(List, 0, Sum).
$sum([], Sum, Sum).
$sum([X|Xs], S, Sum) :- S1 is S + X, $sum(Xs,S1,Sum).
```

With constraints one also has the following option, which is tail-recursive without needing the auxiliary predicate:

```
sum( [], S):- {S==0}.
sum( [X|Xs], Sum):-  {Sum == X + S}, sum( Xs, S).
```

The first clause could also be written less explicitly as:

```
sum( [], 0.0).
```

This will work (but "sum([], 0)" won't !) because the input argument S is by default real; as is the output Sum (unless explicitly declared). Since one expects that the sum of a list of integers should be an integer, this formulation is not ideal. Another advantage of using the more explicit form using '==' is that it will work when the incoming argument is an arithmetic expression (e.g. the call "sum(L, 2 + Y)"), while the second form will not because the *expression* "2 + Y" is not the same as the *expression* "0".

Often- perhaps even usually-the best possibility in BNR Prolog is to perform the sum symbolically:

```
sum( [], 0).
sum( [X|Xs], X + S):- sum( Xs, S).
```

and convert to a constraint afterwards:

```
            ...sum( List, S), { Sum is S }, ...
```

This has several advantages:

(1) one needs only one version of such utilities for both constraint and non-constraint (including purely symbolic) use;

(2) because we have used 'is' and a new variable on the left of it, the result will be an integer whenever all the list elements are integers;

(3) the sum predicate is pure Prolog;

(4) it may be *much* easier to debug; and

(4) the cost of building the summation structure is offset by fewer calls to {}.

This pattern of working symbolically and only afterwards converting to constraints is one to keep in mind, as it seems to be widely useful. It is counter-intuitive by conventional (even conventional Prolog) thinking, which is usually motivated by compilation efficiency issues.

Constraints and Prolog Control

The interaction between constraints and Prolog can be subtle, but is almost always beneficial. The next example illustrates both the problems created by standard arithmetic in Prolog and the use of constraints in controlling Prolog execution. Consider writing a predicate that relates a list to its length- e.g. can either generate a list (of variables) of a specified length *or* compute the length of a list. In ordinary Prolog one is forced to write something like this:

```
plength( List,  N):-  integer(N),!,N>=0,  $mklist(N,List).
plength( List,  N):-  list(List),$plength(List,N).

$mklist(0,[]).
$mklist(N,[X|Xs]):- N>0, N1 is N - 1, mklist(N1,Xs).

$plength([],0).  % don't cut - might be indefinite list
$plength([X|Xs],N):- $plength(Xs,N1), N is N1 + 1.
```

Note that the auxiliary predicates `$mklist` and `$plength` are themselves non-logical because of their use of non-monotone Prolog arithmetic, as discussed previously. The umbrella predicate `plength` *partially* compensates for this by testing the instantiation pattern (using tests which are also non-monotone) in order to call the appropriate subroutine. The net result of all this finagling, however, works fairly well:

```
?- plength( [1,2,3,4], N).
     ?- plength([1, 2, 3, 4], 4).
YES

?- plength( X, 4).
     ?- plength([_H416, _H418, _H420, _H422], 4).
YES

?- plength( [X|Xs], 4 ).
     ?- plength([_X, _H430, _H432, _H434], 4).
YES

?- plength( [_,_], 4).

NO
?-  plength(  [X,Xs..],-3).

NO


?- plength( [X|Xs], N).     % nondeterministic
     ?- plength([_X], 1).

     ?- plength([_X, _H433], 2).
     ?- plength([_X, _H433, _H435], 3).
     ?- plength([_X, _H433, _H435, _H437], 4).
     ?- plength([_X, _H433, _H435, _H437, _H439], 5).
     ?- plength([_X, _H433, _H435, _H437, _H439, _H441], 6).
     ?- plength([_X, _H433, _H435, _H437, _H439, _H441, _H443], 7).
etc
```

The last answer, which is an infinite backtracker (as it must be), is a serious source of problems in Prolog, since the last one of these executed will mask all previous choicepoints; Prolog has no nice mechanism to deal with this problem.

Consider now a (deceptively) simple version using constraints:

```
length( List, N)  :-  N:integer(0,_), $length(List,N).

$length([],0).
$length([X|Xs],N):- {N1 is N - 1, N1 >=0}, $length(Xs,N1).
```

Here we have used a declaration in the umbrella routine to ensure that list lengths are non-negative integers (as one would expect), and we have used 'is' with the new variable as left argument in the induction clause to propagate the type integer down the chain of variables, so it is safe to use 0 in the head. This code is much smaller and cleaner-- no instantiation tests and no '!' required--and it is also tail-recursive. It also gives the same answers to all the above queries.

But, whereas
```
?- plength( X, N).
```
just fails (since X is not a list), we now get instead:

```
?- length( X, N ).
     ?- length([], 0).
     ?- length([_H237], 1).
     ?- length([_H237, _H409], 2).
     ?- length([_H237, _H409, _H584], 3).
     ?- length([_H237, _H409, _H584, _H762], 4).
etc.
```
And, perhaps unexpectedly, we can now do:

```
?-  N:integer(5,7),  length(L,N).
     ?- [5 : integer(5, 7),
          length([_H382, _H548, _H714, _H880, _H1046], 5)].
     ?- [6 : integer(5, 7),
          length([_H382, _H548, _H714, _H880, _H1046, _H1212], 6)].
     ?- [7 : integer(5, 7),
          length([_H382, _H548, _H714, _H880, _H1046, _H1212, _H1402], 7)].

YES
```

Thus, provided we can establish initial bounds on the list length, we can 'tame' the infinite backtracker problem.

Exercise: Work through the logic to see exactly why this works as it does: why is the first answer a list of length 5? why does it stop when it should?

The moral of this story is that constraints and Prolog are complementary. Not only are they conceptually compatible, but constraints provide the sort of arithmetic capability that Prolog has always needed, while Prolog provides the symbolic processing and programming environment needed to make effective use of constraints.

Constraints and Prolog Negation

The usual negation-by-failure construct of Prolog works with constraints according to the normal rules:

```
not( { C } )
```

succeeds only if the constraint C fails, i.e. C is provably inconsistent with the constraints already in the system. For example, if, having already declared variables X and Y and established constraints involving them, we ask

```
...not(   {X>=Y})
```

and it succeeds, it indicates that X cannot possibly be larger or equal to Y. (This may not be obvious from looking at their ranges, which may in fact overlap.) Furthermore, there is a *proof* of this fact; indeed, the fact was discovered by carrying out the proof. Note also, that since constraint goals are monotone, their negations are persistent: obviously if X can not be larger (or equal) to Y, no additional constraints are going to change that.

The double negation,

```
not(not( { C } ))
```

indicates that the constraint C is *possibly* consistent, but does not actually impose the constraint. Success here does not actually guarantee consistency, because the underlying mechanism is incomplete.

Typed Variables and Freeze

Freeze and the freeze-based constructs, which also use '{}' in their syntax (e.g. { nonvar(V)->P}), can be used with typed variables if desired. Since continuous (real) variables are seldom instantiated, however, they are mostly useful only with integer and boolean variables. The interval constraint propagation is finished before the woken goals are executed. Later there will be an example which makes use of this facility.

The unusual construct

```
{{ C }}
```

has the effect of postponing the constraint goal {C} (using freeze) until C is ground. Although it may occasionally be useful for handling disequality constraints on integers, usually it is a programming error. (It can happen when constraint goals are being passed as arguments and there is confusion about whether they already have the braces attached.)

Boolean Constraints I

Boolean variables and operators

Boolean variables can take only the values 0 and 1; they can be introduced explicitly with declarations of the form

```
X:boolean
```

where X is a variables (or list of variables). In most cases, explicit declarations are unnecessary, since the first use of a variable with exclusively boolean operations will force it to be a boolean. The operations supported are the prefix operator ~ (boolean complement) and the infix operators `and`, `or`, `nand`, `nor`, `xor`, `->`. For example, the constraint expression {1==A->B} constrains the usual boolean conditional "A implies B" to be true. There are also the relations '=<' (representing the boolean implication relation) and '==' (representing boolean equivalence when used a s a relation, and biconditional when used as an operator).

Boolean problems almost always require explicit enumeration, which can be done using

```
enumerate( X )
```

where X is a boolean variable or a list of boolean variables. For example, to display the truth table for the primitive `xor`, one can use:

```
?- {C== A xor B}, enumerate([A,B,C]).
     ?- [{0 == 0 xor 0}, enumerate([0, 0, 0])].
     ?- [{1 == 0 xor 1}, enumerate([0, 1, 1])].
     ?- [{1 == 1 xor 0}, enumerate([1, 0, 1])].
     ?- [{0 == 1 xor 1}, enumerate([1, 1, 0])].
```

Similarly:

```
?- {D== (A xor B) nand (B or C)}, enumerate([A,B,C,D]).
     ?- [{1 == (0 xor 0) nand (0 or 0)}, enumerate([0, 0, 0, 1])].
     ?- [{1 == (0 xor 0) nand (0 or 1)}, enumerate([0, 0, 1, 1])].
     ?- [{0 == (0 xor 1) nand (1 or 0)}, enumerate([0, 1, 0, 0])].
     ?- [{0 == (0 xor 1) nand (1 or 1)}, enumerate([0, 1, 1, 0])].
     ?- [{1 == (1 xor 0) nand (0 or 0)}, enumerate([1, 0, 0, 1])].
     ?- [{0 == (1 xor 0) nand (0 or 1)}, enumerate([1, 0, 1, 0])].
     ?- [{1 == (1 xor 1) nand (1 or 0)}, enumerate([1, 1, 0, 1])].
     ?- [{1 == (1 xor 1) nand (1 or 1)}, enumerate([1, 1, 1, 1])].
YES
```

Note that the enumeration is done in the order that the variables appear in the list given to `enumerate`.

Boolean satisfiability is the paradigm NP-complete problem: in the worst case one may need to explore 2**N branches if there are N boolean variables to enumerate. The effect of constraints is to reduce this to 2**M where M<N, at the cost of value propagation in the constraint system. But every constraint that forces a variable (thereby avoiding one choice) *halves* the overall cost.

The general pattern for dealing with highly combinatoric problems using constraints is:

(1) Set up the data structures and the declarations for the principle variables. Any data values or information determining the size and structure of the problem which must be read from files is done at this time. Specifications expressed as Prolog facts usually need to be converted into lists (using e.g. `findall` or `findset`) before being processed into constraint goals.

(2) Set up all the constraints. If the Prolog preparation in step 1 has been done well, the conversion to constraints should be relatively easy and transparently clear. This is important because it can be difficult to discover errors or omissions in the constraints. This part should be strictly deterministic; any non-determinism should be postponed until step 3 (the "Aurora principle"). When writing constraint goals it usually helps to imagine that all variables are instantiated, and you are just testing to see if you have a solution. Test by actually giving it a solution to the problem: all the constraints should of course be satisfied.

(3) Then proceed to the enumeration of constrained variables, or other non-deterministic bits. The bulk of the execution time will normally be spent in this section on difficult combinatorial problems, so it should generally not be doing anything other than enumeration. Step 1 should have buit data structures so that it is easy to extract the enumeration variables. The built-in enumeration predicates have been optimized in various ways and will generally be better (and easier to use and document) than what you might casually write, so use them. Think hard about different enumeration orders, since the proper choice can make a significant difference in performance; try several different strategies and compare the results. Good heuristic orderings can often be accomplished easily by proper use of sorts in step 1.

(4) Once the solution is obtained, there is usually some code required to capture the answer in a suitable form, e.g. in state space or an external file. This part will normally be omitted in the examples in this text.

Digital Logic

As an example of the use of boolean constraints, consider a 1-bit adder expressed as:

```
add1( X, Y, Cin, Z, Cout):-
      { Z  is Cin xor (X xor Y)},
      { Cout is (X and Cin) or (Y and Cin) or (X and Y)  }.
```

A 4-bit adder can then be synthesized as:

```
add4(  [X3,X2,X1],[Y3,Y2,Y1],Cin,   [Z3,Z2,Z1],   Cout):-
      add1( X1,  Y1,  Cin,  Z1,  C1),
      add1( X2,  Y2,  C1,   Z2,  C2),
      add1( X3,  Y3,  C2,   Z3,  Cout).
```

A general adder can be defined by:

```
adder( [], [], C, [], C).
adder(  [X|Xs],[Y|Ys],Cin,   [Z|Zs],Carry):-
             add1(  X,Y,C,  Z,Carry),
             adder(Xs,Ys,Cin,  Zs,  C).
```

Functional components can be tested for equivalence by comparing outputs. For example, if we have defined

```
add1_alt( X, Y, CI, Z, CO):-
      { Z  is CI xor (X xor Y)},
      { CO is (CI and (X or Y)) or (X and Y)}.
```

then we can look for any differences:

```
?-  add1(X,Y,CI,Z,CO),
     add1_alt(X,Y,CI,Z2,CO2), % same inputs, different outputs
    {1== ((Z xor Z2) or (CO xor CO2))}, % compare outputs
     enumerate([X,Y,CI]).

NO
```

with no output differences implying equivalence.

Similarly, we can determine that some function, say f([X1,X2,.X3,X4], Z) depends on a variable e.g. X1 by

```
?-  f([0,X2,X3,X4],Z1),  f([1,X2,X3,X4],Z2),  {1==(Z1  xor  Z2) },
     enumerate([X2,X3,X4]).
```

Functional dependencies

Functional dependencies is a formalized method used to analyze "keys" in databases among other things. A relation in the sense of relational databases is a finite set of tuples, with the fields of each tuple named by a distinct attribute (from a set **A** of attributes) and having values in some domain. A key is a field or set of fields whose values uniquely determine a tuple, and hence the values of the rest of the attributes. This can be expressed succinctly as P=>Q where P and Q are subsets of **A**. Given a set of functional dependencies, one of the things one might want to do is to determine if some other functional dependency is a consequence of them. The first step in answering such questions is to compute the closure operator of the set of functional dependencies: for each subset W of **A** the closure of W is the set of all attributes determined by W. Once we have the closure operator, then W -> U for each U in closure(W) and as W ranges over the powerset of A these are all the consequences of the original set of dependencies.

The following example of a set of functional dependencies (for an airline scheduling system) was given in Chapter 8 of the BNR Prolog User Guide:

[a]	-> [b, e, f, g]
[a, c, d, i]	-> [h]
[c, d]	-> [j]
[c, d, f]	-> [k]
[b]	-> [g]
[c, f]	-> [a, b, e]
[a, c]	-> [d]
[a, d]	-> [c]
[e, g]	-> [b]

Also given there was a conventional Prolog implementation of the closure operation as well as a freeze-based implementation. You should study those examples first. The boolean constraint version given here works a lot like the freeze implementation, but is much faster and more flexible. For a constraint implementation we map attributes to boolean variables, although we will in fact only be using the instantiated value 1. Each dependency is then translated using "and" on the left, for *each* value on the right; here we translate the implication as a relation using =<, but the conditional operator ->

could also be used. Then the above example can be coded very explicitly as:

```
closed([A,B,C,D,E,F,G,H,I,J,K]):-   [A,B,C,D,E,F,G,H,I,J,K]:boolean,
      { A =< B, A =< E, A =< F, A =< G,
        (A and C and D and I) =< H,
        (C and D and F) =< K,
        (C and F) =< A, (C and F) =< B, (C and F) =< E,
        (A and C) =< D,
        (A and D) =< C,
        (E and G) =< B}.
```

After executing closed(X), whatever 1's we put into X will have their consequences set as well. So to compute closures (in terms of attribute names say) we will need to convert a list of names to a list of 1's and variables (a membership vector denoted), and then use the resulting closed membership vector to select a list of attribute names. (Note that these two routines are almost identical; they differ because one is written to interact with the constraint system and the other must not interact.)):

```
% epsilon( OrdList1, OrdList2, Blist*)
epsilon( [], _, []).
epsilon(  [N|Ns],[N|As],[1|Bs]):-  !,epsilon(  Ns,As,Bs).
epsilon( [N|Ns],As,       [B|Bs]):-    epsilon( Ns,As,Bs).

% select(   OrdList2, Blist,OrdList1*)
select( [], [], []).
select(  [N|As],[B|Bs],[N|Ns]):-  B@=1,!,  select(  Ns,As,Bs).
select(  [A|As],[B|Bs],[N|Ns]):-  select(  [N|Ns],As,Bs).

member(X,[X|_]).
member(X,[_|Xs]):-  member(X,Xs).

list_closures( List):-% code  to compute closures
      attribute_names(  Master),
      closed(  X),
      foreach(   member(N,List)  do
                  [  sort(N,NS),
                      epsilon(Master,NS,X),
                     select(Master,   X,C),
                     nl, write( NS,' -> ',C)
                  ]).
```

For the example given above:

```
?- list_closures([[a, c, d], [b, e], [g, d, f], [a]]).

[a,c,d]  ->  [a,b,c,d,e,f,g,k]
[b,e]  ->  [a,b,c,d,e,f,g,k]
[d,f,g]  ->  [a,b,c,d,e,f,g,k]
[a]  ->  [a,b,c,d,e,f,g,k]
```

Exercise: Suppose that the FD's are defined by a predicate of the form fd(lhs->rhs) and write a Prolog translation routine to construct the equivalent closure predicate and the sorted list of all symbolic names, and adapt list_closures to use these.

Propositional calculus problems

Propositional calculus problems for testing boolean satisfaction algorithms are often expressed in conjunctive normal form, that is as a conjunction (or list) of disjunctions of p's and ~p's. This form is the most convenient for some algorithms, but it is sometimes unnatural and often much larger than other representations. The CLP(BNR) algorithms of course do not require any such restricted form of input.

One of the standard benchmarks used for boolean satisfiability testing is the "pigeon-hole(M,N)" problems: placing M pigeons into M holes with every pigeon in one hole and no two pigeons in the same hole. When M is N +1, this is of course impossible, but a propositional calculus proof of this fact is a "worst case" sort of problem.

To generate a conjunctive normal form representation, we use the subroutines:

```
place_pigeon(N,Holes):- % holes will be list of booleans
      length(N,Holes),  Holes:boolean,
      or_reduce( Holes, B), { 1==B}, % pigeon has hole
      at_most_one( Holes).

or_reduce([],0).
or_reduce([X|Xs],  X  or  S):-  or_reduce(Xs,S).

at_most_one( []). % each pigeon in just one hole
at_most_one(  [X|Xs]):-  not_both(Xs,X),  at_most_one(Xs).

not_both([],_).
not_both([X|Xs],Y):-  {  1==  ~X  or  ~Y},  not_both(X,Y).
```

Then pigeonhole predicates can be written as:

```
pigeons(M,N):-  $pigeons(M,N,Hs),
            holes_used_once(Hs).

$pigeons(0,N,[]).
$pigeons(M,N,[H|Hs]):- M>0, M1 is M - 1,
      place_pigeon(N,  H),
      $pigeons(M1,  N).
```

```
holes_used_once(  [[]|_]):-!.
holes_used_once(List_of_lists):-
      column(  List_of_lists,  First_column,  Rest),
      at_most_once(  First_column),
      holes_used_once(Rest).

column( [], [], []).
column( [  [X|Xs]|Ys],  [X|Cs],  [Xs|Rs]):-  column(Ys,Cs,Rs).
```

Using

```
enum_list([]).
enum_list([X|Xs]):-  enumerate(X),  enum_list(Xs).
```

for enumeration, one can test this by frist giving a satisfiable problem:

```
?-  pigeons(3,3,H),  enum_list(H).
    ?- [pigeons(3, 3, [[0, 0, 1], [0, 1, 0], [1, 0, 0]]),
         enum_list([[0, 0, 1], [0, 1, 0], [1, 0, 0]])].
    ?- [pigeons(3, 3, [[0, 0, 1], [1, 0, 0], [0, 1, 0]]),
         enum_list([[0, 0, 1], [1, 0, 0], [0, 1, 0]])].
    ?- [pigeons(3, 3, [[0, 1, 0], [0, 0, 1], [1, 0, 0]]),
         enum_list([[0, 1, 0], [0, 0, 1], [1, 0, 0]])].
    ?- [pigeons(3, 3, [[0, 1, 0], [1, 0, 0], [0, 0, 1]]),
         enum_list([[0, 1, 0], [1, 0, 0], [0, 0, 1]])].
    ?- [pigeons(3, 3, [[1, 0, 0], [0, 0, 1], [0, 1, 0]]),
         enum_list([[1, 0, 0], [0, 0, 1], [0, 1, 0]])].
    ?- [pigeons(3, 3, [[1, 0, 0], [0, 1, 0], [0, 0, 1]]),
         enum_list([[1, 0, 0], [0, 1, 0], [0, 0, 1]])].
```

and then a non-satisfiable problem:

```
?-  pigeons(4,3,H),  enum_list(H).

NO
```

To measure the time for a problem, one can first use `stats/0` to clear the statistics counters, then run a (successful) predicate, and `stats(A,B,C,D,T)` to get the statistics. Here A represents the number of Prolog logical inferences, B the number of primiitve (non-Prolog) calls, C the number of primitive narrowing operations in the constraint system, D is the number of separate invocations of the constraint system, and T is the time (in ms) elapsed since the `stats()` call. For example:

```
?-  stats,  pigeons(4,3,H),  enum_list(H);true,  stats(A,B,C,D,T).
```

The pigeon(M,M) problem is equivalent to finding all the permutations on M objects. Note that there M*M booleans, so for

M=10 we have a raw search space of 2**100 or about 10**30. The 1020 constraints effectively reduce this to 10!, or about 10**6.

Exercise: Compare with a version using inverted (~) booleans.

Structural analysis of Petri nets

Petri nets are widely used to model discrete control systems, transaction systems, and communications protocols. A Petri net consists of a *net* together with a *marking* which represents the state of the net. The net is a bipartite graph consisting of *places* and *transitions* connected by directed arcs; a marking is a distribution of *tokens* over the places. Any transition may have input places (where the arc is directed from the place to the transition) and output places (arc from transition to place). A transition can fire if all of its input places have at least one token; firing a transaction removes a token from each input and puts a token into each output place. In general the evolution of the state is non-deterministic as there may be many transitions that can fire for any given marking.

Structural analysis studies properties that depend only on the topology of the network independent of marking. Of particular importance are structural properties that determine behavioural possibilities. (For a special subclass of Petri nets, the "free choice Petri nets" which are characterized by the property that if two transitions share an input place, it is the unique input place of both, structural properties are particularly useful.) For example:

siphon - a non-empty subset of places such that every transition that outputs to it also inputs from it;

trap - a non-empty subset of places such that every transition that inputs from it also outputs to it.

pre-conservative component - a non-empty subset which is both a siphon and trap.

For a siphon, if none of its places is marked at some time, it will remain empty henceforth. For a trap, if it contains tokens intitially, will always contain tokens.

Let us suppose that a Petri net is specified by a predicate `places(List)` whose argument is a list of places, and `transition(Name,Inputs,Outputs)` specifying the transitions. We wish to find all the siphons in the network. For each siphon S and each transaction t w emust have:

$\text{outputs}(t) \cap S <> \varnothing \text{->} \text{inputs}(t) \cap S <> \varnothing$, inputs(t)

or equivalently

$\text{inputs}(t) \cap S = \varnothing \text{->} \text{outputs}(t) \cap S = \varnothing$.

For each place we create a boolean variable B interpreted as B=0 means that it is *in* the siphon S. Then for each `transition(N,I,O)` we can map I to its associated list of booleans IB and likewise for O to get I=OB, and translate the above condition by

conjunction(IB) =< conjunction(OB).

```
siphon(    List_of_names):-
      places( Places),
      map_table( Places, Map, Bs),
      findall(    trans(I,O), transition(_,I,O), Tlist),
      map_transitions( Tlist, Map),
      enumerate(Bs),
      selectfrom(Map, List_of_names).

% build 'symbol table' for mapping names & export boolean vector
map_table([],[],[]).
map_table( [P|Ps], [ [P,B]|Ms], [B|bs]):-B:boolean,
      map_table(Ps,Ms,Bs).

map_transitions([],_).
map_transitions([ trans(I,O)|Ts], Map):-
      map_places(I,Map,IB), % IB is symbolic conjunctionof I
      map_places(O,Map,OB), % OB is symbolic conjunctionof O
      { IB =< OB },          % no inputs
        map_transitions( Ts, Map).

map_places( [],_,1). % why 1?
map_places( [P|Ps],Map, PB and B):- member([P,PB],Map),!,
      map_places( Ps,Map,B).

member(X,[X|_]).
member(X,[_|Xs]):- member(X,Xs).

% convert solution to list of place names
selectfrom( [], []).
selectfrom( [ [P,0]|Ms], [P|Ps]):- !, selectfrom(Ms,Ps).
selectfrom( [ [P,1]|Ms], Ps):- selectfrom( Ms,Ps).
```

With the following test data:

```
places( [a,b,c,d,e,f,g,h,i,j,k]).

transition( 1, [a,d], [c]).
transition( 2, [c], [b,d]).
transition( 3, [b], [a]  ).
transition( 4,  [b],  [d,e,h]).
transition( 5,  [e],  [f,i]).
transition( 6,  [f],  [g]   ).
transition( 7,  [g],  [e]   ).
transition(  8,   [i,j],[h,k]).
transition( 9,  [k],  [j]   ).
```

we get the following list of siphons:

```
?-  siphon(X).
    ?- siphon([a, b, c, d, e, f, g, h, i, j, k]).
    ?- siphon([a, b, c, d, e, f, g, h, i, k]).
    ?- siphon([a, b, c, d, e, f, g, h, i]).
    ?- siphon([a, b, c, d, e, f, g, h, j, k]).
    ?- siphon([a, b, c, d, e, f, g, i, j, k]).
    ?- siphon([a, b, c, d, e, f, g, i, k]).
    ?- siphon([a, b, c, d, e, f, g, i]).
    ?- siphon([a, b, c, d, e, f, g, j, k]).
    ?- siphon([a, b, c, d, e, f, g]).
    ?- siphon([a, b, c, d, h, j, k]).
    ?- siphon([a, b, c, d, j, k]).
    ?- siphon([a, b, c, d]).
    ?- siphon([a, b, c, e, f, g, h, i, j, k]).
    ?- siphon([a, b, c, e, f, g, h, i, k]).
    ?- siphon([a, b, c, e, f, g, h, i]).
    ?- siphon([a, b, c, e, f, g, h, j, k]).
    ?- siphon([a, b, c, e, f, g, i, j, k]).
    ?- siphon([a, b, c, e, f, g, i, k]).
    ?- siphon([a, b, c, e, f, g, i]).
    ?- siphon([a, b, c, e, f, g, j, k]).
    ?- siphon([a, b, c, e, f, g]).
    ?- siphon([a, b, c, h, j, k]).
    ?- siphon([a, b, c, j, k]).
    ?- siphon([a, b, c]).
    ?- siphon([b, c, d, e, f, g, h, i, j, k]).
    ?- siphon([b, c, d, e, f, g, h, i, k]).
    ?- siphon([b, c, d, e, f, g, h, i]).
    ?- siphon([b, c, d, e, f, g, h, j, k]).
    ?- siphon([b, c, d, e, f, g, i, j, k]).
    ?- siphon([b, c, d, e, f, g, i, k]).
    ?- siphon([b, c, d, e, f, g, i]).
    ?- siphon([b, c, d, e, f, g, j, k]).
    ?- siphon([b, c, d, e, f, g]).
    ?- siphon([b, c, d, h, j, k]).
    ?- siphon([b, c, d, j, k]).
    ?- siphon([b, c, d]).
    ?- siphon([j, k]).
    ?- siphon([]).
YES
```

Exercise: This version tends to produce the large siphons first. Why?

Exercise: What condition has been omitted; how could you fix it?

Exercise: What is the easiest change(s) to make in order to compute traps instead of siphons?

Boolean Constraints II

Relational algebra and transitive closure problems

A binary relation on X is an element of the powerset of XxX, and are hence partially ordered by set inclusion, and have intersection, union, and complement defined. The bottom relation is 0, the top one is U (the universal relation), and the diagonal relation is denoted by 1. In addition, there are operations of relational product:

$x(RS)y \leftrightarrow Ez\ xRz \,\&\, zSy$

(for which 1 is an identity and 0 is a zero)
and adjoint (converse/reverse/inverse)

$x(R^T)y \leftrightarrow yRx.$

A relation R is reflexive iff $R \supset 1$, symmetric iff $R^T = R$, transitive iff RR=R (idempotent).

The reflexive relations form a sublattice. For a reflexive relation R, one is often interested in its transitive closure R*, defined as the smallest transitive R relation bigger than R, which can (in the finite case) be computed by taking powers of R until they stabilize, that is finding the least fixed point of the equation X=RX. For a non-symmetric R, one unions it with 1 before taking the transitive closure.

Let us represent a finite relation by a list of N lists of N booleans. To generate a generic relation we will use:

```
relation(N,R):-   N:integer(0,_),   $rel_row(N,N,R).

$rel_row(0,_,[]).
$rel_row(M,N,[Bs|Rs]):-  {M>=1,  M1  is  M  -  1},
             length(N,Bs),   Bs:boolean,
             $rel_row(M1,N,Rs).
```

The `length` predicate was defined in a previous section; `$rel_row` uses the same technique. A specialized print utility which produces a compact output is useful:

```
pr_rel([]):-nl.
pr_rel([X|Xs]):-   nl,   $pr_bool_list(X),   pr_rel(Xs).

$pr_bool_list([]).
$pr_bool_list([B|Bs]):-   $map_to_ch(B,C),   write(C),   !,$pr_bool_list(Bs).
```

```
$map_to_ch(B,'_'):-var(B).
$map_to_ch(0,'0').
$map_to_ch(1,'1').
```

The partial order on relations can be implemented as:

```
ge( [], []).
ge(  [X|Xs],[Y|Ys]):-  $ge(X,Y),  ge(Xs,Ys).
$ge([],[]).
$ge([X|Xs],[Y|Ys]):-  {X>=Y},  $ge(Xs,Ys).
```

To access the entries we can use an indexing routine (we are thinking of a relation as a column of rows):

```
entry(R,I,J,B):-  arg(R,I,Row),  arg(Row,J,B).
```

To test for or make reflexive relations we can use:

```
reflexive(  R  ):-  $diagonal(R,1).
$diagonal(  [],  _).
$diagonal(  [R|Rs],N):- N1 is N + 1,
                arg(R,N,1),
                $diagonal(Rs,N1).
```

A useful utility is one to transpose such an array:

```
transpose(R,  RT)  :-  $trans(R,RT).

$trans([[]|_],[]).
$trans(R,  [C|Cs]):-  $peel_column(R,C,Rest),  $trans(Rest,Cs).

$peel_column([],  [],  []).
$peel_column([  [C|Rs]|Rest],  [C|Cs],  [Rs|RRs]):-
                $peel_common(  Rest,  Cs,  RRs).
```

Then one way to implement a predicate for symmetric relations is:

```
symmetric(R):-  transpose(R,R).
```

Note that this can be applied to an array of untyped variables (and constants), and it is more effcient to do so.

The relational product can be written in terms of its action on a subset represented as a membership vector.

```
map_rel([[]|_],  B,  []):-!.
map_rel(  R,  B,  [C|Cs]):-
      $peel_column(R,Col,Rest),
      $or_and(B,Col,Dot),!,
      {C == Dot},
      map_rel(Rest,B,Cs).
```

```
$or_and([],[],0).          % boolean inner product
$or_and([X|Xs],[Y|Ys],(X and Y) or C1):- $or_and(Xs,Ys,C1).
```

Now we can easily express the relational product:

```
relation_product([],   _,   []):-!.
relation_product([A|As],B,[C|Cs])  :-
            map_rel(B,A,C),
             relation_product(As,B,Cs).
```

Using this we can express the constraint for a relation to be transitive as:

```
transitive(R):-  relation_product(R,R,R).
```

and transitive closure of a relation S by:

```
transitive_closure(S,R):-  reflexive(S),
          relation(N,S),  relation(N,R),
          ge(R,S),
          relation_product(R,S,R).
```

When value in S have (or take) the value 1, the appropriate values in R will become 1. Conversely, if values in R become 0 they may propagate to S; when S is symmetric this is a version of a coloring problem. Thus when S is given by

```
1___10__11
_1_1___1__
0_1___1100
_1_1____00
__0_1_____
_____1____
_0____1___
_______1__
___1____1_
______1__1
```

R initially (because of call to ge) becomes:

```
1___1___11
_1_1___1__
__1___11__
_1_1______
____1_____
_____1____
______1___
_______1__
___1____1_
______1__1
```

whence the transitivity condition produces R as:

```
11_11_1111
_1_1___1__
__1___11__
_1_1___1__
____1_____
_____1____
______1___
_______1__
_1_1___11_
______1__1
```

Note that one can set up the equations before supplying the initial values for S, as in the query:

```
?-relation(10,R),transitive_closure(R,S),nl,test10(R),   pr_rel(S).
```

Also note that the forward propagation of 1's in transitive closure problems is deterministic, while the backward propagation of 0's is very non-deterministic (as in coloring problems).

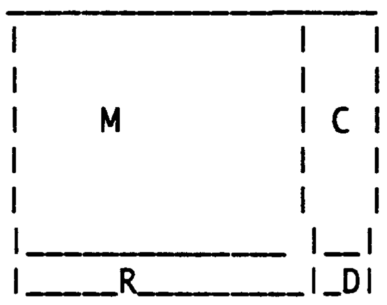
Because the number of booleans in a relation is proportional to N*N, general problems which require finding solutions to equations in relational algebra lead very quickly to difficult problems (in terms of time and space bounds), unless the constraints (including boundary values) are very restrictive. However, small systems of relational algebra equations can be solved quite nicely.

Exercise (suitable for a project): Develop a predicate `rel_algebra(N,{Eqs..})` for solving systems of relational algebra equations supporting union, intersection, complement, transpose, order, relational product and transitive closure operations.

NOTE: For maximum efficiency, symmetry should be imposed at the level of Prolog variables before typing is done, and the presence of reflexivity, symmetry, and idempotence should be propagated symbolically as much as possible before setting up the constraint equations.

Exercise: Formulate and solve some interesting problems in relational algebra using the above technique. Many branches of computer science (eg. parsing technology and finite state automata theory make use of such equations).

The representation of arrays as a list of lists is quite natural, but makes transpose (and hence product) fairly expensive. Also, this representation does not make it easy to relate solutions of order N to those of order N + 1. An alternate representation which is better in these respects treats an order N relation as an order N-1 relation which has been "bordered" by adding a row and column:



This can be represented neatly as a recursive structure of the form:

```
relation( D, C, R, M )
```

where D is a single element, C and R are lists of length N-1, and M is a matrix of order N-1. With this formulation, for example, we can write the reflexive predicate as:

```
reflexive([]).
reflexive(  relation(1,_,_,M)):-   reflexive(M).
```

and transpose as simply:

```
transpose([],[]).
transpose(relation(A,C,R,M),   relation(A,R,C,M)):-   transpose(M).
```

Exercise: Implement relational product in this representation and compare the efficiency of the two implementations.

Note that this only affects the cost of building constraints; the constraints and hence propagation are the same. The big advantage of this representation arises in problems in which one has a solution to the equations defined on a subset and wishes to extend it to a larger set.

Exercise: formulate a large non-deterministic coloring problem as a series of problems for increasing N, and use ! after each stage is completed.

Integer Constraints

Constraints on integer variables work much like those on boolean variables. However, since the operations on integers are the same as those on reals, it is usually necessary to provide more explicit declarations with integer variables. Abstract finite domains must first be mapped into finite integer ranges in order to formulate constraints.

Enumeration of integer variables can be also done with `enumerate`, which enumerates the variables (from lower bound to upper bound) in the order specified. In addition, the `firstfail` predicate (with syntax analogous to `enumerate`) enumerates integer ranges in order of increasing size of domain. This enumeration strategy is in most cases a good one, and usually much better than a random choice of enumeration order, but has higher overheads than `enumerate`. Both `firstfail` and `enumerate` will enumerate any boolean variables first, if the list of variables is of mixed type.

Crypto-Arithmetic

A simple example of the use of constraints on integer variables is the well-known "SEND MORE MONEY" puzzle: to determine the digits corresponding to the letters in the sum

```
          SEND
         +MORE
    =    MONEY.
```

This can be formulated as follows:

```
sendmoremoney([S,E,N,D,M,O,R,Y]):-
      [S,M]:integer(1,9),
      [E,N,D,O,R,Y]:  integer(0,9),
      distinct(  [S,E,N,D,M,O,R,Y]),
      {1000*(S  +M)+  100*(E  +O)+  10*(N  +R)  +  (D  +E)
         ==  10000*M  +  1000*O  +  100*N  +  10*E  +  Y}.

distinct([]).
distinct([X|Xs]):-   $distinct(Xs,X),   distinct(Xs).
$distinct([],_).
$distinct([X|Xs],Y):-   {X<>Y},   $distinct(Xs,Y).
```

By executing without enumeration we can verify that the setup is deterministic, and also see what the initial narrowings are. If the 'YES' response and new prompt '?-' are printed, the call was deterministic; if these are not printed, there was a choicepoint created: entering <CR> will proceed to the next solution, and if the prompt is then returned it indicates that the choicepoint was only "virtual". Another way is to use the count predicate in Base, which counts the number of solutions:

```
?-   count(  [sendmoremoney(L)],N).
       ?-  count([sendmoremoney(_H155)],  1).
YES
```

If there is more than one solution to the setup, the simplest way to fix the problem is to put a ! at the end of the main predicate. In the output, lookout for effectively unbounded ranges, which would suggest that some constraints needed to make the problem well-posed are missing.

```
?-   sendmoremoney(_).
      ?- sendmoremoney([9, _H3461, _H3462, _H3463, 1, 0, _H3466, _H3467])
         where [_H3461 : integer(2, 8),
                _H3462 : integer(2, 8),
                _H3463 : integer(2, 8),
                _H3466 : integer(2, 8),
                _H3467 : integer(2, 8)].

YES
```

Then:
```
?-   sendmoremoney(L),   enumerate(L).
```
or
```
?-   sendmoremoney(L),   firstfail(L).
```
gives:

```
?-   sendmoremoney(L),   enumerate(L).
      ?- [sendmoremoney([9, 5, 6, 7, 1, 0, 8, 2]),
           enumerate([9, 5, 6, 7, 1, 0, 8, 2])].
YES
```

One can easily count the number of solutions to a problem:

```
?-   count([sendmoremoney(_H160),   enumerate(_H160)],   1).
        ?-  count([sendmoremoney(_H159),  enumerate(_H159)],  1).
```

It is often of interest to be able to determine just how many backtrackings are done in such problems. For this, enumerate and firstfail can take an optional argument which is executed just after each backtrack:

```
?-   sendmoremoney(L),   enumerate(L,   write(*)).
***       ?- [sendmoremoney([9, 5, 6, 7, 1, 0, 8, 2]),
             enumerate([9, 5, 6, 7, 1, 0, 8, 2], write("*"))].
<CR>
**
YES
```

For large searches, there is a predicate in the constaint utilities file which is similar to count. To find the number of backtracks before the first solution use:

```
?-  backtracks( C,  [sendmoremoney(L),  enumerate(L,  C)],  N).
```

while to get the total number of backtracks:

```
?-  backtracks( C,  [sendmoremoney(L),  enumerate(L,  C),fail],  N).
```

Exercise: Introduce explicit carry variables ([0,1]) and replace the single constraint with separate sums for each digit.

Eight Queens

A popular early example of constraint programming was the well-known "eight queens" problem and its generalization. The naively declarative generate-and-test Prolog version of eight queens looks like:

```
%
%                   Prolog   generate-and-test   solution
%
eight_queens([X1,X2,X3,X4,X5,X6,X7,X8]):-
      permutation(     [X1,X2,X3,X4,X5,X6,X7,X8],[1,2,3,4,5,6,7,8]),
      safe(   [X1,X2,X3,X4,X5,X6,X7,X8]).

permutation([],[]).
permutation([X|Xs],Ls):-
      delete(X,Ls,Rs),
      permutation(Xs,Rs).

delete(X,   [X|Xs],   Xs).
delete(X,   [Y|Ys],   [Y|Rs]):-   delete(X,Ys,Rs).

safe([]).
safe([X|Xs]):-   noattack(Xs,X,1),   safe(Xs).

noattack([],Y,N).
noattack([X|Xs],Y,N):-  N1  is  N  +1,
      Y<>X  -  N,
      Y<>X  +  N,
      noattack(  Xs,Y,N1).
```

A pure constraint version (generalized to N-queens), and even more declarative, is:

```
%
%  one (of many) versions of queens using constraints
%
queens(N,List):-  length(List,N),  List:integer(1,N),
      $queens(N,List).

$queens(0,[]):-!.
$queens(N,[X|Xs]):-  N1  is  N  -1,
      c_noattack(  Xs,  X,1),
      $queens(N1,Xs).

c_noattack([],Y,N).
c_noattack([X|Xs],Y,N):-  N1  is  N  +1,
      {Y<>X,  Y<>X  -  N,  Y<>X  +  N},
      c_noattack(  Xs,Y,N1).

?-  queens(5,L),  enumerate(L),  nl,  write(L),  fail.

[1,3,5,2,4]
[1,4,2,5,3]
[2,4,1,3,5]
[2,5,3,1,4]
[3,1,4,2,5]
[3,5,2,4,1]
[4,1,3,5,2]
[4,2,5,3,1]
[5,2,4,1,3]
[5,3,1,4,2]
```

The constraint version is also considerably better than the traditional one:

```
8 queens(1st sol):     gen&test        enumerate        firstfail
time(sec/25Mhz-68020)   4.267          0.717             0.6
#backtracks             ?              24                15
```

The primitive disequality constraint '<>' tends to occur often in puzzle problems over finite domains and bounded integers. The CHIP technology is optimized to handle such constraints efficiently. Such constraints are much less efficently handled in interval-based systems, and are almost useless in large or continuous domains; in CLP(BNR) they are in fact restricted to use on integer (or boolean) values.

Exercise: swap the arguments to permutation in the first version above and remeasure the performance.

Exercise: write a hybrid version that retains the use of permutation, but uses constraints for the diagonals.

Exercise: write a pure boolean constraint version of eight queens.

Exercise: the disequality constraint should be much weaker in CLP(BNR) than in CHIP, yet the number of backtracks recorded for the firstfail version of the constraint program above is 15 versus 23 reported for CHIP in Van Hentenryck's book Constraint satisfaction in Logic Programming: investigate this discrepancy.

Cardinality and Pseudo-Boolean Constraints

One of the most important uses of integer constraints is to count boolean variables, the so-called cardinality operator introduced by Pascal van Hentenryck (1991). Since CLP(BNR) takes boolean values as 0 and 1, which can be regarded also as integers, it is possible to explicitly sum booleans to get an integer result. This allows us to define a cardinality operator easily:

```
cardinality( Bs, L, U):- sum(Bs,Sum), { S is Sum, L=<S,S=<U}.
```

For example, with N= length of Bs, using the built-in cardinality it is easy to express a number of useful conditions:

```
cardinality(Bs, 1, 1)     % exactly one B is true
cardinality(Bs, M, M)     % exactly M B's are true
cardinality(Bs, M, N)    % at least  M B's are true
cardinality(Bs, 0, M)    % at most M B's are true
cardinality(Bs, N, N)    % all B's are true .
```

For example, using the first of these allows us to reformulate the pigeonhole problem much more succinctly:

```
place_pigeon(N,Holes):- % holes will be list of booleans
      length(N,Holes),  Holes:boolean,
      cardinality( Holes, 1,1). % pigeon has exactly one hole
```

Then pigeonhole predicates can be written as:

```
pigeons(M,N):-  $pigeons(M,N,Hs),
            holes_used_once(Hs).

$pigeons(0,N,[]).
$pigeons(M,N,[H|Hs]):- M>0, M1 is M - 1,
      place_pigeon(N,  H),
      $pigeons(M1,  N).

holes_used_once(  [[]|_]):-!.
holes_used_once(List_of_lists):-
      column( List_of_lists,  First_column,  Rest),
      cardinality(  First_column,1,1),
        holes_used_once(Rest).

column( [], [], []).
column( [ [X|Xs]|Ys], [X|Cs], [Xs|Rs]):-  column(Ys,Cs,Rs).
```

Exercise: compare the performance (space and time) of this with the previous version.

Comparison Operators

The comparison relations {==,=<,>=} and their negations {<>,>,<}, which usually appear as constraint *relations*, can also be used as boolean valued *operations* of non-boolean arguments in CLP(BNR). For example (and note the *required* parenthesization):

```
            {B == ( X>=Y)}
```

with [X,Y]:integer, and B:boolean, has the following effects:
- if B is 1 then it is equivalent to the relation {X>=Y}
- if B is 0 then it is equivalent to the relation {X<Y}
- if B is indeterminate and X>=Y is *necessarily* true, B becomes 1
- if B is indeterminate and X>=Y is *necessarily* false, B becomes 0.

The arithmetic operation (X>=Y) is necessarily true(resp. false) as soon as the range of possible values for X are disjoint from those of Y and strictly to the right (resp. left) of those for Y. These operations permit a much wider range of interactions between boolean and non-boolean variables to be formulated than would otherwise be possible, In particular, combinatorial aspects of a problem can be formulated alongside the non-combinatorial aspects with propagation going both ways.

For example, one of the common uses of these operators is to express that a variable must be chosen from a finite set of values:

```
element(Y,  Xs,  Bs):-  $element(Xs,Y,Bs),  $sum(Bs,S),  {S==1}.
$element( [],  _,  []).
$element( [X|Xs],  Y,  [B|Bs]):-  {B  is  (X==Y)},  $element(Xs,T,Bs).

$sum([],  0).
$sum([B|Bs],  B  +  N):-  $sum(Bs,N).


?-  Y:integer,  element( Y,  [20,30,50],  Bs),  enumerate(Bs).

?-  Y:integer,  element( Y,  [20,30,50],  Bs),{Y>=40}.
```

As an extension of this, we can easily write a predicate which chooses M distinct values from a list of N:

```
choice( Choices,  List,   BoolArray):-
```

```
      length(Choices,M),   length(List,N),   {M=<N},
      $choice(  Choices,List,   BoolArray),
      distinct(   Choices).            % see above for distinct

$choice( [], L, []).
$choice( [C|Cs], L, [B|Bls]):- element( C,L, Bs),
      $choice(Cs,L,Bls).
```

Another useful utility counts occurrences in a list:

```
occurrences(X,List,N):-   N:integer(0,_),
      $element(List,X,Bs),
      $sum(Bs,S), {N is S}.
```

It may be useful sometimes to export the Bs as well, for enumeration purposes.

All these utilities, plus a few others, can be found in the file named *constraint_utilities*. From now on we will make use of these utilities without repeating the definitions.

Magic Series

The magic series problem (Van Hentenryck, 1989) demonstrates the use of the comparison operations to formulate a tricky problem and also provides an example illustrating the effect of adding well-chosen redundant constraints.

The magic series problem of order N can be thought of as finding a sequence [M0,M1, ... MN] of integers which make the following self-referential text T true:

" T contains M0 occurrences of 0 and
T contains M1 occurrences of 1 and
T contains M2 occurrences of 2 and
...
T contains MN occurrences of N "

The M's are obviously one more than the number of occurrences KN of each N in the list of Ms, so we can formulate the problem succinctly in terms of these Ks as:

```
$initialization:-  load_context(  constraint_utilities).
% this will ensure that context constraint_utilities is loaded

magic_series(N,  Ks):-  length(Ks,N),  Ks:integer(0,_),
        $magic(Ks,0   ,Ks).

$magic([],N,_).
$magic([K|Ks],N,KS):-  N1  is  N  +1,
        occurrences(N,KS,   K),
        $magic(Ks,N1,KS).

?-  magic_series(  4,   Ks),  enumerate(Ks).
      ?-  [magic_series(4,  [1,  2,  1,  0]),  enumerate([1,  2,  1,  0])].
      ?-  [magic_series(4,  [2,  0,  2,  0]),  enumerate([2,  0,  2,  0])].
```

For N=10, the initial ranges for all ten variables will be [0,10]. (Why?) This suggests an initial search space estimated size of 11**10 possibilities. Yet there is but a single solution, and it only takes 27 backtracks to search the entire space. The statistics for the *enumeration alone* are:

```
lips=565 pips=129 ops=19069 its=54
```

This shows that the initial search space size is not always a good indicator of the difficulty of the problem, and this is one reason why empirical investigation is so important.

If one sums over the Ks, the result must be N. (Why?) We can therefore add this as a *redundant* constraint :

```
magic_series2(N,   Ks):-   length(Ks,N),   Ks:integer(0,_),
        sum( Ks,  N),
        $magic(Ks,0   ,Ks).
```

Since the added constraint is redundant, it does not change the set of solutions. However, the number of backtracks required has now dropped to 18, and the stats have become:

```
lips=438  pips=93  ops=5547  its=36
```

and the time is reduced to about a third of the original.

Similarly, we can reason that N also equals the sum of $i*K_i$. (This is another way of summing the K's, using their interpretation.) Adding this constraint as well, formulated as:

```
magic_series3(N,   Ks):-   length(Ks,N),   Ks:integer(0,_),
        sum( Ks,  N),
        $sum2(  Ks,0,   M),{M==N},
        $magic(Ks,0   ,Ks).

% summation of    j*K(j),  j=0,N-1
$sum2([],_,    0).
```

```
$sum2([K|Ks],N,  K*N+S):-  N1  is  N  +1,  $sum2(Ks,N1,S).
```

gives

```
lips=354  pips=69  ops=2427  its=24
```

and the number of backtracks has dropped to 12, and the time is now halved from the previous version. This is summarized in the aphorism: "the more constraints, the faster it goes."

Mastermind

Many people remember enjoying the puzzle game "Mastermind", which involves guessing a linear arrangement of 4 colored pegs, with colors chosen from the set {red,green,blue, yellow, brown,orange}. Guesses are based on information gained from the scoring of the previous guesses. The score consists of two numbers: the first ("bulls') indicating the number of pegs which have the right color in the right position, and the second number("cows') indicating whether the colors are right regardless of position. As a representation of the problem data we define:

```
% defines the space of possible answers
newguess(G):-G=[A,B,C,D],  G:integer(1,9),
            {A<>B,A<>C,A<>D,B<>C,B<>D,C<>D}.
```

The scoring can be computed most easily in terms of bulls and the sum of cows and bulls, as :

```
score(Answer,Guess,  [B,  C])  :-
            bulls( Guess,  Answer,  B),
            cowsbulls( Guess,  Answer,  C).

bulls( Xs,  Ys,  N ):-  $count_equal(Xs,Ys,C),  {N  ==  C}.
$count_equal( [],  [],  0).
$count_equal( [G|Gs],[A|As],    (G==A)  +  S):-
      $count_equal(Gs,As,S).

cowsbulls( Guess,  Answer,  C):-        % cows  +  bulls
      $cowsbulls(Guess,Answer,S),  {C  ==  S}.

$cowsbulls([],_,0).
$cowsbulls([X|Xs],Ys,N+J):-
      in(X,Ys,J),
      $cowsbulls(Xs,Ys,N).

in(X,Ys,N):-  N:integer(0,1),  $in(Ys,X,M),  {N  ==  M}.
```

```
$in([],_,  0).
$in([Y|Ys],X,   (Y==X)+S):-$in(Ys,X,S).
```

The strategy to be used is simple but effective (but not optimal): choose the first arrangement that reproduces all previous scores:

```
mastermind:-   mastermind_1([],[]).

mastermind_1( Gs,  Ss):- make_a_guess(Gs,Ss,G),!,  % take  first  choice
            get_score(G,  S),
            not(S=[4,_])->mastermind_1(  [G|Gs],  [S|Ss]).

make_a_guess(    Previous_Guesses,  Previous_Scores,  G  ):-
            newguess(G),
            matches_all(  Previous_Guesses,   Previous_Scores,G),
            enumerate(G).

matches_all([],[],_).
matches_all(  [G|Gs],  [S|Ss],  New):-
            score( G,  New,  S),
            matches_all( Gs,  Ss,  New).

get_score(  G,   S):-  answer(A),  score(G,A,S).
```

This gives:

```
?-  mastermind2.
guessing  :  [1,2,3,4]
guessing  :  [2,1,5,6]
guessing  :  [2,3,6,7]
guessing  :  [3,1,7,5]
guessing  :  [3,5,4,6]
```

In this version each guess sets up a new problem with one additional constraint; a better strategy would be to just add the new constraint to the current problem. This poses a problem: in order to generate a guess we need to find a (first) solution to the current problem; once we have narrowed the state to that solution, the new constraint will be inconsistent with it (unless the guess was correct), but backtracking across this failure will also remove the new constraint. So, it is necessary to make use of side-effects to remember the first solution (in the state space), get out of the search and back to the generic problem, and only then use the remembered solution as the next guess. In addition, the first guess generated by the above algorithm is not very helpful, and since it is arbitrary anyway, we will always use something using four colors, e.g. [red,yellow,blue,green]. This leads to somewhat different program:

```
mastermind2:-
     forget_all( mastermind(_)),   % clear  state  space
```

```
      newguess(Vars),       % setup variables of problem
      $mastermind([1,2,3,4],   Vars).

$mastermind(G,   Vars):-
      get_score( G,  S),
      next_guess(  G,S,Vars).

next_guess( Guess,  [4,_],  _     ):-!.  % finished
next_guess(  Guess,  Score,  Vars):-
      score(Guess,Vars,  Score),  %  add  new  constraint
      pick_guess(  Vars,  New),
      $mastermind(New,   Vars).

pick_guess( Vars,  _        ):-  % side-effect utility
      once(enumerate(  Vars  )),%  find  first  solution
      remember(  mastermind(  Vars)),  %  make  side-effect
      fail.          % escape from search & undo narrowing
pick_guess( _  ,    Guess) :- % recover the answer
      forget(   mastermind(Guess)).
```

with result:

```
?-   mastermind2.
guessing  :  [1,2,3,4]
guessing  :  [2,1,5,6]
guessing  :  [2,3,6,7]
guessing  :  [3,1,7,5]
guessing  :  [3,5,4,6]
```

Word Algebra

This is a follow-up on the possibilities of the length predicate discussed earlier. The problem is to write a meta-program that can take a list of equations in the word monoid (including the involutive operation of reversal) and solve them. For example, we would like to be able to solve (with & representing concatenate, ~ representing reverse, and # denoting length):

```
?-word_algebra( { #U =< 3, (~U) & [a] & W & U =(~W) & U & [a] & W}).
```

As mentioned earlier, the difficulty with a simple Prolog approach to this problem is the occurrence of infinite choicepoints created by indefinite lists. So the approach that we take here is to first impose all the constraints on the lengths of words implied by the equations, then enumerate the lengths (such that there is at most one infinite choicepoint), and then use Prolog to deal with word equations on lists of definite length.

```
word_algebra({Xs..}  ):-
      setup_constraints(  Xs,   SymbolTable),
      solve_for_lengths(SymbolTable),
      interpret_word_equations(Xs).
```

The first step is to modify the length predicate to make it deterministic by postponing choices until the length is determinate:

```
%                  CLP  length_c
%    uses freeze to maintain the relation
%    from length to list

length_c( List,  N)  :-  N:integer(0,_),  $lengthz(List,N),!.

$lengthz([Tail..],N):-  domain(N,integer(0,_)),tailvar(Tail..),!,
      freeze(N,$lengthz(Tail,  N)).
$lengthz([],0).
$lengthz([X|Xs],N):-  {N1  is  N  -  1,  N1  >=0},  $lengthz(Xs,N1).

For  example:

?-  N:integer(3,_),   length_c(L,N),   N=5.
     ?-  [5  :  integer(3,  _H169),
           length_c([_H252, _H253, _H254, _H408, _H478], 5),
         5 = 5].
```

The second step is provide a symbol table utility to manage an association between a variable, the list it represents, and its length:

```
% lookup  Var  in  symbol  table
lookup(  [V,N,List],  [TV..]):-  tailvar(TV..),!,  %  first  occurrence
      N:integer(0,_),
      length_c(List,  N),   % associate  list  and  length
      TV=[[V,N,List]|_].  % add  to  end  of  table
lookup(  [V,N,L],  [[U,N,L]|_]):-  U@=V,!.    % found  it
lookup(  X,   [_|Xs]):-  lookup(X,Xs).          % keep  searching
```

Next, we write a meta-predicate that extracts length conditions from equations:

```
%
%    Extract  and  set  up  constraints
%
setup_constraints(  [],  ST).
setup_constraints(   [E|Es],ST):-
      constraint(  E,  ST),!,
      setup_constraints(Es,ST).

% equal  lists  have  equal  lengths
constraint(  Expr1=Expr2,   ST):-
      length_of(  Expr1,  ST,  Length1),
      length_of(  Expr2,  ST,  Length2),!,
      {Length1==Length2}.   %  impose  constraint

constraint(  Op(E1,E2),   ST):-   arith_relation(Op),
      evaluate(  E1,ST,V1),
      evaluate(  E2,ST,V2),!,
      {Op(V1,V2)}.      %  impose  constraint
```

```
arith_relation('==').
arith_relation('>=').
arith_relation('=<').

% evaluation of length expressions
evaluate( N,     _,  N ):- integer(N),!.
evaluate( #  V,ST,  N ):- !,length_of(V,ST,N).
evaluate( A + B,  ST,  M + N):- !,evaluate(A,ST,M),  evaluate(B,ST,N).
evaluate( A - B,  ST,  M - N):- !,evaluate(A,ST,M),  evaluate(B,ST,N).

% length of list expression
length_of( V  ,  ST,  N)  :- var(V),!,  lookup([V,N,_],  ST).
length_of( L,    _,   N)  :- list(L),!,$length(L,N).  % only definite lists
length_of(~V  ,  ST,  N)  :- !,length_of(V,ST,N),!.
length_of( U&V  ,  ST,  M+N):-  length_of(U,ST,M),  length_of(V,ST,N).
```

To eliminate all but (at most) one infinite choicepoint, we form the sum of the list lengths so we can enumerate it first, and bind the variables to their (now determinate) lists:

```
%
%     solve_for_lengths(SymbolTable),
%
%           form the sum of all word lengths
%           then enumerate them, sum first
%    (this avoids problems when more than one of them is 'infinite')
%           also binds variables to their lists

solve_for_lengths(  SymbolTable):-  Sum:integer(0,_),  %  decl.  necessary
      sum_of_words(SymbolTable,Sum,  Nlist),
      enumerate(  [Sum]),
      enumerate(  Nlist).

sum_of_words([], 0,  []):-!.  % cut  necessary
sum_of_words([[L,N,L]|Xs],   Sum,   [N|Ns]):-
      { S1 is Sum - N },
      sum_of_words( Xs,  S1,  Ns).
```

Finally, we can interpret the word equations:

```
%
%      Interpreter  for  word  algebra
%
interpret_word_equations([]).
interpret_word_equations([X|Xs]):-
      interpret_equation(X),!,
      interpret_word_equations(  Xs).

interpret_equation(  Expr1=Expr2):-
      interpret(Expr1,L1),!,
      interpret(Expr2,L2),!,
      L1=L2.
interpret_equation(  Op(E1,E2)):-  arith_relation(Op),!.
```

```
interpret( L,    L):-  list(L),!.
interpret(  ~W,  R):-  !,interpret(W,L),
      reverse( L, [], R).
interpret(U  &  W,  R):-
      interpret( U,  UL),
      interpret( W,  WL),
      append(UL,WL,R).
%
%         standard list utilities
%
append( [], L, L).
append(  [X|Xs],  L,  [X|Zs]):-  append(Xs,L,Zs).

reverse(  [],  Ls,  Ls).
reverse(  [X|Xs],  Ls,  R):-  reverse(Xs,[X|Ls],R).
```

As an example:

```
?-word_algebra( { #U =< 3, (~U) & [a] & W & U =(~W) & U & [a] & W}).
     ?- word_algebra({(# []) =< 3,
                       ((~ []) & ([a] & ([] & []))) =
                       ((~ []) & ([] & ([a] & [])))}).
     ?- word_algebra({(# [a]) =< 3,
                       ((~ [a]) & ([a] & ([a] & [a]))) =
                       ((~ [a]) & ([a] & ([a] & [a])))}).
     ?- word_algebra({(# [a, a]) =< 3,
                       ((~ [a, a]) & ([a] & ([a, a] & [a, a]))) =
                       ((~ [a, a]) & ([a, a] & ([a] & [a, a])))}).
     ?- word_algebra({(# [a, a, a]) =< 3,
                       ((~ [a, a, a]) & ([a] & ([a, a, a] & [a, a, a]))) =
                       ((~ [a, a, a]) & ([a, a, a] & ([a] & [a, a,a])))}).

YES
```

Exercise: find an interesting non-trivial application of word algebra solvable by this method.

Bin Packing

A bin packing problem is one where one is given an assortment of objects of different types which are to be grouped into "bins" and where there are restrictions on the number and type of objects that can be placed in a bin. Usually one wants to minimize the number of bins needed, but in large practical problems it may be enough to just provide a "good," but not necessarily optimal, solution. In another variant, one starts with an existing packing and an assortment of additional objects, with the goal of minimizing the additional number of bins, and possibly the restriction that the existing groupings should not be changed, although new items can be added to existing bins. Such problems are often good abstract models for practical problems arising in the configuration of complex systems, and nicely illustrate the interaction of boolean and integer constraints.

Perhaps the most important point about such problems is that one should avoid choosing representations which indicate *which* things go *where*. Such formulations are invariant under (usually very large) symmetry groups which permute equivalent objects among the equivalent placements. Not only does this irrelevant detail enlarge the search space, but the presence of symmetry groups will block the narrowing of the search space. As we shall see in this example, even when exact placement is avoided (by formulations based on counts of objects), there may yet be symmetries present which cause problems. The addition of extra, symmetry-breaking, constraints can largely alleviate these problems, but choosing a representation without symmetries is better when possible.

A specification for bin restrictions might be given conveniently in a declarative form, as a set of simple facts, such as the following "recycling depot" example :

```
bin_types(  [red,green,blue]).

commodities( [glass,  plastic,  steel,  wood,  copper]).

requires( wood,  plastic).
excludes(glass,copper).
excludes(copper,plastic).

capacity(red, 3).       % total number of items in bin
capacity(blue,1).
capacity(green,4).

capacity( red,    wood, 1). % at most 1 wood item in any red bin
capacity( red,     steel,0).
capacity( red,     plastic, 0).
capacity( green,  wood,  2).
capacity(  green,  glass,0).
capacity(  green,  steel,0).
capacity( blue,   wood,  0).
capacity( blue,    plastic,  0).
```

Here `requires(A,B)` means that if any A's are present, then there must be at least one B present, and `excludes(A,B)` means that the presence of either item excludes the other.

To translate all the packing restrictions into constraints, we start by representing the type of a bin by integerand its contents by an integer vector and a size representing the total number of items. The other conditions are then encoded using boolean constraints to formulate the type depedencies and other conditionals:

```
op(500,  xfx,  requires).
op(500,  xfx,  excludes).
op(500,  xfx,  implies ).

bin( Type,  Contents,  Total):-
      Type:integer(1,3),   [Red,Green,Blue]:boolean,
      {  Red  ==  (Type==1),  Green==(Type==2),  Blue==(Type==3)},
      Contents=[Glass,Plastic,Steel,Wood,Copper],Contents:integer(0,_),
      { Binsize is Red*3 + Blue*1 + Green*4 },
      { Total is Glass + Plastic + Steel + Wood + Copper },
      { Total>=1, Total =< Binsize },
      Wood  requires  Plastic,
      Glass  excludes  Copper,
      Copper  excludes  Plastic,
      Blue  implies   (0== Wood + Plastic),
      Red   implies   ((0==Steel + Plastic) and (Wood=<1)),
      Green  implies  ((0==Glass + Steel) and (Wood=<2)).

X  excludes  Y  :-  {X*Y==0}.
X  implies   Y  :-  { X =< Y}.
X  requires  Y  :-  { (X>=1) =< (Y>=1) }.
```

This can be checked most easily by enumerating the possibilities:

```
?-  bin(T,C,A),  enumerate([T,A,C..]),nl,  print([T,A,C]),fail.

[1, 1, [0, 0, 0, 0, 1]]
[1, 1, [1, 0, 0, 0, 0]]
[1, 2, [0, 0, 0, 0, 2]]
[1, 2, [2, 0, 0, 0, 0]]
[1, 3, [0, 0, 0, 0, 3]]
[1, 3, [3, 0, 0, 0, 0]]
[2, 1, [0, 0, 0, 0, 1]]
[2, 1, [0, 1, 0, 0, 0]]
[2, 2, [0, 0, 0, 0, 2]]
[2, 2, [0, 1, 0, 1, 0]]
[2, 2, [0, 2, 0, 0, 0]]
[2, 3, [0, 0, 0, 0, 3]]
[2, 3, [0, 1, 0, 2, 0]]
[2, 3, [0, 2, 0, 1, 0]]
[2, 3, [0, 3, 0, 0, 0]]
[2, 4, [0, 0, 0, 0, 4]]
[2, 4, [0, 2, 0, 2, 0]]
[2, 4, [0, 3, 0, 1, 0]]
[2, 4, [0, 4, 0, 0, 0]]
[3, 1, [0, 0, 0, 0, 1]]
[3, 1, [0, 0, 1, 0, 0]]
[3, 1, [1, 0, 0, 0, 0]]
```

Exercise: Work out the source (in the original specification) and the effect of each constraint in this predicate. Verify the possible bins against the specification.

Exercise: Write a program to translate any specification of the above general form into a constraint generating clause of the same form as that given here.

For small problems (requiring relatively few bins) a complete solution (independent of the packing rules) that adds bins one at a time is feasible. (The algorithms below are given for the specific problem above, for ease of readability, but it is easy to see how to generalize them to any such problem; the specifics of the problem can be isolated entirely to the `bin` predicate.)

```
$pack( 0,  [0,0,0,0,0],  []).
$pack(  Total,   Amounts,[[Type,Contents,Size]|Bins]):-  %  nl,
print([Total,Amounts]),
      bin(  Type,Contents,Size),
      {T == Total  -  Size,  T>=0},
      subtract(  Amounts,   Contents,   Residual),
      $pack(  T,  Residual,  Bins).

subtract(  [],[],[]).
subtract( [X|Xs],  [Y|Ys],  [Z|Zs]):-  {Z  is  X  -  Y,Z>=0},
      subtract(Xs,Ys,Zs).

pack(  [Glass,Plastic,Steel,Wood,Copper]   ,   Bins):-
      Total  is  Glass  +  Plastic  +  Steel  +  Wood  +  Copper,
      $pack(  Total,   [Glass,Plastic,Steel,Wood,Copper],   Bins),
      $enum_bins(  Bins),!.
$enum_bins([]).
$enum_bins([[T,C,S]|Bs]):-   enumerate([T,S,C..]),
      $enum_bins(Bs).

?-   pack([3,4,1,4,2],_).
     ?- pack([3,  4,  1,  4,  2],
              [[1,  [0,  0,  0,  0,  2],  2],
               [1,  [3,  0,  0,  0,  0],  3],
               [2,  [0,  2,  0,  2,  0],  4],
               [2,  [0,  2,  0,  2,  0],  4],
               [3,  [0,  0,  1,  0,  0],  1]]).
YES
stats(144675,   19974,   206185,   4382,   33500)
```

Exercise: How important is it to this implementation to have a variable for the total number of items in a bin?

There is symmetry group here of order N!, where N is the number of bins in the solution, since they could have been listed in any order.

To remove some of this symmetry, one can require that the list be sorted, say by number of items in the bin and type. Because enumeration begins at the low end, it is best to make this an ascending sort.

```
order([_]):-!.
order([X,Y|Xs]):-  $order(X,Y),  order([Y|Xs]).

$order(  [T1,_,S1],   [T2,_,S2]):-
      { 1== (T1<T2) or ((T1==T2) and (S1=<S2))}.

ordpack(  [Glass,Plastic,Steel,Wood,Copper]   ,  Bins):-
     Total is Glass + Plastic + Steel + Wood + Copper,
      $pack(  Total,  [Glass,Plastic,Steel,Wood,Copper],  Bins),
      order(Bins),
      $enum_bins(  Bins),!.

?-  [stats,
        ordpack([3, 4, 1, 4, 2],
                 [[1, [0, 0, 0, 0, 2], 2],
                  [1, [3, 0, 0, 0, 0], 3],
                  [2, [0, 2, 0, 2, 0], 4],
                  [2, [0, 2, 0, 2, 0], 4],
                  [3, [0, 0, 1, 0, 0], 1]]),
         stats(81096, 9739, 84969, 2183, 14133)].
```

Exercise: The order in which the enumeration was being done agrees with that imposed, so the first solution found is the same. Then why is this method so much faster?

For large numbers of commodities and small bins, where the solutions will require many bins, this approach is still very slow. A better strategy in this case is to generate all the possible configurations of individual bins:

```
?-findset(  [S,T,C],  [bin(T,C,S),enumerate([T,S,C..])],  Bs).
```

Note that by using `findset` (instead of `findall`) we have imposed a definite sort order (as described above) on the solution list. The (*unique*) representation is then given by the multiplicities of each configuration in the solution, a vector of non-negative integers. Note that this removes the degeneracy symmetry (multiple bins with the same fill pattern) not handled by the order technique presented above. The constraint equations then become the linear equations saying that the sums of quantities over all bins are equal the total amount to be distributed, for each commodity. As a final step, we remove any bins with a multiplicity of zero:

```
fastpack(  [Glass,Plastic,Steel,Wood,Copper]   ,  NBins):-
```

```
        Total is Glass + Plastic + Steel + Wood + Copper,
        findset(  [S,T,C],   [bin(T,C,S),enumerate([T,S,C..])],   Bs),
        summation(  Bs,   Ns,   NB,   Total,[Glass,Plastic,Steel,Wood,Copper]   ),
        enumerate(  Ns),
        compress(  NB,NBins),!.

summation([],[],[],   0,   [0,0,0,0,0]).
summation([[Sz,T,Cn]|Bs],[N|Ns],[(N*[Sz,T,Cn])|Xs],   Tot,[G,P,S,W,C]   ):-
        N:integer(0,_),
        Cn=[Glass,Plastic,Steel,Wood,Copper],
        { T1==Tot  - N*Sz,
          G1== G   - N*Glass,
          P1== P   - N*Plastic,
          S1== S   - N*Steel,
          W1== W   - N*Wood,
          C1== C   - N*Copper},
          summation(  Bs,Ns,Xs,   T1,[G1,P1,S1,W1,C1]).

compress(  [],[]).
compress(  [(0*X)|Xs],   Ys):-  !,   compress(Xs,Ys).
compress(   [(N*[S,T,C])|Xs],   [(N*[T,S,C])|Ys]):-compress(Xs,Ys)./*
```

The standard enumeration strategy applied to the vector of multiplicities seems to give good (and consistent) performance, even for large quantities:

```
?-  stats,fastpack([3,  4,  1,  4,  2],_),  stats(_,_,_,_,_).
      ?- [stats,
           fastpack([3, 4, 1, 4, 2],
                     [1 * [3, 1, [0, 0, 1, 0, 0]],
                      1 * [2, 2, [0, 0, 0, 0, 2]],
                      1 * [1, 3, [3, 0, 0, 0, 0]],
                      2 * [2, 4, [0, 2, 0, 2, 0]]]),
           stats(36288, 2150, 1134, 163, 1984)].
YES

?-  stats,fastpack([32,  44,  11,  44,  230],_),  stats(_,_,_,_,_).
      ?- [stats,
           fastpack([32, 44, 11, 44, 230],
                     [11 * [3, 1, [0, 0, 1, 0, 0]],
                      1 * [1, 2, [2, 0, 0, 0, 0]],
                      10 * [1, 3, [3, 0, 0, 0, 0]],
                      2 * [2, 3, [0, 0, 0, 0, 3]],
                      56 * [2, 4, [0, 0, 0, 0, 4]],
                      22 * [2, 4, [0, 2, 0, 2, 0]]]),
           stats(36374, 2168, 1478, 167, 2033)].
YES

?-  stats,fastpack([132,  414,  1001,  414,  230],_),  stats(_,_,_,_,_).
      ?- [stats,
           fastpack([132, 414, 1001, 414, 230],
                     [1001 * [3, 1, [0, 0, 1, 0, 0]],
                      44 * [1, 3, [3, 0, 0, 0, 0]],
                      2 * [2, 3, [0, 0, 0, 0, 3]],
                      56 * [2, 4, [0, 0, 0, 0, 4]],
                      207 * [2, 4, [0, 2, 0, 2, 0]]]),
           stats(36375, 2168, 1852, 167, 2083)].
```

```
YES
```

It is certainly not obvious that the first solution found with this is one with the *minimal* number of bins, although it does not seem likely that it would be much worse than minimal, and there is a plausible argument that it is, in fact, optimal. (In any case, this problem could be remedied by adding the number of bins as an explicit variable, to be enumerated first.)

Exercise: Prove that the last version in fact gives the optimal number of bins, or find a counter example.

Exercise: Implement and test the following heuristic algorithm: work throught the bins in decreasing order of binsize, for each take as many as you can use, decrement the commodity totals, and recurse. Does this give the smallest number of bins? Why?

Continuous Primitives

Continuous Variables and Completeness

Most of the problems we have been concerned with up to this point have consisted essentially of discrete variables, either boolean or integer, for which the strategy of enumeration can be used. In the case of bounded discrete (and hence finite) domains and exact arithmetic, enumeration in principle provides complete solutions, although with many variables or large domains the computation is still impractical. In these circumstances it has not really been necessary to think very hard about how the underlying machinery of CLP(BNR) works, and the underlying narrowing semantics serves merely to prune the search space.

When we begin to deal with continuous variables, considered to be in the domain of the mathematical real numbers, it no longer makes sense to think in terms of enumeration, since there are uncountably infinite numbers of "real numbers" in every non-point interval, and we can not even refer to most of them! Furthermore, the arithmetic operations in general are no longer exact, but involve floating point approximation (known traditionally as rounding errors.) In general, then we will no longer have completeness, in the sense that we will usually not be able to give *exact* solutions to problems, nor even be able to tell automatically whether a solution exists or not. We still have correctness, however, and a properly formulated question which fails still indicates that no solution is possible.

As a result of this, we find that there is a now a sharp distinction (which was blurred in discrete domains) in the ways in which we can use the CLP(BNR). In some problems, corresponding to universally quantified variables, we seek rigorous assurance of some properties, and we therefore formulate the problem negatively (using not()) so that failure indicates a successful proof of the statement. In other, more typical, cases, corresponding to existentially quantified variables, we seek a specific solution and use a direct formulation. In this case, the answer must usually be taken as conditional: if the problem has a solution (in the initial domains of the variables), then it has a solution in the final intervals. Even when variables are fully instantiated, a positive answer is generally conditional, since it merely indicates that there is no contradiction detected at the level of precision used in the (approximate) arithmetic operations.

One aspect of this incompleteness is that CLP(BNR) constraints use only closed intervals and closed relations: for example {X==Y} is supported, but {X<>Y} is not, for X and Y continuous variables. Note that since neither X nor Y are likely to ever become instantiated to an exact value, {X<>Y}, if allowed, would likely never be able to 'fire', and even if say X does become instantiated to an exact point, and {X<>Y} fires, it would be unable to narrow Y at all, given the restriction to closed intervals. (It is possible to construct systems which use open,closed, and mixed intervals, and to propagate open/closed conditions, but these do not produce any effective narrowing.)

With discrete variables, when setting a variable to a constant or constant expression, it made little difference whether one used unification (V=C), the primitive is (V is CE), or constraint expressions like {V==C} or {V is CE}. With real variables, however, there is an important distinction: the ("constraint unaware") Prolog expression V=C will bind V to the floating point constant C *regarded as an exact (binary) constant.* Similarly, V is CE will bind V to the *result of evaluating the expression CE using ordinary floating point arithmetic*, and may thereby introduce rounding errors. For example, given X:real (and assuming 5 decimal digits of internal precision for expository reasons)

```
X = 0.33333          =>    X = 0.33333000000000000000...
X is 3               fails
X is 3.0             =>    X = 3.00000000000000000000...
X is 2/3             =>    X = 0.66667000000000000000...
```

On the other hand, the constraint expressions {V==C} or {V is CE} will treat floating point constants (other than 0.0) as approximate (and fuzz them slightly), and perform all evaluations using interval arithmetic, and also automatically coerce the type of results if necessary:

```
{X == 0.33333333}    =>    X in [0.33333000..., 0.33334000...]
{X == 3 }            =>    X = 3.00000...
{X == 3.0}           =>    X in [ 2.99999000...,3.00001000...]
{X == 2/3}           =>    X in [ 0.66666000...,0.66667000...]
```

and similarly for is. As a result, the consistent use of {} when dealing with real variables will avoid many subtle problems.

From our previous work we know that narrowing alone is not usually strong enough to solve most problems. Only on small, simple problems with at most one solution, is narrowing alone sometimes adequate. (Narrowing is deterministic, so produces at most one answer: if a problem has more than one classical solution, this answer must big enough to contain them all.) So something nondeterministic like enumeration is required, if only to "split" (or separate) solutions. At present it seems unlikely that any single technique will be able to handle all problems efficiently. One such heuristic technique provided in CLP(BNR), called **solve**, is useful for many problems, particularly those with multiple point solutions and not too many variables. Finding more general techniques, able to deal with more complex problems, is currently a research topic, and several promising lines of investigation are being pursued.

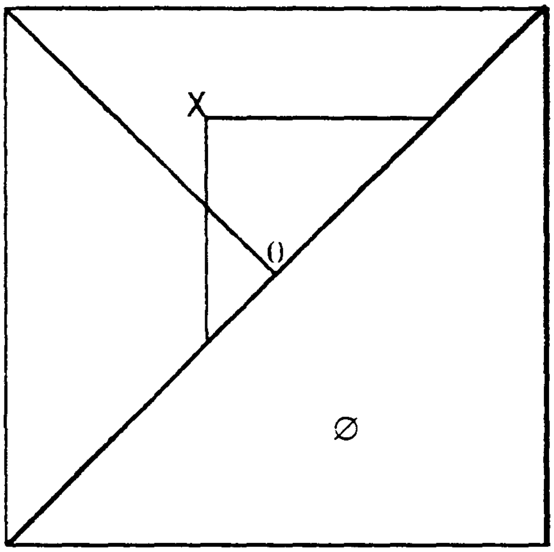
In our exploration of discrete problems we encountered two general techniques which sometimes had dramatic effects on performance: one was the judicious use of redundant constraints, and the other was the elimination of symmetries in the problem formulation. Both of these operate by effectively making narrowing stronger, and thus increasing the pruning. With continuous domains, these techniques become much more important, and will represent an important recurring topic during the rest of the course.

Intervals and Interval Arithmetic

It is useful at this point to go into a little detail about intervals and the basic concepts of classical interval arithmetic. *Interval arithmetic*, initiated by Moore in the 1960's, is both an elementary algebraic theory of arithmetic operations on intervals and a closely related computational technique for estimating worse-case rounding errors in floating point arithmetic. *Interval analysis* is a rigorous mathematical discipline, based on interval arithmetic, which lies somewhere between classical real analysis and classical numerical analysis.

The basic concepts of interval arithmetic are important basic ingredients in CLP(BNR). However, the theoretical structure underlying CLP(BNR) is quite different from that studied in classical interval arithmetic. Since these differences arise from some fairly subtle points, it is necessary to discuss both to a sufficient degree.

A closed interval [xl,xu] is the subset of the reals { x ∈ℜ| xl=<x & x=< xu}. (Here, xl and xu are in general themselves arbitrary real numbers, for theoretical purposes, but will later be restricted to floating point values for computational purposes.) Obviously, an interval is empty, unless xl=<xu. Generally we will use uppercase letters to denote intervals and lowercase letters for reals. The space of real intervals ℑℜ then can be pictured as consisting of the non-empty intervals in the upper closed half-plane, together with the empty interval (the lower open half-plane) as shown:

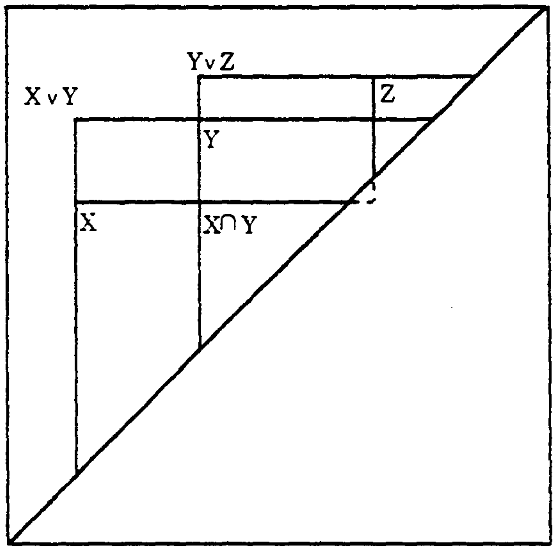


Space of Intervals

The line xl=xu, the point intervals, or "points" for short, is then isomorphic to the reals (and will be henceforth identified with the reals), and is the boundary of the space. Everything below this line corresponds to the empty interval.

The largest interval, the "top" or universal interval, which contains all other intervals, is [-∞,+∞]. (This can be done by adjoining the ideal points -∞, +∞ and adjusting the topology accordingly, the so-called two-point compactification of the real line.)

Intervals (*in classical interval arithmetic)* are considered equal if and only if they have the exact same bounds, i.e. they are identical. Intervals are partially ordered by set inclusion (⊃), and the intersection (∩) of intervals is again an interval. With closed intervals (but not with open intervals) this is also true for arbitrary (e.g. infinite, even non-countably infinite) intersections. The union of two intervals (regarded as sets) is not generally an interval, but there is a *join* interval (v), which is defined as the smallest interval containing both, always exists when there is a universal interval. This makes the set of intervals into a (complete, but not distributed) *lattice*. The intersection (sometimes called *meet*) and join are illustrated below:



Meet and join

The basic *relations* of equality and inclusion between intervals, it should be noted, are inherited from sets, and have nothing to do with the reals as such. With this relational structure in place, classical interval arithmetic proceeds to extend the basic *functions* of the reals (as a field) to the intervals, which we will now explore.

The half line xl=-xu (corresponding to intervals of the form [-b,b], b>=0) will be called the *symmetric intervals*. The only symmetric point interval is 0. The symmetry operation here, which we will call "flip", is defined by

$$\sigma([xl,xu]) := [-xu,-xl],$$

so, σ(σ(X))=X. Obviously, we have the following *inclusion theorem*:: a point x is in interval X iff -x is in σ(X), so σ corresponds to unary negation. This operation does not involve any rounding when done with floating point numbers, so we say it is *exact*.

Sometimes it is convenient to describe intervals in terms of their midpoint and width (or delta), according to the change of coordinates:

midpoint := (L + U)/2,
delta := (U-D).

In this representation points are those with delta=0, and symmetric intervals are those with midpoint =0.

Addition of intervals is like vector addition in the plane:

[xl,xu] + [yl,yu] := [xl + yl, xu + yu].

Interval sum is hence commutative and also associative when doen in infinite precision. Note that the midpoint(delta) of the sum is the sum of the midpoints(deltas). Because addition is a monotone increasing function in both variables, it follows that we get the inclusion property:

$x \in X$ & $y \in Y \Rightarrow (x+y) \in (X + Y)$.

The interval defined as the sum is the smallest interval for which this holds.

Since subtraction in the reals can be defined as

x - y := x + (-y) ,

we can *lift* this definition and define subtraction on intervals as:

$X - Y := X + \sigma(Y)$,

and it will then follow that:

$x \in X$ & $y \in Y \Rightarrow (x-y) \in (X - Y)$.

Note that the midpoint of the difference is the difference of the midpoints, but the delta of the difference is the *sum* of the deltas. In the reals, we always have 0=x-x, but in intervals S= X - X is not zero unless X is a point. (Such an S is always symmetric, however. Note that the sums and differences of symmetric intervals are also symmetric.) As a consequence of this, subtraction is for intervals *not* the inverse of addition: i.e. it is *not* the case that X + Y - Y = X.

An interval X is *non-negative* iff xl>=0, and *positive* if xl>0. Since multiplication x*y is monotonically increasing for non-negative x and y, we can define the product of two non-negative intervals as

[xl,xu] *[yl,yu]=[xl*yl,xu*yu], for yl>=0,xl>=0

and show that this is the smallest interval such that

$x \in X$ & $y \in Y \Rightarrow (x*y) \in (X *Y)$.

For non-positive X and non-negative Y, $\sigma(X)$ is non-negative, and since -(x*y) = (-x)*y we can define $X*Y = \sigma(\sigma(X)*Y)$ for non-positive X, non-negative Y, from which inclusion results follow, and similarly for the other cases. In fact, all cases of multiplication can be subsumed with a single definition:

[xl,xu] *[yl,yu]=[ml,mu

where ml=min(xl*yl, xl*yu, xu*yl, xu*yu)

and mu=max(xl*yl, xl*yu, xu*yl, xu*yu).

Multiplication is commutative and associative (in infinite precision), but the distributive law is weakened to the so-called *subdistributive law*:

$$X*(Y+Z) \supset X*Y + X*Z$$

Any interval X times a symmetric interval Y yields a symmetric interval, so that symmetric intervals are analogous to ideals in the "pseudo-ring" structure (+,*,0,1). (It is not really a ring, of course, since subtraction does not undo addition and the distributive law fails.)

For a positive (resp. negative) interval X, $\rho(X) := [1/xu, 1/xl]$ is an interval, and $x \in X \iff 1/x \in \rho(X)$. Together with multiplication this allows us to define interval division as

$X/Y := X*\rho(Y)$, provided $\sim (0 \in Y)$,

with the usual inclusion results. But, as with subtraction, X/X does not equal 1, except when X is a point.

The ideal intervals which we have been discussing are useful theoretically, but for computations one is interested only in intervals with floating point bounds (of some fixed precision, i..e. fixed-size representations). These are sometimes called F-intervals, where F is the set of permitted bounds. Because of the limited precision, the result of these interval operations is in general not an F-interval when the inputs are F-intervals. This is remedied by using a F-closure operation defined as:

ϕ(X) = smallest F-interval larger than X.

This is also known as "outward rounding," since the upper bound is rounded toward $+\infty$ and the lower bound toward $-\infty$. By rounding the result outwards after each interval operation we can preserve the inclusion properties. But the associative laws of addition and multiplication (as well as some other properties) are lost, since floating point arithmetic is not associative due to the rounding.

These definitions and their assorted formal properties constitute interval arithmetic. As a formal structure it differs a great deal from the formal structure of the reals: most of the axioms of the reals no longer hold for intervals. *Consequently, most algebraic derivations which are formally justified in the reals will not be justified for intervals, so formal reasoning with these classical intervals is difficult.*

However, given a real function $f(x_1...,x_n)$ expressed in a definite finite syntactic form in terms of (+,-,*,/) and intervals $X_1...X_n$ such that the corresponding interval operations are defined (specifically, no divisions by intervals containing 0), then we can form the *natural interval extension* $F(X_1,....,X_n)$ by syntactically replacing x_j by X_j and real operations by the corresponding interval operations. Since the inclusion property holds at each operator, an induction over the syntax tree yields the result that

$$\forall x_1 \in X_1,... \forall x_n \in X_n \quad f(x_1,...,x_n) \in F(X_1,...,X_n).$$

This is called the *fundamental theorem of interval arithmetic.*

In general syntactic forms which are mathematically *equivalent* over the reals generate *different* extensions and produce *different* resulting intervals. But, if the natural interval extensions exist, each satisfies the inclusion theorem (and so therefore does their meet.) To derive interval inclusions for a function, then, one can proceed by manipulating the function over the reals in the ordinary way until it is in any suitable syntactic form, and then take the natural inclusion of that form to get a valid inclusion. (Much of the theory of classical interval arithmetic was concerned with choosing good syntactic forms which make the inclusion as tight as possible, and the formal properties of intervals are useful for this.)

An implementation of interval arithmetic would minimally consist of an abstract data type for intervals together with the appropriate constructors and the basic relations and operations described above. Such an implementation, regarded as an abstract language, would necessarily have the (somewhat peculiar) algebraic/axiomatic structure described above, a structure very different from that of real arithmetic.

Relational Interval Arithmetic

At the CLP(BNR) linguistic level the key entities are the typed variable and the constraint. The *current* domain of a typed variable is always an interval in the sense of interval arithmetic; as the domain is narrowed during a forward computation one sees a nested sequence of such intervals. The conceptual variable in CLP(BNR) thus corresponds to many such nested sequences of intervals (under different forward computations and backtrackings), so the relationship to interval arihtmetic is not such a simple one.

Similarly, a single constraint at the CLP(BNR) level corresponds to many different functions in the interval arithmetic world - one for every way in which the relation could be uniquely 'solved' for any variable in terms of all the rest, and even this may not exhaust its potential. Thus even relatively simple CLP(BNR) formulations can explode into a bewildering complexity when translated into the classical language of interval arithmetic. This point has usually not been appreciated.

In order to get closer to the level where comparisons with interval arithmetic become meaningful, one needs to drop below the CLP(BNR) linguistic level to the underlying technology. Relational interval arithmetic (RIA) is the name used for this technology-- data structures, algorithms, and relevant theory--which supports CLP(BNR) and similar languages.

In the early days (Cleary,1986) RIA was thought of as an adaptation of interval arithmetic to the needs of the Prolog environment. However, as the technology evolved and the structure of the theory became clearer, it has become evident that although they use the same basic concepts of intervals, outward rounding, and inclusion properties, and share many of the same goals, they are structurally very different and have very different properties and uses.

The fundamental construct in relational interval arithmetic is the approximation of relations on the reals by narrowing operators on lattices created from intervals. The crucial difference with interval arithmetic is the choice of relations (rather than functions), and the ubiquitous use of narrowing operators. (Narrowing operators do appear in interval analysis, but only occur in special circumstances.) The fact that RIA maps constraints (and sets of constraints) to operators is an important one, and often misunderstood because it seems so abstract. To overcome this, we will spend some time looking at specific primitive relations and their operators, and so develop a better intuition for what it really means . .

We have already noted that closed intervals on the extended line form a complete lattice - a partially ordered set with top and bottom elements and meet and join always defined. This lattice is also *atomic*, where the atoms (smallest non-empty elements) are the point intervals. It is sometimes convenient to take the lattice as an abstract lattice L of *states*, with many possible realizations: e.g., all sets, all closed sets, all intervals, all closed intervals, all closed F-intervals,... , and postpone specialization until needed.

In order to treat a relation of arity n, which is a subset of $\Re n$, it is useful to construct a lattice L of states in $\Re n$ by defining a state in $\Re n$ to be the n-fold Cartesian product of states in $\Re$. A non-empty state X can be conveniently represented as a vector of 1-dimensional states (sometimes called an interval vector):

$$X = [X_1,\dots,X_n],$$

with all X_j non-empty. The empty state will be denoted by $\oslash$. Partial order between states is defined by set inclusion, and meet by set intersection, and join by the lattice join (which, as above, is not in general set union). Equivalently, these operations can be defined termwise in the interval vector representation, with the proviso that if any component of the termwise meet is empty, the result is empty:

$$X \supset Y <=> X_1 \supset Y_1 \&\dots X_n \supset Y_n$$
$$X \vee Y <=> [X_1 \vee Y_1, \quad ,X_n \vee Y_n]$$
$$X \cap Y <=> [X_1 \cap Y_1, \quad ,X_n \cap Y_n] \text{ if all are non-empty and } \varnothing \text{ otherwise.}$$

The lattice of states defines a closure relation on sets (as in the 1-dimensional case) according to:

$$\phi(X) = \{ \text{smallest state } Y \text{ such that } Y \supset X\},$$

which can also be expressed termwise as:

$$\phi([X_1,\dots,X_n]) = [\phi(X_1),\dots,\phi(X_n)]$$

These closure relations have the following properties:

expanding: $\phi(X) \supset X$
monotone: $X \supset Y => \phi(X) \supset \phi(Y)$
idempotent: $\phi(\phi(X))=\phi(X)$.

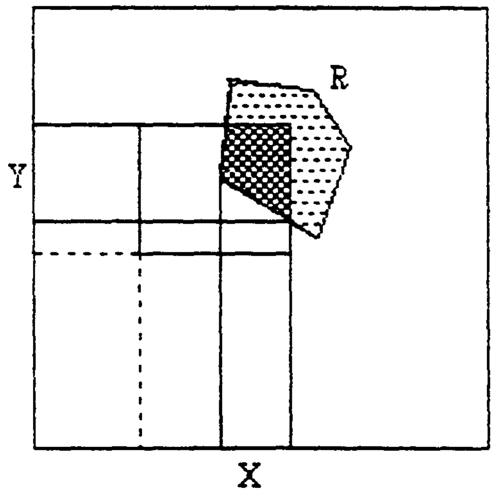
When the lattice is the full powerset, the closure operation is the identity; when it is the lattice of closed sets it is the usual topological closure on the reals.

(Note: The existence of the closure operation depends only on the poset of states having a top and being closed under arbitrary meets, since it can then be defined as $\phi(X) = \bigwedge \{Y \mid Y \supset X\}$ with the set of Y's being non-empty. But these conditions also guarantee the existence of the join and make the states a lattice.)

Given an arbitrary n-ary relation R, we can then define the operator R:L->L by:

$$R(X) =\phi(X \cap R).$$

This is illustrated for n=2 in the figure below:



Canonical Narrowing Operator

This operator R is correct (or "conservative") with respect to the relation R since $R(X) \supset X \cap R$. It also has the following abstract properties:

contracting: $X \supset R(X)$

monotone: $X \supset Y \Rightarrow R(X) \supset R(Y)$

idempotent: $R(R(X))=R(X)$.

We will henceforth call any lattice operator with these properties a narrowing operator. (Note that this generalizes the notion of narrowing operator used earlier for Prolog by weakening persistence to idempotence, which is a obvious consequence of persistence. The

Prolog operators will henceforth be called persistent narrowing operators. Later on we will show how the persistence gets put back-with a vengeance!)

There may be many narrowing operators correct for some relation R, but the definition makes it clear that this R is the *smallest* correct narrowing operator for R, and is therefore unique. We call it the *canonical narrowing operator of R.*. It provides the basic recipe for implementing optimal interval approximations to the various primitive relations supported by CLP(BNR). Usually several refinement steps are required to transform this abstract recipe into a sufficiently low-level and efficient executable specification, and in many instances there is an explosion of cases (possibly over 200 for a single primitive!) to be considered, so that the resulting code is extremely complex and it is not easy to discern its properties. Hence, it is important that all the fundamental properties can be so easily established from the abstract specification.

We have already dealt with operators on a lattice before, at the very beginning of this course, and you may recall the basic notation:

Recall that operators are partially ordered by

$P \supset Q \iff P(X) \supset Q(X)$ for all states X in L.

Equality of operators is similarly defined by

$P = Q \iff P(X) = Q(X)$ for all states X in L,

so we have $P = Q \iff P \supset Q \;\&\; Q \supset P$.

The meet and join of operators with respect to this partial order exist and are given by:

$P \cap Q \iff P(X) \cap Q(X)$ for all states X in L,

$P \vee Q \iff P(X) \vee Q(X)$ for all states X in L.

The smallest operator ("bottom") in the partial order, called 0, is obviously the one which maps every state to $\varnothing$:

$0(X) = \varnothing$ for all states X in L.

(This operator would be one which just fails, whatever you give it.) The identity operator, which does nothing at all, is denoted by 1, and is similarly defined by:

$1(X) = X$ for all states X in L.

Finally, the product of operators is defined by composition:

$PQ(X) := P(Q(X))$ for all states X in L.

We will be using this notation below, but not get into the algebra until later.

Exact Primitives

There are a number of important primitive relations which do not perform any arithmetic operations, so rounding issues do not arise.. For these *exact* relations, it makes no difference whether we use F-intervals or intervals as the state lattice.

Integral/1

The "integral" primitive (of arity 1) does not appear explicitly in CLP(BNR), but is used *implicitly* to construct the integer and boolean types. Its formal definition as a unary relation is

integral(x) <=> x is an integer.

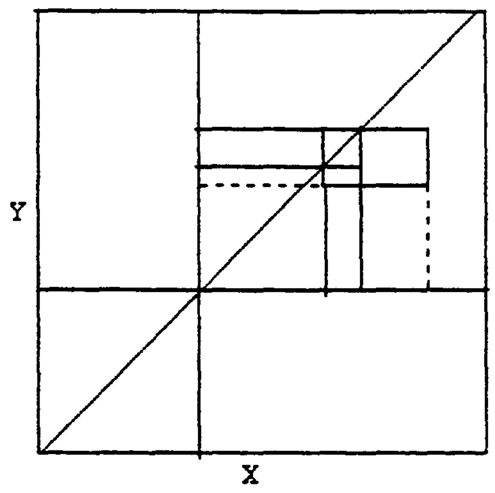
The corresponding narrowing operator (by inspection) is

integral([xl,xu])= [xl',xu']
where xl' = ceiling(xl), xu'=floor(xu).

Here [xl,xu] denotes the state (a single interval) before the operation, and [xl',xu'] the state after the operation. This primitive seems very simple--too simple to be useful. But all of the applications of CLP(BNR) to integer and boolean problems, and hence the subsumption of specialized finite domain and boolean solvers, follow from it.

Equality/2: X==Y

The equality relation on the reals is represented by the diagonal set D in $\Re x \Re$. Given initial state [X,Y] as shown in fig., we form the intersection as shown and take the interval closure to get the final state [X',Y'].



The operational effect for an arbitrary placement of the initial state [X,Y} can be expressed succintly as:

U <- X ∩ Y,
X'<- U, Y'<-U.

(Compare this with the narrowing semantics of unification.) Thus the *output* intervals X' and Y' are equal in the (static) sense of interval arithmetic. But the '==' relation in CLP(BNR) refers to the process of making them equal (and, as we shall later see, maintaining them equal), and not the mere fact of equality itself. This will have important consequences later.

When expressed in even lower level language, in terms of bounds, this becomes:

```
equality( [[xl,xu],[yl,yl]]) --> [[xl',xu'],[yl',yu']]:-
      xl'<- yl'<- max(xl,yl);
      xh'<- yh'<- min(xu,yu);
      if xl'>xh' then fail.
```

and even lower:

```
equality( [[xl,xu],[yl,yl]]) --> [[xl',xu'],[yl',yu']]:-
      xl'<-xl'; xh<-xh; yl'<-yl;yh'<-yh;   % default
      if  xl >yl then yl'<- xl
      else if yl>xl  then xl'<- yl
      else ;
      if   xh<yu then yu'<- xu
      else if yu<xu then xu'<- yu
      else ;
      if xl'>xu' then fail.
```

This can be summarized in a *propagation diagram*:

xu <-> yu

xl <-> yl

which indicates all the ways that bounds can be propagated in the primitive. Note here that only upper bounds can affect upper bounds (either way) and similarly for lower bounds.

This, like integral, appears to be trivial, as indeed it is, when regarded as a mere piece of code.. But also like integral, it has the most profound consequences in the CLP context. (In fact, much of the rest of the course will be spent exploring these consequences.)

To begin with, we will establish three simple properties of this definition. In order to state them succintly we need to use the operator notation summarized earlier. Here we will use { E } to mean the canonical operator for E, where E is a primitive relation and possibly a list of such relations.

Property 1. { X == X } = 1.
This says that if we use the same interval for both input arguments to equality, it will do no narrowing, and hence is equal (as an operator) to the identity operation.

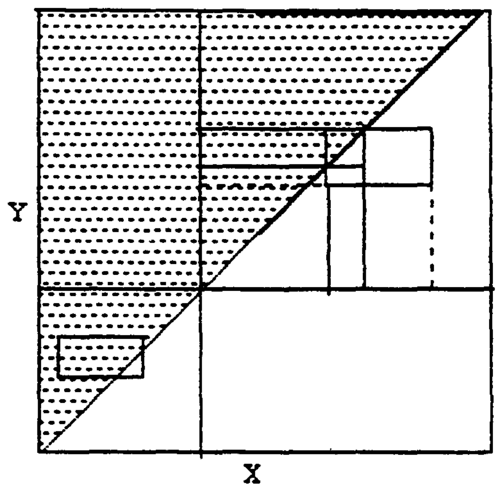
Property 2. { X == Y } = { Y == X }.
This expresses the fact that equality is symmetric under intercahnge of its input arguments.

Property 3. { X == Z } ⊃{ X ==Y, Y == Z }.
The left side doesn't use Y at all, so doesn't narrow it. We have not yet explained what the comma means (that comes later), but suppose {P,Q} to mean that we do P, then Q, then P again and so on until nothing more is changing. Then the right side winds up with interval X and interval Z the same, regardless of the starting values, i.e. just as does the left side. But the right side has narrowed Y also (to the same interval), hence the ⊃.

These properties should look sort of familar by now: they are essentially the reflexive, symmetric, and transitive laws of equality as they are expressed in narrowing semantics. Note that (in this particular case) the ⊃ corresponds to backwards implication <- , = to the biconditional, and comma to 'and'.

Inequality/2: X =< Y

The relation '=<' is represented by the half-plane shown in fig.



X=<Y

Carrying out the steps of the operator construction yields the operational specification:

```
less_than( [[xl,xu],[yl,yl]]) --> [[xl',xu'],[yl',yu']]:-
      xl'<-xl'; xh<-xh; yl'<-yl;yh'<-yh;   % default
      if  xl >yl then yl'<- xl
      else ;
      if xu>yu then xu'<- yu
      else ;
      if xl'>xu' or yl'>yu'  then fail.
```

and the propagation diagram:

xu <- yu
xl -> yl

stating that upper bounds propagate down (towards the lower value) and lower bounds only propagate up.

This operator has the following (not unexpected) properties:

$\{ X =< X \} = 1$
$\{ X =< Y, Y=< X\} = \{ X == Y \}$
$\{X=<Z\} \supset \{ X=< Y, Y=< Z\}$

The propagation diagram above suggests that we can operationally define primitives with similar diagrams, such as:

xu <- yu
xl <- yl

This primitive would be written as X<=Y. Since it's semantics propagate changes in Y to X, but not the other direction, it acts like a sort of "diode". Note that

{X<=Y,Y<=X} = {X==Y}.

If X and Y are point intervals, it is equivalent to equality, but it is not the canonical narrowing operator for equality. Its declarative meaning is that X is *constrained* to be a subinterval of Y, $Y \supset X$. If X is thought of as an unknown point, it can be interpreted as $X \varepsilon Y$. The ability to enforce these second-order relations as constraints will be exploited in some of the more advanced algorithms discussed later.

There is one important elementary use of diode: in many applications it is undesirable that input data should become narrowed by the computation that uses it. To avoid such narrowing it is necessary to use diodes to read the data into intermediate variables first, e.g.

```
{V <= Data},
```

which narrows V to the same bounds as Data, but insulates Data from any subsequent narrowing of V.

The other second-order primitives are X|=Y ("start together") and X=|Y ("end together") correspond respectively to:

```
xu    yu          xu<->yu
xl <-> yl         xl    yl
```

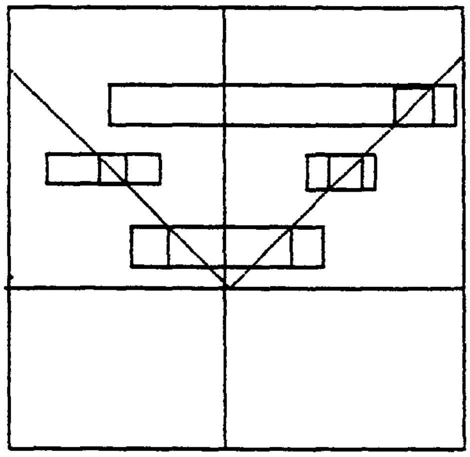
Note that

{X|=Y, X=|Y} = {X==Y}.

Probably the chief use of these is to enforce overlap conditions: given X constrained to be to the left of Y (ie. {X=<Y}) and initially overlapping Y, we can define a new overlap variable XOY by { Y|=XOY, XOY=|X}. This creates a situation where if X and Y ever cease to overlap due to narrowing, XOY becomes empty and triggers failure.

Absolute value/2: Y==abs(X)

The binary relation corresponding to the absolute value function looks like:


Y ==abs(X)

For positive intervals X, it behaves like equality; for negative one, it is equivalent to negation. For X containing zero and Y positive, as shown above, the final X interval may contain points which cannot be solutions because of the interval hull closure. In other such cases, as seen below, the canonical narrowing operator will do be better:

Note that this is due to the difference between the canonical prescription:

$X' := \pi_1 R(X,Y)$
$Y' := \pi_2 R(X,Y)$

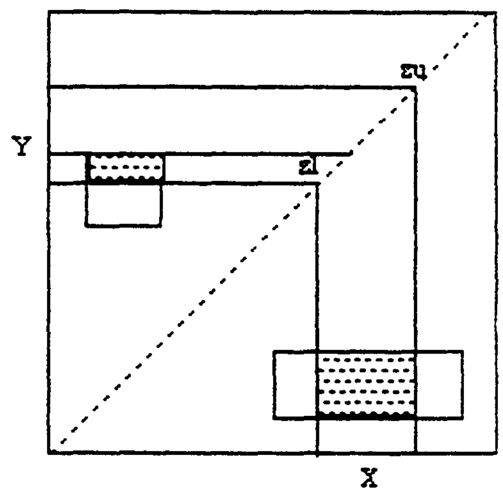
and the weaker prescription

$X' := \pi_1 R(T,Y)$
$Y' := \pi_2 R(X,T)$.

Similar behaviour is seen in any relation with a fold, such as even powers and trigonometric functions.

minimum/3: Z==min(X,Y)
maximum/3: Z==max(X,Y)

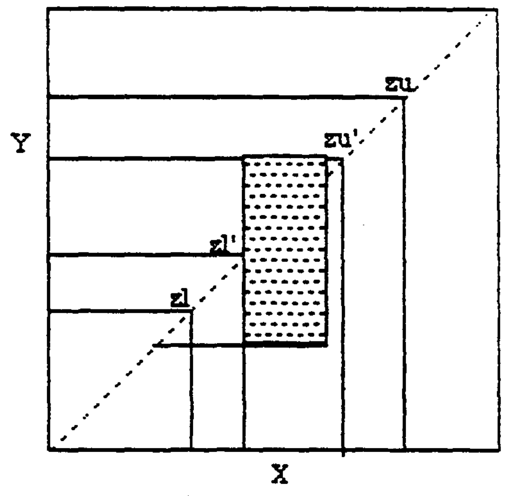
The minimum and maximum relations, being of arity 3, are harder to visualize. The most useful approach is to graph the XY rectangle and the level curves of the corresponding function for the Z bounds. For maximum the level curves are L-shaped with the point of the L on the diagonal, as shown below. From this we see that for disjoint intervals X and Y, Z becomes equal to the larger of the two.



Z==max(X,Y)

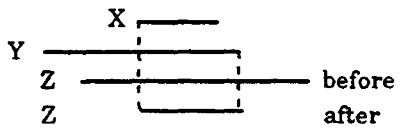
Minimum is similar, with the direction of the L reversed.

With overlapping intervals, the behaviour is much more complex:

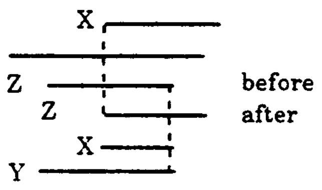


Z==max(X,Y)

The intuitive meaning becomes clearer if we project the intervals onto the diagonal, both before and after the operation:



We see that the lower bound of Z becomes the larger of the two lower bounds, and the same with its upper bound. Further narrowing of Z can be propagated backwards to X and Y also:



A high level (and somewhat redundant) pseudo-code description for minimum is:

```
minimum([Z,X,Y]) --> [Z',X',Y']:-
    less_than([Z,  X])->[Z',X'],
    less_than([Z',Y])->[Z',Y'],
    Z' <- Z' ∩min(X,Y).
```

where the less_than calls can be macro-expanded from the definition given earlier, and `min(X,Y)` is the interval arithmetic inclusion for min:

```
    min([xl,xu],[yl,yu] -> [ zl,zu]
        where zl:= min(xl,yl), zu:=min(xu,yu).
```

Here upper bounds affect upper bounds only and lower bounds affect lower bounds only, with a propagation diagram:

xu<->zu<->yu
xl <-> zl <-> yl.

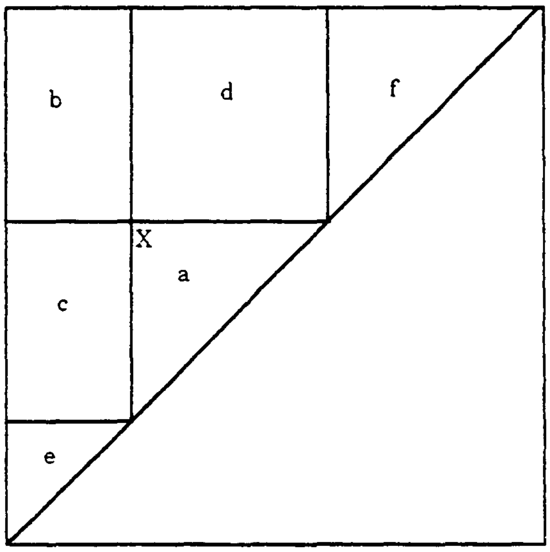
The maximum relation is analogous.

choice/3: Z ==(X ; Y)

The choice function in ordinary arithmetic 'Z is (X;Y)' is non-deterministic, returning the two answers Z=X or Z=Y. (Note the mandatory parentheses!) The corresponding constraint primitive, like all constraint primitives, is deterministic. If Z is initially the universal interval and X and Y are bounded, Z becomes the join of X and Y. Subsequent narrowing of Z does not propagate to either X or Y, until one or the other becomes disjoint from Z, at which point it reduces to equality with the remaining candidate. Thus, the constraint serves to (1) defer the choice, while (2) propagating as much as possible from X and Y to Z, and (3) automatically making the choice as soon as no alternative is available.

Summary of basic interval relations

Most of the basic relations between intervals are summarized in the following diagram. It supposes that some general interval X is given, and divides the space of intervals into distinct regions such that any interval Y in any one of the regions has the same qualitative relation to X.



Relative to X, interval Y is
- a: subinterval, Y<=X
- b: superinterval, X<=Y
- c: =< & overlapping
- d: >= & overlapping
- e: =< & disjoint
- f: >= & disjoint

The horizontal line through X corresponds to =|

The vertical line through X corresponds to |=

Interval Relations

Addition/3 : Z == X + Y

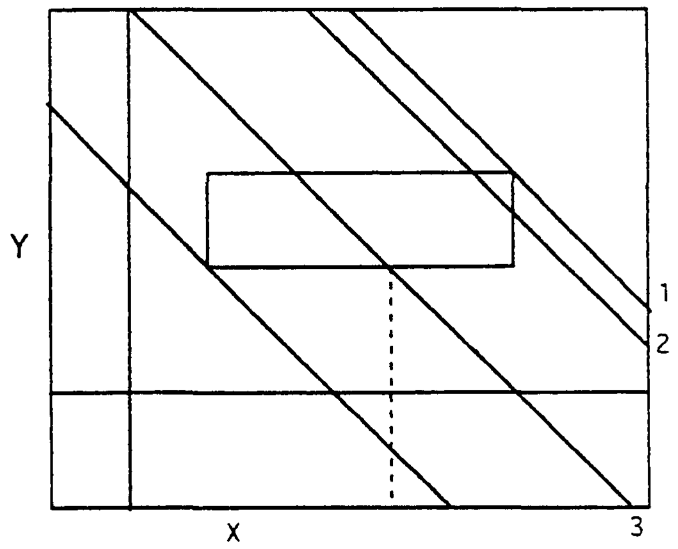
As an example of a non-exact primitive we consider addition. The relation $z=x+y$
can be solved in turn for each variable:

$$z=x+y,\ x=z-y,\ y=z-x$$

in which the right hand sides are continuous functions. The corresponding interval extensions:

$$Z':=X+Y,\ X':=Z-Y,\ Y':=Z-X$$

then provide a narrowing operator which is minimal (because of the properties of the interval functions +,-) and therefore is the canonical narrowing operator. Although convenient for implementation, it is worthwhile examining the canonical construction as well:



Z==X + Y

Here the outermost Z - bounds (upper bound labeled 1 and unlabeled lower bound) resulted from the forward narrowing of Z. If the upper bound of Z is subsequently lowered (line labeled 2), we see that the narrowing does not propagate to either X or Y until a certain threshhold is reached (i.e. the lower right corner of the rectangle). Past that point, there is narrowing, in this case of X, as shown by the dashed line. Eventually, when the upper left corner is passed, both X and Y will be narrowed.

The threshhold effect is common in arity 3 primitives. Note, however, that the narrowing which goes into overcoming the threshhold, and which apparently disappears, is not necessarily lost, but may be released by a subsequent narrowing of Y, which reduces the threshhold, thereby generating a "rebate". The second point to notice, is that the backwards narrowing appears first in the larger of the two input intervals, X in this case. This also is a general qualitative phenomenon, although a precise definition of "larger" is difficult in the case of more complex primitives. The general tendency is for information to flow from the more precisely known intervals to the least precisely known, tending towards a state of more uniform uncertainty.

We note some obvious properties of addiiton:

$$\{ Z==X+Y\} = \{Z == Y+X\}$$

and

$$\{ Z== X+0\} = \{Z == X\}.$$

(The latter depends on 0 being treated as an exact value.)

What about the associative law? We can not write

$$\{ U==X+Y, W==Z+U\} \mathrel{?} \{ V== Z + X, W==V + Y\}$$

since clearly U and V have no comparison. But if we write the operators as $\{ W== Z + (X + Y)\}$ and $\{ W==(Z + X) + Y\}$, so that the same variables are explicitly mentioned in both, and the comparison is done only on these variables, it is still not the case that these operators are equal, because floating point addition is not always associative. However, both operators are correct and hence contain all possible real solutions to the relation over the original domains. We can therefore write:

$$\{ W== Z + (X + Y)\} \approxeq \{ W==(Z + X) + Y\}$$

indicating a form of asymptotic equivalence: roughly, that both sides are correct approximations to the same set of ideal solutions. Note also that this can also be expressed as:

$$\{ Z + (X + Y)==(Z + X) + Y\} \approxeq 1$$

with the interpretation that the operator is one that never fails, ie. is an effective tautology.. The reasoning is that each sum computes an outer estimate for the true sum, so the intervals going into the equality must have a non-empty intersection.

Thus the associative law is formally obeyed in RIA, unlike the situation in classical interval arithmetic. Since both are using the same implementation of interval addition, the difference is due to the choice of equality: interval arithmetic used a strict and static notion of equality on intervals, while RIA constructs equality by the same process by which it constructs addition. The same considerations apply also to the other laws of real analysis, such as the distrbutive law, which holds in RIA but not in interval arithmetic even in infinite precision. This formal validity of the standard laws has many profound consequences which we will investigate later.

CLP Prototype Implementation

INTRODUCTION

The interval arithmetic system of BNR Prolog was one of its most advanced (and complex) features. For various purposes it is useful to have a succint executable specification written in 'normal' (BNR) Prolog of a simplified CLP implementation.. This document describes such a protypeI

The specification is divided into several levels, which are treated in bottom-up fashion.

FUNCTIONAL INTERVAL ARITHMETIC

The principles of interval arithmetic are very old, but were first systematically explored by Moore in the 1960's. The basic idea is to compute *inclusions* of arithmetic functions of intervals, i.e. interval outputs which always include *all* the real solutions. In particular, it is necessary to bound the effect of rounding errors when fixed precision arithmetic is employed. We therefore assume that we are given two basic routines for doing arithmetic: `$isL`, and `$isH`.

```
op(700,  xfx,  '$isL').   % rounds  low  (left)

op(700,  xfx,  '$isH').   % rounds  high  (right)
```

These are similar to `is`, but round results left (resp. right). (Note: rounding left means towards negative infinity, not towards 0!) Such special routines would normally have to be added to a Prolog system as new primitives or as externals. In addition we assume the primitive arithmetic comparison operators, which will be denoted by `<=`, `=>` so as not to be confused with =< and >=, their interval-extended counterparts. These primitives can be implemented independently of rounding issues. Interval values (i.e. closed intervals with floating point representable endpoints) are represented as 2-element lists `[L,U]` with `L <= U`.

The most important basic function is that to intersect two interval values:

```
intersect( [XL,XH] , [YL,YH],[IL,IH] )   :-
           IH  $isL  min(XH,YH),
           IL  $isH  max(XL,YL),
           IL  <=  IH.
```

For notational convenience, we will construct a little function evaluation language to perform the traditional functions on intervals:

```
op( 700, xfx, :=).      %     functional  "assignment"
op( 500, xfx, ^ ).      %   ^ denotes  interval  intersection

Z := X ^ Y    :-    Z1 := Y, intersect( X,Z1,Z).

[L,H]   := [Exp1, Exp2] :-          % general (explicit )   case
      L $isL Exp1,   H $isH Exp2.
```

The rest of the interval functions are defined by:

```
Z := [XL, XH] + [YL,YH]   :-   Z := [XL + YL, XH + YH].
Z := [XL, XH] - [YL,YH]   :-   Z := [XL - YL, XH - YH].
Z  :=  min([XL,XH],[YL,YH]):-   Z  :=  [min(XL,YL),  min(XH,YH)].
Z  :=  max([XL,XH],[YL,YH]):-   Z  :=  [max(XL,YL),  max(XH,YH)].
Z := [XL, XH] * [YL,YH]   :-   Z := [ XL*YL ,   XH*YH ].
                     %   (use   * only  with  non-negative  intervals)

[SL,SH]  := [XL, XH] / [YL,YH]   :-   %  dividing  intervals
             universal_interval(  [Neg_inf,Pos_inf]),
             YH > 0 -> SL   $isL  XL / YH ; SL = Neg_inf,
             YL > 0 -> SH   $isH  XH / YL ; SH = Pos_inf.
                     %   (use   / only  with  non-negative  intervals)

universal_interval(   [-3.4000e+38,   3.4000e+38]).
```

(Note: The universal interval is implementation specific and may depend on the details of the floating point algorithms/hardware. Ideally the universal interval is from negative infinity to positive infinity. Note that division by an interval containing 0 produces an answer in this system because of the existence of a universal interval.)

The fundamental principle of traditional interval arithmetic is simply that if each functional operation is implemented as above it computes an inclusion and is thus guaranteed to include the correct answer given that the inputs are within the specified ranges. It follows that any function built up from these primitive functions also computes an inclusion.

RELATIONAL INTERVAL OPERATIONS

In the first part we have discussed functions from intervals to intervals. We now turn to the relations between intervals. Note that these relations are *not* part of traditional (functional) interval arithmetic, nor do they correspond to any of the relations used in that subject. The basic relational operations used here correspond to equality, less-than-or-equal-to, addition, multiplication, and min and max:

```
ternary_relation($add).          binary_relation($eq).
ternary_relation($mul).          binary_relation($le).
ternary_relation($min).
ternary_relation($max).
```

The equality relation reduces to interval intersection:

```
$eq( X, Y, New, New) :-    New :=X ^ Y.
```

The first two arguments represent the inputs , the last two the outputs. (Do not confuse this with interval equality in the interval arithmetic literature, which would correspond to = here.)

The inequality relation is only slightly more complicated:

```
$le( [LX, UX], [LY, UY], NewX, NewY):-                  % X =< Y
            universal_interval(  [Neg_inf,   Pos_inf]),
            NewX :=    [LX,UX]  ^  [Neg_inf  ,UY],
            NewY :=    [LX,  Pos_inf  ]  ^  [LY,UY].
```

The $min and $max relations are defined by:

```
$min(   X,Y,[LZ,UZ],    NewX,NewY,NewZ):-                    %   Z==min(X,Y)
            NewZ1  :=  [LZ,UZ]  ^    min(X,Y),
            not(  NewZ1   ^  X)   ->  $eq(Y,NewZ1,NewY,NewZ)
            |  not(  NewZ1   ^  Y)   ->  $eq(X,NewZ1,NewX,NewZ)
            |  [  universal_interval(  [_,   Pos_inf]),
                 NewX  := X  ^  [LZ,  Pos_inf],
                 NewY  := Y  ^  [LZ,  Pos_inf]
                 NewZ  := NewZ1
              ].

$max(   X,Y,[LZ,UZ],    NewX,NewY,NewZ):-                    %   Z==max(X,Y)
            NewZ1  :=  [LZ,UZ]  ^    max(X,Y),
            not(  NewZ1   ^  X)   ->  $eq(Y,NewZ1,NewY,NewZ)
            |  not(  NewZ1   ^  Y)   ->  $eq(X,NewZ1,NewX,NewZ)
            |  [  universal_interval(  [  Neg_inf,_]),
                 NewX  := X  ^  [Neg_inf,UZ],
                 NewY  := Y  ^  [Neg_inf,UZ],
                 NewZ  := NewZ1
              ].
```

Addition takes three input intervals and returns three updated intervals (or else it fails):

```
$add(  X,Y,Z,   NewX,NewY,NewZ)  :-  ...
```

The inputs define a set (a closed parallelopiped) $X \times Y \times Z$ while the *relation* z=x+y has a graph G which is also a closed set when restricted to $U \times U \times U$ (where U is the universal interval). The procedure is then to intersect these two sets to form a new closed set G', and then to project this onto the coordinate axes to form the outputs X',Y',Z'. Note that this abstract mapping produces outputs which are the same or smaller than the corresponding inputs(i.e. it is *contracting*) , it is jointly *monotone* with respect to set inclusion (i.e. *inclusion isotone*) and it is *idempotent*. Similar arguments apply to the other interval relations, so these properies apply to them as well.

In terms of the previously defined interval functions the result of this procedure can be written simply as:

```
$add( X, Y, Z, NewX, NewY, NewZ):-
            NewZ := Z ^ ( X + Y),
            NewX := X ^ ( Z - Y),
            NewY := Y ^ ( Z - X).
```

From the logic point of view, the interval query expression Z==X + Y is taken as having implicit existential quanitfiers (as usual). The procedure outlined above trims the intervals of any points (x,y,z) which can be proved *not* to be solutions. Hence, *if* there are solutions in the initial ranges, *then* they will also be in the final ranges. (Whether there are any solutions or not will, in general, depend ultimately on some form of the completeness axiom for the real numbers, and hence goes well beyond the competence of the system, although knowledgeable users may make such inferences.)

The most complex operation is multiplication, which must be broken into a number of cases because of the discontinuity at 0. By using symmetry, the 27 cases can be reduced to just 3 essentially different ones.

```
$mul( X, Y, Z, X, Y, NewZ) :-    % either X or Y exactly 0
      ( zero(X);zero(Y))  ,  zero( NewZ),!.

zero([0.0,0.0]).

$mul( X, Y, Z, NewX, NewY, NewZ) :-
      non_neg( Px, X, XP),                    % flip into + if possible
      non_neg( Py, Y, YP),                    % ditto
      Pz is Px*Py,                      % sign logic
      switch( Pz, Z, ZP),               % flip Z if necessary
      multcase(Pz,  Px,Py,F),!,         %  case-by-case
      F( XP, YP, ZP, NXP, NYP, NZP),
      switch( Px,NXP, NewX),            % switch back if flipped
      switch( Py,NYP, NewY),
      switch( Pz,NZP, NewZ).

non_neg( 1,[XL,XH], [XL,XH]) :- XL >= 0 ,!.
non_neg(-1,[XL,XH], [YL,YH]) :- XH =<0,YL is -XH, YH is - XL,!.
non_neg( 0,[XL,XH], [XL,XH])  .

switch(-1,[XL,XH], [YL,YH]) :- YL is -XH, YH is - XL.
switch( 1, X, X).
switch( 0, X, X) .

%  multcase(Pz,Px,Py,  Use)
multcase( 0, 0, 0, $mulC).    %  contains origin in interior
multcase( 1, _, _, $mulA).    % first quadrant
multcase(-1, _, _, $mulA).
multcase( 0, _, 0, $mulBy).   % right half
multcase( 0, 0, _, $mulBx).   % top  half

$mulA( X, Y, Z, NewX, NewY, NewZ):-  % all first quadrant
      NewZ := Z ^ ( X * Y),
      NewX := X ^ ( Z / Y),
      NewY := Y ^ ( Z / X).
```

```
%                            $mulBx:  X spans 0, Y non-negative
$mulBx( [XL,XH],    Y,  [ZL,ZH], NewX, NewY, NewZ):-   ZL >= 0,!,
        $mulA(  [0.0,XH],Y,[ZL,ZH],  NewX,  NewY,  NewZ).

$mulBx( [XL,XH],    Y,  [ZL,ZH], NewX, NewY, NewZ):-   ZH =< 0,!,
        $mulA(  [XL,0.0],Y,[ZL,ZH],  NewX,  NewY,  NewZ).

$mulBx(  [XL,XH],  [YL,YH],  [ZL,ZH],  NewX,  NewY,  NewZ):-
        NewZ  :=  [ZL,ZH]^[XL*YH,  XH*YH],
        YL>0  ->  NewX  :=  [XL,XH]^[ZL/YL,ZH/YL].

$mulBy( X,Y,Z,  NewX,  NewY,  NewZ):-   % Y  spans  0,  X  non-negative
        $mulBx(  Y,  X,  Z,  NewY,  NewX,  NewZ).

$mulC( [XL,XH],  [YL,YH],  Z,  NewX,  NewY,NewZ)  :-  %  all  span  zero
        NewZ  :=  Z  ^  [min(  XL*YH,  XH*YL),max(  XL*YL,  XH*YH)].
```

The "C" case (all signs unknown) does not do very much ; in particular it does not update either factor. (Note that if "new" variables are left uninstantiated they count as unchanged values in the update procedure.) There is actually a fourth case corresponding to the product having a known sign (possibly 0) but the signs of the factors unknown. This case can only be handled by exploring alternatives (e.g. both factors positive OR both negative).

DATA STRUCTURES AND ITERATION

The natural data structure for keeping interval constraints is a bipartite graph or network in which interval nodes alternate with operation nodes. The obvious Prolog implementation (in those Prologs which allow it) is as a cyclic structure. The constructor for interval nodes is given by:

```
interval_obj(  int(I,[  V,_Values..],Nodelist)):-
      unique_identifier(I),
      universal_interval( V).     % initial  first  value
```

Each interval node consists of a sequence of interval values (a monotone inclusion-decreasing sequence in fact), and a list of nodes. In addition, since logically distinct intervals must be kept distinct even if they happen to have the same contents (e.g. just after creation, when the nodelist is empty and the the value sequence contains only the universal range, an id-less interval would unify with any other interval), a unique identification label is attached to each new node. The uniqueness requirement is very local (it suffices if it discriminates among the nodes in the system at any one time) but *essential.*

The operation node constructor is given by:

```
node(   N  ):-  N=Op(  int(_,_,NX),int(_,_,NY),int(_,_,NZ)  ),
      ternary_relation(Op),
      new_member(  N,  NX),
      new_member(  N,  NY),
      new_member(  N,  NZ).

node(   N  ):-  N=Op(  int(_,_,NX),int(_,_,NY)  ),
      binary_relation(Op),
      new_member(  N,  NX),
      new_member(  N,  NY).
```

The cyclic structures are formed here when node N (which contains/ references interval objects) is put onto the nodelists of its intervals. The `new_member` predicate is essentially an optimized version of `[member(N,L),!]` and is given in BNR Prolog by:

```
new_member(X,[List..]):-   termlength(List,_,[X,_..]).
```

(There is a subtle difference, however: `[member !]` would not actually add a duplicate node to the lists, while `new_member` essentially *assumes* that the item is to be regarded as distinct.)

The interval object data structure is accessed in only two ways, one to fetch the "current" value of its range and one to update it. The latter operation is logically describable as a `[member(New,Val_seq) ,!]` to the indefinite list of ranges; thus if the new value is already on the list (i.e. the last item) the list need not be extended. If the list *is* extended, however, then the associated nodes need to be "informed" of the change to one of their inputs. This "broadcast" is simulated by keeping an indefinite scheduling list (`Agenda`) and adding nodes to it using `[member(..),!]`. In this case, using `member` has the important effect of *not* putting a node on the list if it is already there. A BNR Prolog program for doing get/put of range values is:

```
                                    % ( use Agenda =[] for get-only)
current_value(  int(_,V,NL),   [L,U],   Agenda):-
       termlength(V,N,Tail),      % see  BNR  Prolog  Ref.  manula
       % GET  last  value  XOR   PUT  new  value
       arg(N,V,[L,U])      ;    [ L =< U,        % check  for  null  interval
                                 Tail =[[L,U],_..],        % append
                                 subset(NL,  Agenda) ],  %  broadcast
       !.          %   just  one   branch

subset(   [X..],_):-   tailvar(X..),!.
subset(  [X,Xs..],  List):-  member(  X,  List),!,  subset(  Xs,  List).
```

The use of this can be seen best in the general setup and execute routine for operation nodes:

```
 donode(  Op(  X,  Y,  Z  )  ,    Agenda):-
       current_value(X,XV,    []),
       current_value(Y,YV,    []),
       current_value(Z,ZV,    []),
       Op(  XV,  YV,  ZV,  NewX,  NewY,  NewZ),
       current_value(  X,  NewX,     Agenda),
       current_value(  Y,  NewY,     Agenda),
       current_value(  Z,  NewZ,     Agenda).

donode(  Op(  X,  Y  )  ,    Agenda):-
       current_value(X,XV,    []),
       current_value(Y,YV,    []),
       Op(  XV,  YV,  NewX,  NewY),
       current_value(  X,  NewX,   Agenda),
       current_value(  Y,  NewY,   Agenda).
```

Finally, the fixed point iteration code is given by:

```
stable([]):-!.            %  terminate  when  agenda  comes  to  end
stable([Node,    Agenda..]):-
       donode(  Node,[Node,  Agenda..]),  %  note:Node  still   on  agenda
       stable(   Agenda).
```

Here the cut in the first clause is necessary because the list is an indefinite one, and the second clause would still apply even when the "end" is reached. Note that in the second clause the agenda with the currently executing node on it is passed as an argument to the currently running node operation, so a node won't trigger its own reexecution.

Most of the theoretical questions center around the `stable` predicate.

SETTING UP CONSTRAINTS

We will describe the final part of the system, the simplified user interface.The first statement to consider is the type declaration : which is used to create intervals:

```
X:real  :-  !,  interval_obj(X).

X:real(LB,UB):-
      interval_obj(  X),              %  test  for/create  interval .
      current_value(  X,  [CL,CU],  []),
      [LBI,UBI]:=  [CL,CU]  ^  [LB,UB],
      current_value(  X,  [LBI,UBI],  Agenda),
      stable(  Agenda).               %  process  any  repercussions
```

Note that if the range is set it may trigger a cascade of additional narrowings; if the setting is inconsistent the call to `stable` may fail.

The rest of the major arithmetic relations are handled (so far as intervals are concerned) by:

```
{Y  is  X}  :-  interval_obj(Y),  $arith_rel(  $eq,  Y,  X).
{Y  ==  X}  :-  $arith_rel(  $eq,  Y,  X).
{Y  =<  X}  :-  $arith_rel(  $le,  Y,  X).
{Y  >=  X}  :-  $arith_rel(  $le,  X,  Y).

$arith_rel(F,Expression1,   Expression2):-
      evaluate(  Expression1,  E1,    Agenda),
      evaluate(  Expression2,  E2,    Agenda),
      node(  F(E1,E2)  ),                        %  make  a  node
      new_member(  F(E1,E2),    Agenda),         %  add  it  to  agenda
      stable(  Agenda).                          %  fixed  point  iteration
```

The interval evaluation procedure is essentially a mapping from the usual tree expression to the constraint network. (Again, we ignore the addiitonal code required to implement non-interval arithmetic expressions.) The first three cases handle the leaves of the tree:

```
evaluate(X,_,_)   :-   var(X),!,X:real.%  implicit  interval  creation
evaluate(  X,  X,  _):-       interval_obj(X),!.
evaluate(  X,  XE,_):-        point_interval(  X,XE),!.
```

The first of these clauses is to create an interval when an uninstantiated variable is discovered in an arithmetic expression. A simple implementation for point interval is

```
point_interval(  X,  XE):-
      coerce(X,R),
      interval_obj(XE),
      current_value(XE,[R,R],[]).    %  assert:  nodelist  is  empty

coerce(  X,X):-float(X).
coerce(  X,Y):-integer(X),  Y  is  float(X).
```

The actual implementation in BNR Prolog differs from this in that the endpoints are not identical. This is an attempt to work around the problems caused by using binary internal representations which, in general, cannot represent finite decimal numbers exactly.

Finally, the recursion step in evaluation is given by:

```
evaluate( F(X,Y), ZE,   Agenda):-
      $fmap( ZE is F(XE,YE), M),
      evaluate( X, XE,   Agenda),
      evaluate( Y, YE,   Agenda),
      interval_obj(ZE),
      node( M),
      new_member(M,  Agenda).

$fmap( Z is X + X,      $mul(Two,X,Z)):- point_interval( 2, Two),!.
$fmap( Z is X + Y,      $add(X,Y,Z)).
$fmap( Z is X - X,      $eq(Zero,Z) ):- point_interval( 0, Zero),!.
$fmap( Z is X - Y,      $add(Z,Y,X)).
$fmap( Z is X * Y,      $mul(X,Y,Z)).
$fmap( Z is X/Y   ,     $mul(Z,Y,X)).
$fmap( Z  is  min(X,Y), $min(X,Y,Z)).
$fmap( Z  is  max(X,Y), $max(X,Y,Z)).
```

We have omitted the functions `midpoint`, `median`, and `delta` which take intervals as arguments but which return floating point values. Also omitted are the integer and boolean types and their primitives and transcendental functions and special utilities such as `accumulate`. Finally, the recently added new predicates `lower_bound` and `upper_bound` are given by:

```
lower_bound( X):-    interval_obj(X),
      current_value(X,[L,U],[]),
      current_value(X,[L,L],  Agenda),
      stable(  Agenda).

upper_bound( X):-    interval_obj(X),
      current_value(X,[L,U],[]),
      current_value(X,[U,U],  Agenda),
      stable(  Agenda).
```

EXAMPLES

Some examples of execution traces can help to give a feel for the dynamics. This first example illustrates "waves" of change sweeping across a small network. Note how the lower bound of 1 is propagated incrementally, while the upper bound 5 is propagated backwards "all at once". When these two networks are then bridged by the final equality, there are two update waves which get processed in parallel. Notice also the final bounce off the boundary conditions.

```
problem:
      X:real,   Y:real,U:real,V:real,
      1 =< X,   X =< Y,
      U =< V, V =< 5,
      Y == U

trace:
   X:real    %   _Interval_984396
   Y:real    %   _Interval_985196
   U:real    %   _Interval_985996
   V:real    %   _Interval_986796
      =<  -->  $le(_Interval_987736,_Interval_984396)  %  first  =<
      node:  $le(_Interval_987736,_Interval_984396)
            _Interval_984396    <-    [1.0,3.4000e+30]    % X changes

      =<  -->  $le(_Interval_984396,_Interval_985196)
      node:  $le(_Interval_984396,_Interval_985196)
            _Interval_985196    <-  [1.0,3.4000e+30]     % Y changes

      =<  -->  $le(_Interval_985996,_Interval_986796)
      node:  $le(_Interval_985996,_Interval_986796)     % no change

      =<  -->  $le(_Interval_986796,_Interval_992976)
      node:  $le(_Interval_986796,_Interval_992976)
            _Interval_986796    <-  [-3.4000e+30,5.0]    % change V
      node:  $le(_Interval_985996,_Interval_986796)
            _Interval_985996    <-  [-3.4000e+30,5.0]    % then U

      ==  -->  $eq(_Interval_985196,_Interval_985996)
      node:  $eq(_Interval_985196,_Interval_985996)
            _Interval_985196    <-  [1.0,5.0]           % change Y
            _Interval_985996    <-  [1.0,5.0]           % change U
      node:  $le(_Interval_984396,_Interval_985196)
            _Interval_984396    <-  [1.0,5.0]           % change X
      node:  $le(_Interval_985996,_Interval_986796)
            _Interval_986796    <-  [1.0,5.0]           % change V
      node:  $le(_Interval_987736,_Interval_984396)
      node:  $le(_Interval_986796,_Interval_992976)

final  results:
_Interval_984396:[1.0,5.0]
_Interval_985196:[1.0,5.0]
_Interval_985996:[1.0,5.0]
_Interval_986796:[1.0,5.0]
```

The next example illustrates function evaluation. Note that information generally passes from narrower to wider intervals; in the case of function evaluation this is naturally the same direction that conventional computation takes.

```
problem:     X:real(-100,100),     Y:real,
             Y == -6 + X * (X - 1),
             X == 2

trace
      X:real(-100,100)
      _Interval_984760     <-   [-100.0,100.0]
      Y:real
      -     -->       $add(_Interval_987568,_Interval_987348,_Interval_984760)
      *     -->       $mul(_Interval_984760,_Interval_987568,_Interval_988220)
      +     -->       $add(_Interval_986728,_Interval_988220,_Interval_988872)
      ==   -->   $eq(_Interval_985644,_Interval_988872)
      node:   $add(_Interval_987568,_Interval_987348,_Interval_984760)
              _Interval_987568      <-   [-101.0,99.0
      node:   $mul(_Interval_984760,_Interval_987568,_Interval_988220)
              _Interval_988220       <-   [-1.0100e+4,1.0100e+4]
      node:   $add(_Interval_986728,_Interval_988220,_Interval_988872)
              _Interval_988872       <-   [-1.0106e+4,1.0094e+4
      node:   $eq(_Interval_985644,_Interval_988872)
              _Interval_985644       <-   [-1.0106e+4,1.0094e+4]

      % now do the X==2
      ==   -->   $eq(_Interval_984760,_Interval_995740)
      node:   $eq(_Interval_984760,_Interval_995740)
              _Interval_984760      <-  [2.0,2.0]
      node:   $add(_Interval_987568,_Interval_987348,_Interval_984760)
              _Interval_987568      <-  [1.0,1.0]
      node:   $mul(_Interval_984760,_Interval_987568,_Interval_988220)
              _Interval_988220      <-  [2.0,2.0]
      node:   $add(_Interval_986728,_Interval_988220,_Interval_988872)
              _Interval_988872      <-  [-4.0,-4.0]
      node:   $eq(_Interval_985644,_Interval_988872)
              _Interval_985644      <-  [-4.0,-4.0]

final  results:
_Interval_984760:[2.0,2.0]
_Interval_985644:[-4.0,-4.0]
```

The last test illustrates fixed point iterations:

```
problem:
      X:real(-100,100),Y:real
      Y == -6 + X * (_X - 1),
      Y == 0,
      X >= 0

trace:
      X:real(-100,100)
```

```
            _Interval_984560     <-   [-100.0,100.0]
       Y:real
       -      -->      $add(_Interval_987368,_Interval_987148,_Interval_984560)
       *      -->      $mul(_Interval_984560,_Interval_987368,_Interval_988020)
       +      -->      $add(_Interval_986528,_Interval_988020,_Interval_988672)
       ==   -->   $eq(_Interval_985444,_Interval_988672)
       node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
            _Interval_987368     <-   [-101.0,99.0]
       node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
            _Interval_988020     <-   [-1.0100e+4,1.0100e+4]
       node:  $add(_Interval_986528,_Interval_988020,_Interval_988672)
            _Interval_988672     <-   [-1.0106e+4,1.0094e+4
       node:  $eq(_Interval_985444,_Interval_988672)
            _Interval_985444     <-   [-1.0106e+4,1.0094e+4]
% network has now been constructed

%  setting Y==0: this shows computation coming "back up" the evaluation
%  tree, opposite to the conventional direction
       ==   -->   $eq(_Interval_985444,_Interval_995540)
       node:  $eq(_Interval_985444,_Interval_995540)
            _Interval_985444    <-  [0.0,0.0]
       node:  $eq(_Interval_985444,_Interval_988672)
            _Interval_988672    <-  [0.0,0.0]
       node:  $add(_Interval_986528,_Interval_988020,_Interval_988672)
            _Interval_988020    <-  [6.0,6.0]
       node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
% iteration stops here- the $mulC case mentioned previously
% the multiplication is 6==X*(X-1), but both X and (X-1) span 0

% X >= 0  ; this breaks the impasse
       >=   -->   $le(_Interval_1001820,_Interval_984560)
       node:  $le(_Interval_1001820,_Interval_984560)
            _Interval_984560     <-  [0.0,100.0]
       node:  $add(_Interval_987368,_Interval_987148,_Interval_984560]
            _Interval_987368     <-  [-1.0,99.0]
       node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
            _Interval_984560     <-   [0.060606,100.0]
            _Interval_987368     <-   [0.06,99.0]
       node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
            _Interval_984560     <-  [1.06,100.0]
       node:  $le(_Interval_1001820,_Interval_984560)
%  note that $le is never going to do anything but gets scheduled
%  just in case; some $le cases are naturDally persistent, but we
%  aren't taking advantage of it
       node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
            _Interval_987368     <-   [0.06,5.6604]
% note that X -1 has now narrowed substantially
       node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
            _Interval_987368     <-   [0.06,5.6604]
            _Interval_984560     <-   [1.06,6.6604]
       node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
            _Interval_987368     <-   [0.90085,5.6604]
       node:  $le(_Interval_1001820,_Interval_984560)
%  note we are now in a loop with intervals decreasing
       node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
            _Interval_984560     <-   [1.9009,6.6604]
```

```
        node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
               _Interval_987368      <-  [0.90085,3.1565]
        node:  $le(_Interval_1001820,_Interval_984560)
%   each stage is able to eliminate more  points in a sort of
%   inductive proof
        node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
               _Interval_984560      <-  [1.9009,4.1565]
        node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
               _Interval_987368      <-  [1.4435,3.1565]
        node:  $le(_Interval_1001820,_Interval_984560)
%     convergence is slow, comparable with regula  falsi
        node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
               _Interval_984560      <-  [2.4435,4.1565]
        node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
               _Interval_987368      <-  [1.4435,2.4555]
        node:  $le(_Interval_1001820,_Interval_984560)

        node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
               _Interval_984560      <-  [2.4435,3.4555]
        node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
               _Interval_987368      <-  [1.7364,2.4555]
        node:  $le(_Interval_1001820,_Interval_984560)


  ......
% next cycle is different- both factors narrow in the multiplication
% ( printing precision does not show the difference )
        node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
               _Interval_984560      <-  [2.9757,3.0163]
        node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
               _Interval_984560      <-  [2.9757,3.0163]
               _Interval_987368      <-  [1.9892,2.0163]
        node:  $le(_Interval_1001820,_Interval_984560)

.......
        node:  $add(_Interval_987368,_Interval_987148,_Interval_984560)
               _Interval_984560      <-  [2.999,3.0006]
        node:  $mul(_Interval_984560,_Interval_987368,_Interval_988020)
               _Interval_984560      <-  [2.999,3.0006]
               _Interval_987368      <-  [1.9996,2.0006]

% at this point the changes become invisible because of printing
% precision limitations, but the iteration finally stops with the
% correct answers printed as 2.0 and 3.0. Internally, these are very
% small pointlike intervals, the width determined by the implementation
% of point_interval and the precsion of arithmetic used in $isL and
% $isH.
```

BIBLIOGRAPHY

BNR Prolog (1988) *User Guide* and *Reference Manual.*

Cleary, J. C. (1987) "Logical Arithmetic", Future Computing Systems, Volume 2, Number 2, pp.125-149.

Davis, E.(1987) "Constraint propagation with interval labels" *Artificial Intelligence* 32 pp. 281-331.

Moore, R. E.(Ed.) (1966) *Interval Analysis*, Prentice Hall, New Jersey.

Moore, R. E.(Ed.) (1988) *Reliability in Computing* (The role of Interval Methods in Scientific Computing) Perspectives in Computing; Vol. 19, Academic Press.

Older, W. and Vellino, A. (1989) "Extending Prolog with Constraint Arithmetic on Real Intervals", CRL Doc#89023 .

Vuillemin J., (1987) "Exact real computer arithmetic with continued fractions", Institut National de Recherche en Informatique et Automatique (INRA).

APPENDIX A: THEORY

This appendix is concerned with the proofs of properties of the fixed point iteration operator represented by the `stable` predicate. The properties in question are the existence of fixed points, independence of scheduling order, monotonicity and termination. Partial correctness is already established since the primitive operations never discard points except when there is a proof that they contain no consistent solutions.

The proof is based on establishing an appropriate partially-ordered state space, based on topological considerations, and then employing simple lattice-theoretic arguments.

STATE SPACE

?

The first thing to note about this operator is that it does not change the structure of the constraint network, but at most alters the current values of some intervals. Thus the state can be construed as a fixed set of interval objects each with a current value which is an interval range. Formally, let $\mathcal{I}$ be the set of interval ranges (including the empty range) over the *reals*, $\mathcal{O}$ be the finite set of interval objects, and V be the map V:$\mathcal{O}$ -> $\mathcal{I}$ defining the current value. Then the state is defined as $S = \times_{i \in \mathcal{O}} V(i)$, the Cartesian product of the current values. The state space $\boldsymbol{L}$ is defined to be all such Cartesian products, partially ordered by inclusion. (For simplicity, we will consider U as fixed and have suppressed it from the notation.) The largest element (U*) in the partial order is the n-fold product of the universal interval U; the smallest is the null set. Since ranges are closed intervals, they are compact topological spaces and therefore so are all their Cartesian products. In particular, U* as a topological space is compact and Hausdorff (with the usual induced topology from the reals) , and therefore all states are closed sets. Hence the arbitrary intersections of states are all closed compact sets and in fact states.

It follows that since $\boldsymbol{L}$ is closed under arbitrary intersections, and since there is a largest element, a join operation $\vee$ can be defined (by intersecting all elements bigger than all the items in a given collection), and $\boldsymbol{L}$ is a complete lattice. (Since meets in the lattice are just set intersections, we will generally use "intersection" for "meet" in the following and refer to the elements of $\boldsymbol{L}$ as "sets"; however, the lattice we are dealing with is *not* the usual power set lattice of U since the join is not the set union.)

$\boldsymbol{L}$ is also intersection-compact, i.e. if the intersection of an arbitrary collection of sets is empty, then there is a finite subcollection which has empty intersection. (This is just the compactness of U* restated in the finite-intersection property form.) For inconsistent sets of constraints, which eventually end in failure, this guarantees that the failure occurs after a finite number of steps.

CONTRACTIONS ON *L*

Now let us turn to the operations, i.e. the maps p: *L* -> *L* . The primitive operations $le, $add, etc. all had the following properties (with respect to the space spanned by their input arguments):

a.	contracting	$Y \supseteq p(Y)$
b.	monotone (isotone)	$X \supseteq Y$ implies $p(X) \supseteq p(Y)$
c.	idempotent	$p(p(X)) = p(X)$.

(Note that failure is regarded as mapping to the null set.) When applied to a selected two or three dimensions out of *L* , all others being left unchanged, these properties carry over directly to similar statements over *L* .

Given two maps p,q: *L* ->*L* with these properties, the composite maps pq and qp are obviously contracting, but need not be idempotent. (Recall the last example in the main text.) To investigate this question, a good place to start is the space of all contraction maps over *L* , which we will denote by C:

def C := { p :*L* ->*L* | p is contraction }.

C inherits a partial order and lattice structure from *L* in the usual way, e.g

$p \supseteq q$ iff for all X in *L* $p(X) \supseteq q(X)$

$(p \cap q)(X) := p(X) \cap q(X)$ for all X in *L*

The bottom element is the null map 0 (the analog of "fail" in Prolog); the top element is the identity map 1.

C is also a semigroup under function composition with 1 as two-sided identity and 0 as a two-sided zero element:

$1p = p$
$p1 = p$
$0p = 0$
$p0 = 0$

Only the last of these depends on the maps being contractions.

The lattice and semigroup structures are tied together by two basic properties:

for all p,q in C : $q \supseteq pq$
for all p,q,r in C : $p \supseteq q$ implies $pr \supseteq qr$.

The first merely restates that p is a contraction; the second follows from the definition of the partial order relation. Note that the second states that the semigroup operation is monotone in the first variable.

There are a couple of useful results on idempotents in C. If p and q are idempotents in any semigroup, and p commutes with q, then pq=qp is also an idempotent, since

$$(pq)(pq) = p(qp)q = ppqq = pq.$$

Second, given any contraction p then $p^* := \lim p^n$ exists. To show this, fix an arbitrary set X; then $X \supseteq p(X) \supseteq pp(X) \supseteq \ldots$; this monotonically decreasing sequence has a limit since the lattice L is complete, and this defines the action of p* on X. Since X is arbitrary, p* is well defined. From the method of construction, it is clear that p* is idempotent. Given two idempotents, p,q, we can then define $p^*q := (pq)^*$; by the preceding comments, this operation is a well defined operation on the set of idempotent contractions.

If p and q are idempotents, then define the (Green's) relations by

p L q iff p =pq
p R q iff p =qp
p S q iff p=qp=pq (i.e. iff p L q and p R q).

These relations capture the idea of one idempotent p being "stronger" than q. Note that all three are trivially reflexive. L is transitive, since p L q and q L r implies p=pq and q=qr, so $pr= (pq)r=p(qr)=pq=p$.

R is transitive by a similar proof, and the transitivity of S follows. Since pSq and qSp implies p=qp and q=qp implies p=q, S is a partial order on C , and pSq => $q \supseteq p$ since p is a contraction. If i is idempotent and iLp, then iLp^n, and iLp* and similarly for R and S. Hence, p* is the weakest idempotent stronger than p.

MONOTONE CONTRACTIONS

To get a significantly stronger connection between the semigroup and lattice structures, we need to specialize to **CM**, the space of monotone contractions.

def **CM** := { p:L ->L | X$\supseteq$ p(X) and X$\supseteq$Y => p(X) $\supseteq$ p(Y)}.

(Since all the basic node operations are monotone, this is a reasonable restriction.) Since the composition of two monotone maps is monotone, **CM** is a subsemigroup of **C**. The intersection of monotone maps is also monotone, since intersection is itself monotone. Since 0 and 1 are both monotone maps, **CM** also inherits the lattice and monoid properties as well. Monotonicity (as maps on L) provides for symmetrical relationships between the product and order structures:

for all p,q in **C** : q $\supseteq$ qp
for all p,q,r in **C** : p $\supseteq$ q implies rp $\supseteq$ rq .

Note that the product is now monotone in the second argument as well as the first. A simple proof then shows that the product is in fact jointly monotone in both arguments. From this it follows that p $\supseteq$ q implies pp $\supseteq$ qq and p* $\supseteq$ q*. If p is monotone, then so is p*.

Suppose p,q are idempotents in **CM**. Since p is a contraction, q $\supseteq$ pq, so pLq implies q $\supseteq$ p. Conversely, if q $\supseteq$ p, then since p is monotone, pq $\supseteq$ pp = p, but since q is a contraction 1 $\supseteq$ q and p $\supseteq$ pq, so p=pq and pLq. Similarly, pRq iff q $\supseteq$ p and pSq iff q $\supseteq$ p. Hence these partial orders all collapse into the state-induced partial order.

Since q $\supseteq$ qp and p $\supseteq$ qp, it follows that p $\cap$ q $\supseteq$ qp ; similarly, p $\cap$ q $\supseteq$ pq. By induction, $(pq)^{n-1} \supseteq (qp)^n$ and $(qp)^{n-1} \supseteq (pq)^n$, so $(pq)^{n-1} \cap (qp)^{n-1} \supseteq (qp)^n$ and $(pq)^{n-1} \cap (qp)^{n-1} \supseteq (pq)^n$. Thus these sequences are mutually cofinal and have the same limit: p*q=(pq)*=(qp)*=q*p. Hence, regarded as an operation on idempotents, * is commutative.

If p and q are idemptents in **CM**: since pa nd q are contractions then X $\supseteq$ p(X) and X $\supseteq$ q(X), so X $\supseteq$ p(X) $\vee$ q(X) $\supseteq$ p(X), and then p(X) $\supseteq$ p(p(X) $\vee$ q(X)) $\supseteq$ p(p(X))=p(X), and hence p(X) = p(p(X) $\vee$ q(X)) and similarly q(X) = q(p(X) $\vee$ q(X)) Then (p $\vee$ q)(p $\vee$ q)(X) =(p $\vee$ q)(p(X) $\vee$ q(X)) = p(p(X) $\vee$ q(X)) $\vee$ q(p(X) $\vee$ q(X)) =p(X) $\vee$ q(X) = (p$\vee$q)(X), so the join of idempotents is idempotent. This can be generalized to arbitrary joins.

If p,q are idempotents in **CM** , then X is a fixed point of p*q iff X is a fixed point of p and X is a fixed point of q. Obviously, if X is a fixed point of both p and q, (p*q)(X)=(pq)*(X)=pq(X)=X. Coversely, if X is a fixed point of p*q then X$\supseteq$ pq(X) $\supseteq$ (p*q)(X) =X, so p(X)==pq(X)=X and similarly for q.

Fixed Points

From previous results, p*q is monotone and idempotent, p*q S p, p*q S p, and if r is a monotone idempotent with rSp, rSq, then r S p*q. Thus the binary operation * constructs the greatest lower bound with respect to the S partial order. It follows, by the usual lattice arguments, that * is an associative as well as commutative operation. Operationally, this implies that the final state of the fixed point iteration is independent of the order in which the individual primitive operations are done.

An alternate approach is to define the set of fixed "points" associated with an idempotent in CM: **F**(p) = { X | p(X)=X}. The set of fixed points contains the bottom state and is closed under arbitrary joins.

For any join-closed family F of states containing bottom, define **G**(F)(X)= to be the largest Y in F such that $X \supseteq Y$. (This is unique because of the join-closed property.) Then **G**(F) is easily shown to be an idempotent monotone contraction.

The maps **F**,**G** satisfy **G**(**F**(p))=p and **F**(**G**(A)=A, and form a bijective correspondence between narrowing operators (idempotent monotone contractions) and join-closed subsets of stable states. Note that from the result of the last section, **F**(p*q) = **F**(p)$\cap$**F**(q).

Compactness

We have noted earlier that compactness implies finite termination of all iterations which eventually fail. Non-failing iterations can, in principle, continue indefinitely over the reals; each such monotonically decreasing sequence of intervals contains a real number by Cantor's nested set theorem. However, in the implementation the precision of the arithmetic is limited by the floating point machinery; the floating point representation space being finite, the aproximation sequences converge after a finite number of operations.

The Application of CLP(BNR) to Powder Method X-ray Diffraction Crystallography

William J. Older

February 1995

ABSTRACT

This paper illustrates the use of relational interval arithmetic in CLP(BNR) for the determination of primitive cell structure from powder method X-ray diffraction data. It serves to illustrate the way in which a uniform declarative programming style can be employed for problems involving a search over a space of models parameterized by both discrete and continuous parameters.

INTRODUCTION

The techniques natural to Prolog programming are particularly suitable for problems involving searches over discrete spaces. Simple algorithms to do exhaustive search can be written very easily in Prolog by taking advantage of its natural backtracking ability, and more efficient search algorithms can usually be obtained by manipulating the order of generation of candidates, arranging for early failure, or otherwise constraining the search.

Traditionally, however, the treatment of searches over spaces parameterized by continuous variables has been limited by the conventional nature of the floating point arithmetic available. Since continuous spaces cannot practically be aproximated by discrete spaces, the advantages of the declarative style are lost and one is forced to treat such problems in ways that would be familiar to the Fortran programmer.

The introduction of relational interval arithmetic in CLP(BNR) now makes it possible to adopt the declarative style in problems involving continuous variables. This is particularly convenient in problems where both discrete and continuous parameters are present, since it allows both to be treated with the same paradigm. To illustrate this point we present a simplified version of a typical problem of this sort, and show how a declarative Prolog solution can be expressed using interval arithmetic.

THE PROBLEM

The problem is that of identifying the shape of the *primitive cell* of a crystal lattice from X-ray diffraction data. A primitive cell can be characterized by six continuous parameters, which may be taken as the lengths of three non-coplanar vectors **a,b,c** and the angles between them. When X-rays interact with a crystal lattice, the phenomenon of Bragg diffraction causes scattering of the X-ray beam at specific angles Θ depending on the wavelength λ of the X-rays and the spacing d between the planes of the crystal lattice, according to the well-known Bragg equation:

$$n\,\lambda = 2d \sin \Theta.$$

In the powder method, a crystalline sample is ground to powder and the powder mixed with glue is mounted in an X-ray beam, so that the scattered rays impinge on a detector such as a photographic film. Individual crystals then take many different orientations relative to the optical axis of the beam, thus averaging over all orientations. The resulting image at a fixed radius shows a spectrum which is a function of the interplanar distances and the effective X-ray wavelength. The problem is then to determine the correct shape of the primitive cell which would account for all the observed lines in the spectrum.

The mathematical formulation of the problem is simplest when expressed in terms of the parameters of the *reciprocal lattice,* the dual basis in Fourier space. The squared distances from the arbitrarily chosen origin to the lattice points (h,k,l) (with h,k,l integers) can be related directly to the observed line spacings, given the X-ray wavelength and test-equipment geometry. Thus given some initial transformation of the raw data to squared distances, the essential problem is to infer the cell parameters from the pattern of squared distances in the lattice. This is illustrated schematically in figure 1 below,, which shows the two-dimensional case.

This is an example of an *inverse problem* with a discrete labeling component: the forward problem of computing the distances given cell parameters and line labeling is trivial. Likewise inferring cell parameters from known labeling and distances, although not so trivial, is relatively straightforward by conventional techniques. With both the cell parameters and labeling unknown, however, it is difficult to find a way to apply conventional techniques. One is forced to adopt an approach which guesses tentative labelings until there enough to determine cell parameters, and to do this systematically until the right guess is made, and this is difficult to do correctly using conventional techniques.

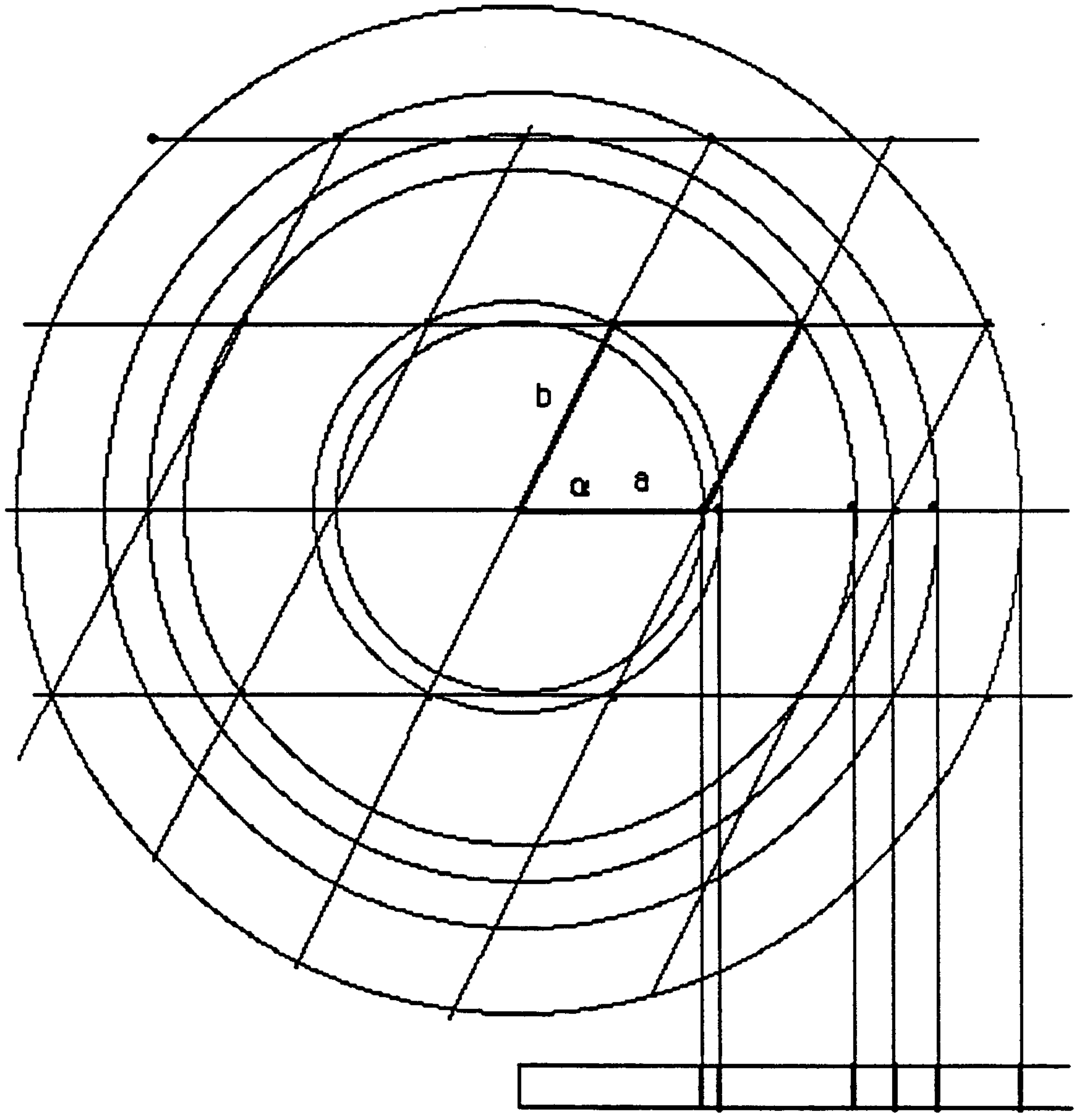


Fig.1.-Reciprocal lattice and observed spectrum

For any fixed lattice point (h,k,l), there is a simple quadratic formula for computing the squared distance as a function of the cell parameters:

```
    distance2(  [H,K,L],   [A,B,C,CosA,CosB,CosC],   D2):-
        { D2  is (H*A)**2 + (K*B)**2 + (L*C)**2 +
              2*(H*K*A*B*CosC + K*L*B*C*CosA   + L*H*C*A*CosB) }.
```

where `A,B,C` are the lengths of the primitive vectors and where `CosC` is the cosine of the angle between vector **a** and vector **b** and so on. Since only the cosines of the cell angles appear in this formula, it is evidently more efficient to use these cosines instead of the angles themselves in our representation for a primitive cell.

The mathematical structure of the problem is now becoming clear: given the primitive cell shape there is a function to determine the squared distance to any particular lattice point. Furthermore, given any six sufficiently different lattice points (and knowing which ones they are) and their corresponding squared distances, one could solve the six simultaneous non-linear equations to determine the cell shape. (Solving such a system is, of course, not exactly trivial but there are traditional techniques for accomplishing it.) The essential difficulty is then that one does not know *a priori* which distance values correspond to which lattice points. This is complicated by the fact that there may be addiitonal lines present due to contamination of the sample, or the sample might just be a mixture. In this manner the continuous system identification problem has acquired a significant combinatorial complexity.

There is one more significant theoretical point to be made. The cell shape which we are seeking has been described as three lengths and three cosines. This description, however, is both imprecise and inadequate. To be more precise, we can choose a specific tuple representation such as that used above:

```
[ A, B, C, CosA, CosB, CosC]
```

In addition, however, we need to specify that these six parameters are continuous and indicate their initial domains. In CLP(BNR) this can be done by means of a type declaration. Since `A,B`, and `C` are distances, their domains are [0,infinity] and this would be written as

```
[A,B,C]:real(0,_)
```

This is not quite right, however, since it contains a hidden redundancy. Since we do not know which base vector `A,B`, or `C` is which, any permutation of them (together with the corresponding angles) would represent the same primitive cell, so there is a six-fold degeneracy in the representation. To remove this degeneracy, we can agree to always list them in order of increasing size. This can be added to the specification above, by adding the constraint:

```
{ A =< B, B=< C }
```

after the declarations.

The cosines must be treated similarly; since the angles between the vectors go from 0 to 180 degrees, the cosines would have initial domains [-1,1] and should be declared as such. However, this is also a redundant representation since the signs of any of the base vectors of the cell can be inverted (thus changing the signs of the cosines) without describing a different lattice at all. A symmetry analysis (which can be found in reference 1) shows that there are two classes of crystals depending on whether their primitive cells have an odd or even number of obtuse angles. By restricting ourselves to Type I crystals (those with an even number of obtuse angles), it is always possible to choose base vectors such that all the angles are acute, by reversing the sign of the base vector opposite the acute angle. Hence, for Type I crystals, the proper initial domain restrictions are:

```
[ CosA,CosB,CosC] : real(0,1) .
```

For Type II cells (with an odd number of obtuse angles), the declaration should correspondingly be:

```
[ CosA,CosB,CosC] : real(-1,0).
```

and to handle both possibilities together:

```
[ CosA,CosB,CosC] : real(-1,1), B:boolean,
 {  B==  (CosA>=0),B==(CosB>=0),B==(CosC>=0)}.
```

We will henceforth assume for simplicity that we are dealing with Type I crystals.

This specification can then be packaged as a Prolog "type description" which checks/creates instances of the type:

```
type_I_cell([A,B,C,CosA,CosB,CosC])  :-
      [A,B,C]  :  real(0,_),
      [CosA,CosB,CosC]  :  real(0,1),
      { A =< B,  B =< C }.
```

THE PROGRAM

The next step is to provide a generator for lattice points. It will be more efficient in the final algorithm if computed distances are aproximately ordered by size. One way to accomplish this is to generate lattice points [H,K,L] from the center out. For this one can simply use a set of facts lof the form isometric(N, [H,K,L]), ordered by N:

```
%  N is h**2 + k**2 +l**2
% defines search order in general case

isometric(  1,[1,0,0]).
isometric(  2,[1,1,0]).
isometric(  3,[1,1,1]).
isometric(  4,[2,0,0]).
isometric(  5,[2,1,0]).
isometric(  6,[2,1,1]).
isometric(  8,[2,2,0]).
isometric(  9,[3,0,0]).
isometric(  9,[2,2,1]).
isometric(  10,[3,1,0]).
isometric(  11,[3,1,1]).
isometric(  12,[2,2,2]).
isometric(  13,[3,2,0]).
isometric(  14,[3,2,1]).
isometric(  16,[4,0,0]).
isometric(  17,[4,1,0]).
isometric(  17,[3,2,2]).
isometric(  18,[4,1,1]).
isometric(  18,[3,3,0]).
etc.
```

This table lists one point of each distinct type in order of distance from the origin in a cubic lattice; it can easily be computed in Prolog to any desired limit by:

```
findset( isometric(N,  [H,K,L]  ),
      [integer_range(H,0,   Max),
        integer_range(K,0,   Max),
        integer_range(L,0,   Max),
       N is J*J + K*K + L*L
      ],
      List),
foreach(  member(X,List)  do  assert(X)),
```

For each such point all its distinct permutations are also lattice points:

```
distinct_perm( [H,K,L], [H,K,L]).
distinct_perm( [H,K,L], [H,L,K]):- K<>L.
distinct_perm( [H,K,L], [K,H,L]):- H<>K.
distinct_perm( [H,K,L], [L,K,H]):- L<>H.
distinct_perm( [H,K,L], [K,L,H]):- H<>K,K<>L,L<>H.
distinct_perm( [H,K,L], [L,H,K]):- H<>K,K<>L,L<>H.
```

Similarly, any non-zero coordinate can also occur with the opposite sign; the opposite point (with all signs reversed) is redundant and should be eliminated. This specification can then be written as

```
sign(  [H,K,L],   [H,K,L]).
sign(   [0,K,L],   [0,-K,L]):-K<>0,L<>0,!.
sign(   [H,0,L],   [H,0,-L]):-L<>0,H<>0,!.
sign(   [H,K,0],   [H,-K,0]):-H<>0,K<>0,!.
sign(  [H,K,L],   [-H,K,L]):-   H<>0,K<>0,L<>0.
sign(  [H,K,L],   [H,-K,L]):-   H<>0,K<>0,L<>0.
sign(  [H,K,L],   [H,K,-L]):-   H<>0,K<>0,L<>0.
```

The complete generator for lattice points is then:

```
hkl(    [H,K,L],  Limit):-
     integer_range(N,1,  Limit),
     isometric(N,HKL),
      distinct_perm(HKL,HKLU),
     sign(HKLU,[H,K,L]).
```

We will want to generate a list of squared distances to lattice points: one way to accomplish this is:

```
dist2_list(N_limit,   Cell,   List):-
      findall(  HKL,   hkl(HKL,N_limit),L),
      $dist2( L,  Cell,  List).

$dist2([],Cell,[]).
$dist2([HKL|Ls],Cell,[D2|Ds]):-
      distance2(HKL,  Cell,    D2),
      $dist2(Ls,Cell,   Ds).
```

where `Cell` is as defined previously. Converting the lattice indexes toa list first, and using list processing to compute distances, is necessary if we want to use this with `Cell` consisting of intervals (since we must have a conjunction of constraint equations).

We also need a predicate, essentially an arithmetic version of the well-known member predicate, to check if an arithmetic quantity is arithmetically equal to any on an ordered list of observational data. Since there are several sources of uncertainty in the experimental setup, such as the fact that the xray beam is a mixture of frequencies, the observed data is slightly fuzzy, and will therefore also be represented as interval values. Since it is conceptually misleading that the comparison with a predicted value should narrow the interval actually observed, the first clause makes use of the inclusion primitive '<=' of CLP{(BNR), which reads the data but does not change it. The second clause includes a constraint check that cuts off search once the proper place in the list is past.

```
op( 700, xfy, in).    % member for intervals

X in [D|Ds]:- { X <= D}. % inclusion
X in [D|Ds] :-{ X>=D }, X in Ds.
```

Finally, here is a simple version of the complete algorithm which takes the measured values of distance squared (as a list of intervals), and non-deterministically finds all reciprocal lattice parameters which can be found in the data. By formulating the problem this way, we can handle both mixtures and contaminated samples. (However, it does assume that we are not missing any data points, such as those too large or too small to have been captured.)

```
analyze( Limit, Cell, Observations) :-
      type_I_cell( Cell),
      dist2_list(Limit, Cell, Computed_list),
      allin( Computed_list, Observations).

allin([],_).
allin([X|Xs],L):- X in L, allin(Xs,L).
```

The declarative reading of this program is straightforward. The first line ensures that the reciprocal lattice parameters meet the requirements specified for a Type I lattice. The second line then computes a list of all the squared distances up to some order; the higher the order, the more likely that the answer will be unique (provided one is not looking for something beyond the end of the observational data). The third line then requires that all computed lines are among the observed ones.

An operational interpretation in CLP(BNR) would be something like this: the first line creates six interval variables with known initial domains and a couple of constraints. The second line creates many more variables connected to the original six through the *relation* distance2. Since initially the six parameters of the reciprocal lattice are only slightly constrained, the conceptually distinct computed values all have ranges from 0 to infinity. The third line *non-deterministically* matches observations to computed values.

For example, the first computed value will always match (<=) the first observation, thus constraining further the cell parameters on which the first computed value depends, which in turn affects all the computed values. Each successful match narrows the ranges of the cell parameters and thus makes further matches more difficult. As usual, a matching failure triggers backtracking to try another possibility.

To show this in a more concrete form, the program was modified to output the cell parameters after each successful match. The cell parameters after the first seven matches (for artificial input data) are:

```
1      [0.32848,0.33152]   [0.32848,9.2196e+18]   [0.32848,9.2196e+18]
     [0.00000,1.0001]    [0.00000,1.0001]         [0.00000,1.0001]

2     [0.32848,0.33152]  [0.49899,0.5010]        [0.49899,9.2196e+18]
      [0.00000,1.0001]    [0.00000,1.0001]       [0.00000,1.0001]

3     [0.32848,0.33152]  [0.49899,0.5010]        [0.99949,1.0006]
      [0.00000,1.0001]    [0.00000,1.0001]       [0.00000,1.0001]

4     [0.32848,0.33152]  [0.49899,0.5010]        [0.99949,1.0006]
      [0.00000,1.0001]    [0.00000,1.0001]        [0.00000,0.0091536]

5     [0.32848,0.33152]  [0.49899,0.5010]        [0.99949,1.0006]
       [0.80937,0.81946]  [0.00000,1.0001]        [0.00000,0.0091536]

6     [0.32848,0.33152]  [0.49899,0.5010]        [0.99949,1.0006]
        [0.80937,0.81946]  [0.63359,0.64923]      [0.00000,0.0091536]

7     [0.32848,0.33152]  [0.49899,0.5005]        [0.99949,1.0006]
        [0.81018,0.81896]  [0.63359,0.64923]      [0.00000,0.0091536]
```

Illustration of convergence of reciprocal lattice parameters
form: step number A B C
CosA CosB CosC

Notice that after the first match (obviously the (1,0,0) line) that all three sizes have changed because of the ordering convention imposed on our representation. At step 2 (the (0,1,0) line) , B becomes very narrowly defined and the lower bound of C is changed. C is defined by the third step, the (0,0,1) line. The next three steps fill in the cosine values. Note that on the seventh step (the first constraint which is "redundant"), there is only a small adjustment to one cell size and slight narrowing of one of the cosines. Subsequent match steps may also result in a refinement of the cell parameters; to explain why, it is necessary to discuss what would usually be called an error analysis.

The raw experimental data consists of the spacings of lines on the photographic film; such lines must have a finite thickness to be observable. Some lines may be substantially thicker than others, however, so one should actually measure distances to both the left and right edges of each line in order to obtain an interval representation for the line spacing. The line spacings will be related to the observation data by the above algorithm through a mathematical relation, the Bragg formula, combined with relations derived from the geometry of the apparatus. These relations involve other imperfectly known parameters such as apparatus dimensions and the wavelength of the incident beam. If sources of systematic error (e.g. shrinkage of the photographic film during development) are known or suspected, these effects may be explicitly incorporated in the model and assigned appropriate (and hopefully narrow) initial ranges. All these sources of uncertainty will make the observation sequence more uncertain than the raw data. Even when the raw data has uniform errors, the non-linear nature of this transformation will mean that the errors in the observation data sequence may differ substantially from datum to datum. The tedious task of inferring proper 'error bars' on the observation data from the uncertainties in the raw data (and other parameters) is done automatically by the interval arithmetic machinery.

It is a characteristic of the usual apparatus employed that errors in the raw data are comparable, determined by the precision of distance measurements. The errors in the observed squared distances (the observation list) then depend directly on the cotangent of the scattering angle, so the most precise observations are those in which the scattering is aproximately a right angle, and this generally occurs at higher order lines. On the other hand, the quadratic formula alone makes higher order lines more sensitively dependent on the cell parameters. The net result of these effects is that higher order lines, when matched against computed values in the above algorithm, can cause a slight reduction in the size of the cell parameter intervals, thereby refining the original estimate due to the first six matches.

From a practical point of view this 'error propagation' property of interval arithmetic can be considered an alternative to the use of more complicated statistical methods, such as least squares. Of course, conceptually speaking, statistical techniques such as least squares (based on L2 norms) make quite different assumptions from the interval method (which would correspond more closely to the L1-norm-based statistical approach), and the results have quite different interpretations: a least squares solution is a 'point' solution which *best approximates* the data in the presence of essentially normal errors, while the interval solution describes a region in which the solution *must* lie if it is to account for the data (with error bars explicitly specified).

EFFICIENCY ISSUES

As a benchmark, artificial observational data was generated as described earlier, and then solved using this method, using the low precision CLP(RI) system on an 25Mhz 68030.

The first test used the "point" cell parameters [1, 2, 3, 0.12, 0.23, 0.45] and produced the answer

```
[0.99999,  1.0001],[1.9999,  2.0001],  [2.9999,  3.0001],
[0.11999,0.12001],[0.22999,0.23001],[0.44999,0.45001]
```

after 19.9 seconds. The actual search, however, only consumed about 1.13 seconds of this.

The second test, which used the partly nearly orthogonal cell parameters [1.0, 1.3, 1.8, 0.32, 0.01, 0.2] produced the answer:

```
[0.99999,  1.0001],[1.2999,     1.3001],[1.7999,  1.8001],
[0.31999,0.32001],[0.0099,0.010003],[0.19999,0.20001]
```

in 7.25 seconds, only 0.43 of which was spent in the search.

Finally, we note that *a priori* information limiting the range of cell sizes or angles (provided by other physicochemical tests or by measuring a the angles of a macroscopic crystal), or additional symmetry information (e.g. all sides equal) can be incorporated directly and declaratively into the specification of the cell. In some cases of common occurrence (e.g. orthogonal crystals), a simplified version of the algorithm would provide performance advantages. The main point is that tuning the performance of the interval arithmetic involves many of the same tradeoffs (such as that between speed and completeness) and the same strategies (such as maximizing the use of *a priori* information to prune searches) that appear in ordinary Prolog programming.

CONCLUSION

This example illustrates several aspects of relational interval arithmetic. One is the use of a declarative programming style which emphasizes the conceptual structure of the problem and which makes it easier to manipulate the problem description to take advantage of special circumstances. The clarity and generality achieved has a price: one can no longer exercise such fine control over the operational details of the algorithm. The logic programming methodology suggests starting with the most general problem statement, and then transforming to less general but more efficient formulations by applying additional knowledge (or additional explicit assumptions) . In addition, the use of interval arithmetic permits new approaches to the treatment of traditional problems of precision and error analysis. Finally, essentially the same paradigm can be used for both discrete and continuous variables.

References

[1] L. V. Azaroff and M. J. Buerger, The Powder Method in X-ray Crystallography, McGraw-Hill, 1958.

Using Interval Arithmetic for Non-Linear Constrained Optimization

William J. Older

Bell- Northern Research

P.O. Box 3511, Station C

K1Y 4H7, Ottawa, Ontario

1.0 Introduction

One of the most important problems of applied mathematics which arises in most areas of the quantitative sciences and engineering disciplines is that of optimizing some function of many variables subject to various side constraints. The general case of real-valued non-linear functions of real variables, where the constraints may be non-linear and either of equality or inequality type, is especially difficult. This paper is an informal preliminary report on the use of relational interval arithmetic to solve such problems under the restriction that all functions are explicitly and finitely expressible and have continuous first derivatives.

As a physically meaningful example of this sort of problem, consider the problem of minimizing the cost of a heat exchanger, taken from section 7.2 of reference [4], and simplified slightly by removing some obviously redundant constraints. It is expressed in BNR Prolog syntax in precisely the form required by the algorithm described in this paper.

```
minimize(
        1300* exp( 0.6 * ln( 20000*6/(4*sqrt(T11*T12)+(T11+T12))))        (EQ 1)
      + 1300* exp( 0.6 * ln(12000*6/(4*sqrt(T21*T22) +(T21+T22))))        (EQ 2)
where [
                        T11==500 -To1,                                     (EQ 3)
                        T12== 250 - Ti1,                                   (EQ 4)
                        T21== 350 - To2,                                   (EQ 5)
                        T22==200 - Ti2,                                    (EQ 6)
                        Ti1>=150, Ti1=<240,                                (EQ 7)
                        To1>=250, To1=<490,                                (EQ 8)
                        Ti2>=150, Ti2=<190,                                (EQ 9)
                        To2>=210, To2=<340,                                (EQ 10)
```

0=<Fi1, 0=<Fi2, 0=<FB12, 0=<FB21, 0=<Fo1, 0=<Fo2, (EQ 11)

2.941=<FE1, FE1=<10, (EQ 12)

3.158=<FE2, FE2=<10, (EQ 13)

FE2*(To2 - Ti2)==600, (EQ 14)

FE1*(To1 - Ti1)==1000, (EQ 15)

150*Fi1 + To2*FB12 - Ti1*FE1==0, (EQ 16)

150*Fi2 + To1*FB21 - Ti2*FE2==0, (EQ 17)

Fi1 + Fi2 == 10, (EQ 18)

Fo2 + FB12==FE2, (EQ 19)

Fo1 + FB21==FE1, (EQ 20)

Fi1 + FB12==FE1, (EQ 21)

Fi2 + FB21==FE2 (EQ 22)

```
]).
```

In this particular case most of the constraints are in fact linear constraints, but the presence of some non-linear ones (Eqs. 14-17) plus the very non-linear objective function (Eqns 1 & 2) make the problem difficult. Symbolic elimination of variables is not helpful, since there are bounding constraints on most of the variables, which cannot therefore be eliminated cleanly. Although it is not obvious, it is stated in [4] that the problem is in fact convex, and therefore has a unique local minimum, thus making the minimization problem much easier. Maximizing this function, however, is still difficult, since there will generally be local maxima on each facet of the boundary of the feasible region. The algorithm which we will describe below is completely general, and can be used to solve either of these problems.

General problems of this sort have a variety of difficulties at most levels. Conceptually, however, there is no problem with existence of solutions under mild conditions such as compactness of the set of feasible points and continuity of the objective function. The traditional numerical approaches to optimization have usually been through some form of "hill-climbing" or gradient-ascent methods, with numerous variations to speed convergence. The first and fundamental problem is that in the absence of *a priori* knowledge, such as knowing that the problem is convex, a single such ascent is not sufficient since it will find only a local optimum. Many ascents leading to the same solution suggests that it may be the global optimum, but does not of course represent proof. There has not been, to my knowledge, any practical, systematic, and easy way to ensure (in a strong and precise sense) that a global optimum has been achieved. More modern techniques, like simulated annealing, seem to work quite well at finding good solutions on even very big problems, but still one never seems to achieve the desired degree of certainty that the solution has in fact been reached.

The second and more technical issue is simply to get one's favorite gradient (or whatever) hill climber to actually do what it is supposed to do. There is usually a certain amount (possibly implicit) of sampling and curve-fitting going on in such algorithms, and the situation may often be quite different than what the algorithm is assuming to be the case.

The third complication is non-linear equality constraints. Here there is a basic choice in techniques. One way is to replace the constraints with penalty functions and continue to use the basic hill-climbing technique for unconstrained problems. In this case the effect of the constraints is to make the "terrain" rougher (in the topographical sense), thus making the first and second problems much more severe. The alternative is to use projected gradient techniques, which requires programming of the partial derivatives, evaluating Jacobian matrices, and a matrix inversions (or equivalent) at every step. Naturally, each of these involve new problems as well as costs.

The fourth difficulty is due to inequality constraints, each of which may be either slack or tight at each feasible point. The slack ones can be ignored; the tight ones must be treated as equality constraints as described above. The problem is then that at each new point the algorithm reaches, the inequalities must be scanned to sort out which are tight, which are slack, and which are violated. Violations must be addressed first to bring the point back into the feasible region; this requires (at every step) reevaluation of the constraints, which may cause new violations, etc.

It should also be remembered that simple phrases like "invert the matrix" can hide a lot of nastiness. In particular, for problems in which there are redundant constraints (not uncommon) or local degeneracy the matrix will not be invertible; in other cases it may just be ill-conditioned and take you someplace you did not expect to ever get to. Similar things could be said about the infamous phrase "until it converges". It seems to me that it is a characteristic of these traditional techniques that there is a very large and rather scary gap between the high-level descriptions which one finds in journal articles and the concrete reality of the FORTRAN codes that implement them. A lot of things (such as, for example, all of classical numerical analysis) have been thrown into this gap in the last half-century, but the gap does not seem to have grown much smaller.

2.0 The CLP Approach

Relational interval arithmetic such as that of CLP(BNR) [1,2] offers an alternative to this traditional approach. Because it is tied to a strict logical framework and tracks in precise detail the consequences of floating-point approximations, it avoids many of the low-level pitfalls of the traditional approaches. Its meta-language (Prolog) makes it possible to encode abstract concepts and strategies directly into programming constructs, so that there is a fairly direct correspondence between the abstract theory and its implementation. There is, however, a price for these advantages, as one gives up not only the low-level control over execution (as one might expect), but also many of the traditional strategies, concepts, and presuppositions of traditional numerical computation. Our experience seems to suggest, in fact, that the technique of relational interval arithmetic works more effectively when coupled with the pure abstract mathematical formulations, rather than with their lower level translations generally encountered in traditional approaches.

The algorithm described in this paper will illustrate this thesis. Read declaratively, it consists mainly of a (very redundant) statement of the classical mathematical conditions for a local optimum. Read procedurally (to the extent that this can be done), it appears to be doing an exhaustive search over the feasible region, including every lower dimensional facet of its boundary set. It involves some matrix inversion (or at least something like what we usually call matrix inversion), except it is done just once before we start, and we do not at that point know what the matrix really is, and this part of the algorithm is in fact somewhat optional. Finally, it gives as output not just the position and value of the optimum (and its sensitivity coefficients), but *all* the optima and a rigorous proof (modulo certain hypotheses) of optimality.

The basic idea of this general algorithm is simply to convert the original continuous problem into a search over a discrete space of candidate solutions, and use branch-and-bound and enumeration techniques to solve the new problem. The candidate solutions will be just those states which *might be* local minima. The issue is then to formulate the necessary conditions for candidate solutions as a conjunction of explicit arithmetic constraints, which is precisely what the classical indirect method of optimization does.

2.1 The Kuhn Tucker Conditions

In this section we will review the ideas behind the classical indirect method for expressing such problems: the Kuhn-Tucker equations. For a full treatment, see reference [3], especially Chapter 9. We will deal with the case of minimization problems only, since minimizing -F is equivalent to maximizing F. Most of the difficulties in the classical theory have to do with finding *sufficient* conditions for a local minimum, but all we need is *necessary* conditions that are sufficiently difficult to satisfy that they serve to eliminate all but a few points.

We start with problems in one unrestricted variable x and with no constraints. Then since the objective function f is assumed to be continuously differentiable, a *necessary* condition for a minimum at x' is that the derivative f' is zero at x'. (Of course, it may also be a maximum or inflection point there, but we don't care.) If x is restricted by x>=0, then we find that there are two cases for minimum at x': either [x'=0 & f'(x')>0] or [x'>0 & f'(x')=0]. See figure 1. These two disjunctive cases can be packed quite conveniently into the single conjunctive expression: x'>=0 & f'(x')>=0 & x'f(x')=0. (This is a cheap mathematician's trick in some sense, but it is also the fundamental basis for this entire algorithm, since it avoids using "or" explicitly.)

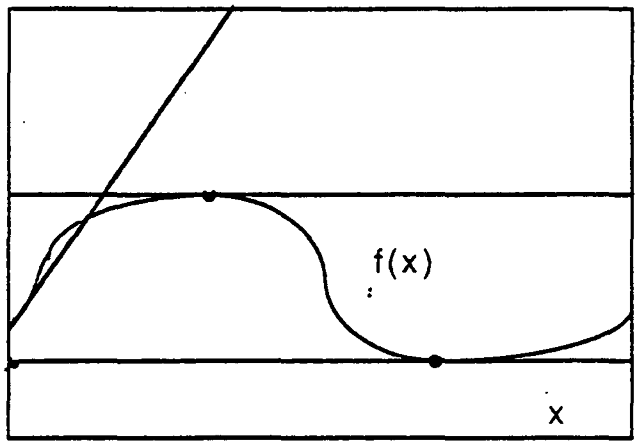


FIGURE 1. Necessary conditions for minimum: x>0 & f'=0 or x=0 & f'>0

For problem equality constraints of the form h(x)=0, the classical theory of Lagrangian multipliers tells us to look at the critical points of the Lagrangian function:

$$L(x, \lambda) := f(x) + \lambda h(x), \tag{EQ 23}$$

by solving the simultaneous equations:

$$\partial_x L = \partial_x f + \lambda \partial_x h = 0, \tag{EQ 24}$$

$$\partial_\lambda L = h = 0. \tag{EQ 25}$$

Note that the second equation is just the imposed constraint. The first equation involves the derivatives of the objective function and constraint function with respect to the state variable x, and the Lagrangian multiplier appears linearly. The value of the Lagrangian multiplier at any solution of equations 24 & 25 can be interpreted as a sensitivity coefficient, that is, it is the rate at which that particular optimum value (of f) changes as a function of the right hand side of the equation h(x)=0, i.e. if the 0 were changed to a small ε.

For inequality constraints of the form g(x)>= 0, we combine the previous techniques. First we define the slack variable s=g(x) and impose the condition s>=0. This results in the inequality constraint being replaced by an equality constraint s - g(x)=0 and a 0 lower bound on the new state variable s. Thus we form the Lagrangian:

$$L(x, \lambda) := f(x) + \lambda(s - g(x)), s >= 0, \tag{EQ 26}$$

and the critical points will be given by solutions to the simultaneous equations:

$$\partial_x L = \partial_x f - \lambda \partial_x g = 0, \tag{EQ 27}$$

$$\partial_s L = \lambda >= 0, \tag{EQ 28}$$

$$\partial_\lambda L = s - g = 0, \tag{EQ 29}$$

$$s >= 0, \tag{EQ 30}$$

$$\lambda s = 0, \tag{EQ 31}$$

where x and s are regarded as independent variables, so $\partial_x s = 0 = \partial_s x$. Equations 29 and 30 are equivalent to the original imposed inequality g>=0. Equation 31 is the *complementary slackness condition*, which says that either the constraint is tight (s=0) or else its sensitivity is 0 (=0). This set of relations is known as the Kuhn-Tucker conditions.

For the general case, we suppose a set of state variables $\{x_i, i=1,.,n\}$, an objective function $f(x)_i)$, a set of equality constraints $\{h^j(x_i)=0, j=1,m\}$ and a set of inequality constraints $\{g^k(x_i)>= 0, k=1,...l\}$. The Lagrangian function becomes:

$$L(x_i, \mu_j, \lambda_k) := f(x)_i) + \Sigma_j \mu_j h^j(x_i) + \Sigma_k \lambda_k(s_k - g^k(x_i)), s_k >= 0, \tag{EQ 32}$$

and the Kuhn-Tucker conditions are then:

$$\nabla_x L = \nabla_x f + \Sigma_j \mu_j \nabla_x h^j - \Sigma_k \lambda_k \nabla_x g^k = 0, \tag{EQ 33}$$

$$\nabla_\mu L = h^j = 0, \tag{EQ 34}$$

$$\nabla_s L = \lambda_k >= 0, \tag{EQ 35}$$

$$\nabla_\mu L = s_k - g^k = 0, \tag{EQ 36}$$

$$s_k >= 0, \tag{EQ 37}$$

$$\lambda_k s_k = 0. \tag{EQ 38}$$

Equation 33 can also conveniently be written in terms of the matrices $\nabla_x h^j$ and$\nabla_x g^k$ as

$$\nabla_x f + \Sigma_j \mu_j \nabla_x h^j = \Sigma_k \lambda_k \nabla_x g^k \tag{EQ 39}$$

which separates the equality- and inequality-parts nicely. (These are sometimes referred to as the Kuhn-Tucker *equations*.)Equations 34 and 36 just impose the original problem constraints, while equation 34 requires non-negativity of multipliers associated with inequality constraints, and equation 38 are the complementary slackness conditions.

These conditions are *necessary* conditions for a local minimum provided that a certain regularity assumption holds. (The significance of this restriction will be discussed later.) These then are the constraints which are imposed in the algorithm, which will be described in the next section.

3.0 Description of Algorithm

The algorithm consists of about nine pages of BNR Prolog source, which sits on top of the CLP(BNR) system. The CLP(BNR) system in turn consists of about fifteen pages of BNR

Prolog code which provides a run-time "compiler" for arithmetic expression as well as control and enumeration predicates, plus an "arithmetic inferencing engine" which is about 30K of C (or Assembler on 68XXX machines). Most of this engine code is for the 25 or so primitive relations of the CLP(BNR) language, and is in fact code generated by Prolog programs from "abstract machine" source code written in Prolog.

The algorithm can be conveniently decomposed into five parts: initialization and setting up the constraints for the feasible region, setting up the Kuhn-Tucker equations (including computing the symbolic gradients), "redundification", enumeration, and branch-and-bound. The form of the predicate, modeled somewhat on `setof`, is:

```
        minimize(F where Constraint_List, Y, Multipliers)          (EQ 40)
```

where initially `F` is an arithmetic expression, `Constraint_List` is a list of equality (==) or inequality (=< or >=) arithmetic constraints involving unbound (and unconstrained) variables, `Y` is a logic variable, and `Multipliers` is a logic variable. On (each) success, `Y` is the value of F at a global minimum, all the variables of `F` and `Constraint_List` have their corresponding values, and `Multipliers` is a list of values of Lagrangian multipliers corresponding to the constraint list. By "values" here is meant in general an interval, i.e. a Prolog variable constrained to lie in a closed real interval, however in some cases (especially "0.0") the variable may actually be instantiated to a number.

There is a current restriction that the entire problem must be given *explicitly* in the call. This means, for example, that there can be no user-defined functions in the arithmetic expressions. It would be relatively easy to relax this somewhat and allow a simple "macro" facility, e.g. a `tanh` function defined in terms of `exp`. But for arbitrary user-defined functions (e.g. a recursive function definition or fixed point definition) the problems of computing symbolic derivatives would be formidable. For much the same reason, we do not allow any of the incoming variables to be intervals, since this would allow the existence of "hidden" constraints.

Multiple solutions on output, although rare, can happen for several reasons. One obvious and common cause is symmetry in the problem. Another possibility, but unusual, is for two local optima to be so close in value that the finite precision arithmetic prevents us from definitely choosing one or the other. A third reason is redundant constraints at the optimum, which means that there is more than one distinct set of Lagrangian multipliers for the same state, and the predicate returns these as two different solutions.(See Figure 2 below.)

3.1 Initialization and Constraint Setup

The first step in the algorithm is to collect a list, with duplicates removed, of all variables used in the problem statement. Each variable is checked to ensure that it is not already constrained, as mentioned above, and is then declared to be real, i.e. an interval with unspecified bounds. The list of interval-variables, state variables, will be used to define the gradient operator.

The second step is to sort out the list of constraints. For each constraint we decide whether it is equality or inequality and transform it to the standard form: `F op 0`, where `op` is either == or >=. The current implementation performs this standardization as a separate pass over the constraint list, and returns the standardized constraints in two separate lists depending on type. (The rest of the processing is very similar for the two types, but it is convenient later to have two separate lists.) The transformations are given by:

```
normal_form_of_constraint(E1== 0, E1==0).

normal_form_of_constraint(E1== E2, (E1-E2)==0).

normal_form_of_constraint(E1>= 0, E1>=0).

normal_form_of_constraint(E1>= E2, (E1-E2)>=0).

normal_form_of_constraint(E1=< E2, G)
          :- normal_form_of_constraint(E2>=E1,G).
```

Then the gradient of each constraint with respect to all the state variables is computed symbolically, using some Prolog rules for partial differentiation. Since differentiation produces many expressions of the forms x + 0 or 1*x, some minor simplification is done during differentiation. The gradients expressions are "compiled" into CLP(BNR) constraints and returned as a list of intervals:

```
gradient([], F, []). % gradient(Variables, Expression, Result)

gradient([X,Xs..],F, [FX, Ds..]):-
           pd(F, X, DxF), % calculate partial derivative wrt X
           FX is DxF, % compile expression DxF to interval
           gradient(Xs, F, Ds).
```

After the expression `G` has been differentiated, we execute `S is G` to construct the interval network for `G`, and `S` op 0 to impose the constraint. Note that for inequalities the variable `S` represents a slack variable. For each constraint we create a new interval variable `L` to be its Lagrangian multiplier; it is unbounded for equality constraints but constrained to be non-negative for inequality constraints, which also impose the complementary slackness conditions `L*S`==0 at this time. The multipliers are returned in an output list. Another output list for inequalities also returns the formal product `L*S` which will be used later for enumeration.

```
form_ineq_constraint(G >= 0, Vlist, Grad, L, L*S):-

         gradient(Vlist, G, Grad),
         S is G,     % define slack variable
         S >= 0, % impose original constraint
         L >= 0, % impose Kuhn-Tucker non-negativity condition
         S*L == 0.    % impose complementary slackness
```

Finally, the objective function's gradient is added to the list of gradients for the equality constraints. This is done conveniently by adding it to the end of the equality constraints and treating this as the base case. The final output objective variable is created at this time.

At the end of part 1 we have imposed all the constraints, defined the objective function, defined the multipliers, and imposed the complementary slackness conditions. The multipliers have not been coupled to the state variables as yet, nor have the final Kuhn-Tucker equations (Eq. 33) been imposed yet. Hence failure at this point or before must be due to inconsistent constraints, or non-differentiable functions.

3.2 Kuhn-Tucker Equations

At the end of part 1 we have two vectors of Lagrangian multipliers, and two Jacobian matrices, and can form the Kuhn-Tucker equations (Eq. 33) for coupling the state variables to the multipliers by doing two matrix multiplications as in Eq. 39. The algorithm actually does the matrix multiplications for the two sides separately, each producing a different vector, and these vectors are then equated in a separate step.

3.3 Redundification

This step is very strange by conventional ways of thinking, and it is very strange on many levels. First of all, it is not logically necessary at all, but is solely for purposes of improving performance by making the problem statement harder! Early versions of this algorithm did not include it at all, and were able to do simple optimization problems quite well. However, as the size and difficulty of problems was increased, the enumeration costs became prohibitive until this step was added. For the sample problem given earlier, for example, it reduced the enumeration time by a factor of about 1500. (Of course, it also increased setup time considerably (a factor of 3 or so), but this cost is largely controllable and is expected to become substantially reduced in the near future.) It should also be mentioned that almost half of the source code is currently devoted to this step.

The basic idea of this step in general terms is very simple: the direct statement of a set of simultaneous equations will generally produce a constraint network with too many large fixed points. However, if any logical consequence (i.e. arithmetic relation) of the existing constraints is added to the system it will eliminate some (hopefully many) of these extra fixed points while leaving the "true" fixed points unchanged (becausethey are logical consequences).

For those who prefer to think in process algebra terms, these extraneous fixed points are essentially deadlocks: A and B are both in the "I won't narrow before he does" mode. But unlike the usual situation in communication systems, where adding a new communicating process just results in the deadlock absorbing the new participant, here adding extra constraints results (or may result in) breaking the deadlock.

The details of the redundification step, because of their general interest and utility, are discussed at length in section 4.

3.4 Enumeration

The task of enumeration is to force separation of the various point solutions. (Since the join of two solutions is necessarily a solution in a narrowing algebra, this forcing must be done by some external mechanism.) As we remarked in section 1, the complementary slackness conditions `L*S==0` (with `S>=0`, `L>=0`) is really shorthand for the statement "`S==0` or `L==0`". This expansion, with "or" being a Prolog choicepoint, is basically what we do during the first phase of enumeration. Geometrically speaking, setting the slack variable `S` to 0 means that the corresponding constraint is tight, while `S` > 0 (and hence `L==0`) corresponds to being "interior" with respect to a particular constraint and so this multiplier drops out of the Kuhn-Tucker equations. Each pattern of zero and nonzero values for the different slack variables represents a different m-dimensional facet of the n-dimensional polytope defined by the inequality relations. For redundant *problem constraints* it is possible to have both `S==0` and `L==0`, and this leads to multiple solutions with the same state but different sets of multiplier values, as seen in figure 2.

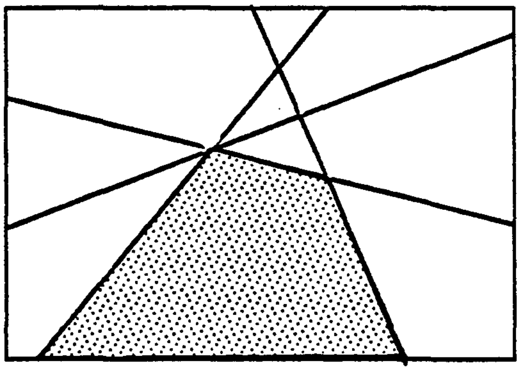


FIGURE 2. Redundant problem constraint

Note that since the complementary slackness conditions were imposed as CLP(BNR) constraints in part 1, if either `S` or `L` become nonzero because of the other constraints present, the other one will automatically become 0. As a result, for a problem with say 16 inequality constraints, the enumeration which formally has 2**16 ~ 65,000 branches, may in fact have very few. Some will be pruned away by failure (i.e. there could be no possible solution to the Kuhn-Tucker equations on that facet), others will be determinized by previous decisions forcing either `S` or `L` to 0. This is, of course, the same pruning effect we see in the semantic version of the Davis-Putnam Procedure with unit resolution and discrete constraint systems such as CHIP. The basic enumeration code is thus apparently very simple:

```
complementary_slackness([], _).

complementary_slackness([L,Ls..], X):-
```

```
        alternative(L, X),

        complementary_slackness(Ls, X).

%   L*S where L is multiplier, S is constraint

alternative(L*S, X):- S==0. % constraint is tight

alternative(L*S, X):- S > 0. % constraint is slack
```

It should be mentioned that we enumerate in such a way that we set S==0 first, i.e. we save the interior of the feasible region until last. This reflects a belief that practical industrial problems seem often to have solutions at a boundary point, so it makes sense to search the low dimensional facets first. Furthermore, these solutions (which seem to be easier in general to compute) provide fodder for the branch and bound algorithm, and thus may help to prune the search space in the potentially most difficult case, which is the interior of the feasible region.

Finally, after the enumeration of the complementary slackness conditions succeeds, if the state variables and multipliers are not already reduced to points (or "pointlike" intervals), it is necessary to use some general purpose forcing technique to pin down the possible solutions on that facet. In particular, the interior of the feasible region, if it has many critical points, will need to be decomposed so as to isolate them. So far, for relatively simple problems, the solve predicate of CLP(BNR) seems to be adequate for this. In more complex cases we expect to use a new control predicate presolve, which is much more "intelligent" in the sense of exploiting much more specific knowledge of the specific constraint network.

3.5 Branch and Bound

The last component of the algorithm is the branch-and-bound strategy, which works exactly the same here as it does in discrete constraint logic programming. That is, one has a generator of candidate solutions, and keeps the value of the best solution seen so far, and imposes the additional constraint that the next solution must be at least as good as the best seen so far.

There are two variations on this strategy in BNR Prolog. One makes use of Prolog side-effect primitives to remember the first solution of the current problem. Then the problem is restarted from the beginning of the enumeration with the added constraint that the new solution must be as good or better than the previous, in which case it is added or replaces (respectively) the previous solution. This is repeated until the search fails, in which case the remembered solution(s) are recalled non-deterministically.

The second variation is more subtle insofar as it is done entirely in the CLP(BNR) constraint network using a special (CLP(BNR) side-effect) primitive which enables one to export bounds from one branch of an or-computation into subsequent branches. In this case the enumeration is only done once, but each success tightens global constraint which makes subsequent successes harder. This naturally generates a monotone sequence of improving solutions, so something like setof is needed here to capture the sequence as a list, from which all but the last one(s) can be removed. (We suspect that this approach may usually be more efficient than the previous approach, although we have seen certain instances where it is not.)

4.0 Results for Example Problem

For the example problem given above, the entire solution process requires about 25,000 primitive interval operations, or about half a second on a HP-400 series machine to do the interval arithmetic. (Setting up all the equations takes roughly an order of magnitude longer than this in our current implementation.) Only two facets of the ~250000 possible facets are actually visited during enumeration; all other branches failing immediately. These two solutions have the same state and value, but different multipliers, and are due to a redundant constraint in the original problem statement.

The solution found (twice) has To1=310.0, To2=210.0, Ti1=210.0, Ti2= 150.0 and a cost of $56,825.83 in agreement with the "best known solution" given in reference [4].

5.0 Redundification and Variable Elimination

The general strategy of "redundification" has been described above. This can be illustrated nicely with a very simple example: the pair of equations C1=X + Y, C2 = X - Y. When C1 and C2 are known constants, the unique solution is of course the intersection P of the two lines representing the two equations as shown in figure 3.

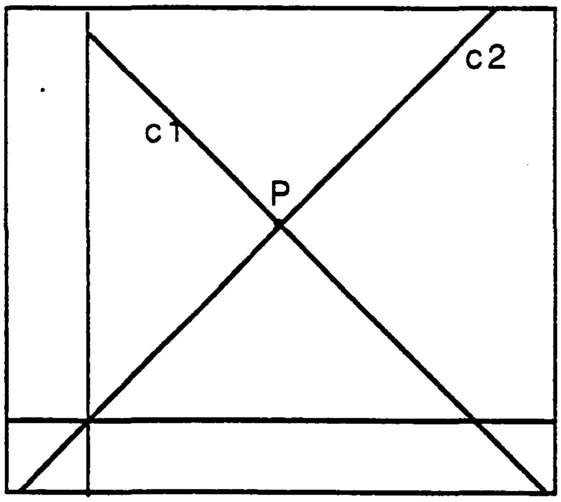


FIGURE 3. The Geometric Solution is P

However, the narrowing operator constructed from the two equations has as "fixed points" every square centered on P and with corners on the two lines, as shown in figure 4. (This case is very rare in two dimensions, for if either line is tilted slightly or bent slightly most of these fixed points will vanish, but cases like this becomes more common as the number of dimensions increases.)

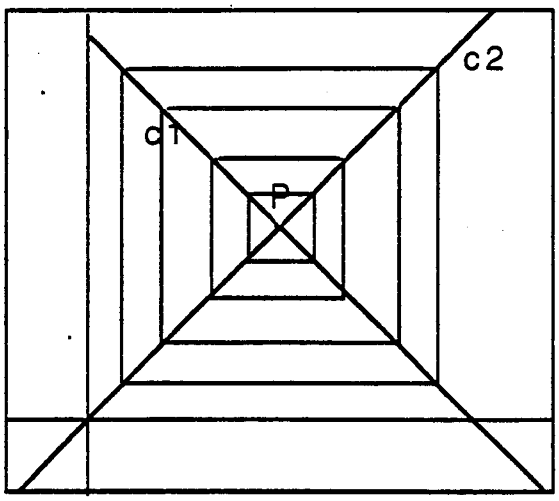


FIGURE 4. There may be extraneous fixed points: every square centered on P.

Adding a redundant equation, i.e. *any* other line through P, eliminates all the fixed points but P itself, as shown in figure 5.

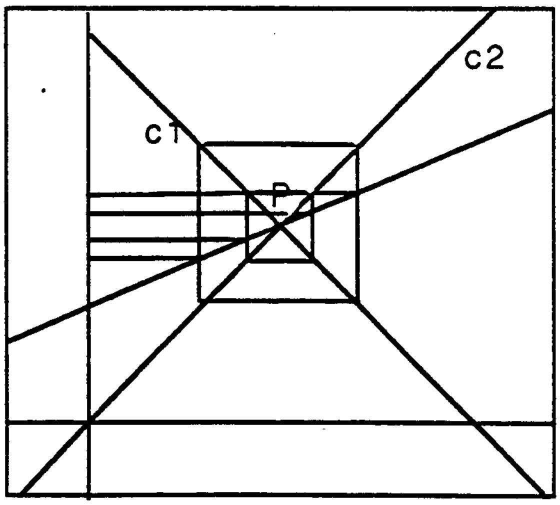


FIGURE 5. Adding redundant constraints destabilizes fixed points

In particular, adding a vertical or horizontal line through P, as in figure 6, will eliminate

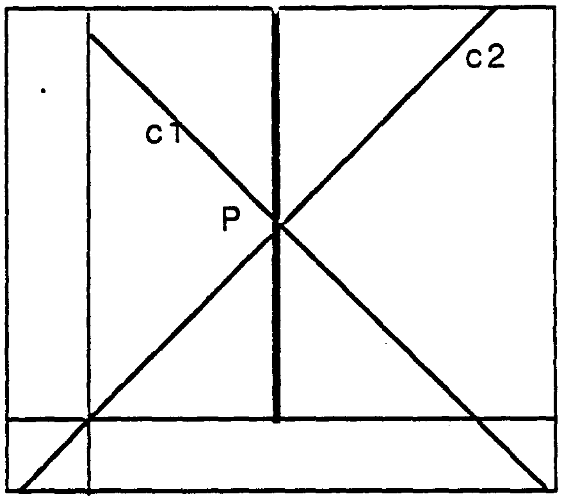


FIGURE 6. Pivoting as adding redundant constraints

all the extra fixed points most efficiently. But this is just the "pivoting" operation used in Gaussian elimination.

In general, consider two equations:

$$B1 = C1*X + D1, \tag{EQ 41}$$

$$B2 = C2*X + D2. \tag{EQ 42}$$

If these equations hold, so does

$$U*B1 + V*B2 = (U*C1 + V*C2)*X + (D1 + D2) \tag{EQ 43}$$

for any real numbers (or real valued functions) U and V. In particular, if we use U=C2 and V= - C1, we will get the consequent

$$C2*B1- C1*B2= 0* X + C2*D1- C1*D2, \tag{EQ 44}$$

i.e., with X eliminated. Passing over to interval arithmetic, and making sure that we use *only* interval arithmetic to perform all arithmetic steps, we can get a redundant equation which is guaranteed (since it no longer depends on X) to eliminate many extra fixed points (if there are any). (Note that doing this with ordinary floating point arithmetic does not work, since the resulting equation will most likely be formally inconsistent with the original pair because of rounding errors.)

Of course this is just pure algebra of the most elementary sort, and it is exploited all the time in traditional numerical programs, isn't it? One thinks, of course, of the usual algorithms for matrix inversion and the simplex method of linear programming. But most of the time (in the real world) the operations are performed in floating point arithmetic, for which the required axioms are not true, and so the *implication* has been lost along with correctness. Furthermore, these applications are mostly only done on numbers, not *functions*, so the full power of this simple algebra is hardly exploited at all.

Now in the Kuhn-Tucker equations (Eq. 39) we conveniently have matrix expressions of the form MX=Y where M is an nxm matrix of interval quantities, X is an m-vector of intervals (representing Lagrangian multipliers) and Y is an n-vector of intervals. The fact that there may be a number of constraints imposed on or between any of these quantities is irrelevant. Therefore, we can systematically reduce M to an upper triangular form by pivot operations of the above general type, with each pivot operation generating a number of redundant equations. (As an optimization, whenever the chosen pivot element is never 0, we can use the standard pivot operations which involve dividing by the pivot, exactly like in the usual Gaussian elimination algorithm.) Since M in general is not even square (let alone invertible), the last step in the triangular reduction has different cases depending on whether n >m, n=m, or n<m, as seen in figure 7. If one wishes, one can put some extra effort into exploiting sparsity, or trying to choose pivot elements that do not contain 0 or are very narrow intervals, or suchlike.

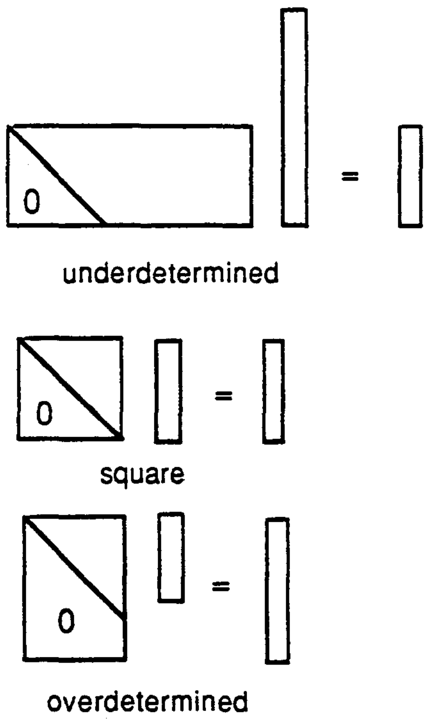


FIGURE 7. Pivoting cases

This algorithm is very much like a traditional one: apart from its being written in Prolog, it looks like a conventional algorithm, and the difference is entirely in the *meaning* and the *behavior.* In BNR Prolog and CLP(BNR) one can in fact just write it *as if* it were a traditional one for numerical matrices and test it, and then go back and make the tiny modifications required to handle intervals. Usually the intervals do not change much until near the very last step; if the matrix M and the right side vector Y are essentially known, the last step fixes one variable in X and this fixes another and so on. For the Kuhn-Tucker equations, where there are generally matrix products on both sides of the equation, it was found best (for ease of debugging) to treat each side of the equation separately by introducing two new vectors Y and Y', and only afterwards coupling the two systems together by equating Y and Y'. This can trigger a dramatic amount of narrowing, and in some problems will just produce the answer directly without enumeration at all.

Naturally the Kuhn-Tucker equations, which are linear in the Lagrangian multipliers (regardless of how non-linear they may be with respect to the state variables) will respond very well to this treatment. However, *it is not necessary that the equations be linear equations for this technique to be applicable.* All that is required is that they can be written as linear forms, where the coefficients can be any quantity at all. More generally, other operations besides * and + and =, if they satisfy similar axioms, will yield similar algorithms.

6.0 Proof Procedure

The method described above, unlike traditional methods, can be regarded as an automated proof procedure, modulo certain assumptions. It is worthwhile to make these assumptions explicit.

First, obviously we are assuming the correctness of the implementation. This includes especially the floating point implementation, the interval arithmetic "engine" implementation, the translation of the equations into interval networks, and the Prolog programs which implement the algorithm. Some degree of strict, even formal, verification of all this, while not easy, is at least conceivable. In particular, some strict verification of IEEE floating point implementations may be required for other purposes. (Unfortunately, some crucial issues, such as the precision of transcendental functions is not covered by the standard, nor is it usually specified by the vendors to the degree necessary for interval arithmetic.) We are currently looking at ways of doing formal verification of primitive interval operations in cooperation with Dr. Bruce Spencer of the University of New Brunswick.

Second, the Kuhn-Tucker conditions are not strictly necessary, but depend additionally on a regularity assumption which essentially says that the optimum does not occur on a cusp as shown in figure 8. A sufficient condition for this is that the Jacobian matrices appearing in the first Kuhn-Tucker equation have maximal rank. Alternative formulations expressed in terms of spaces of feasible directions avoid this technicality, but thereby lose the valuable practical advantage of generating the values of Lagrangian multipliers as a by-product. There are some subtle issues here which need to be examined very closely. One issue is that rank maximality is generic, so an infinitesimal change in the problem (e.g. by altering a coefficient slightly) should restore regularity. Therefore, it is at least plausible that a treatment by interval arithmetic in which all constants are fuzzed slightly would be unable to miss such a solution. On the other hand, it is also unclear whether such an unstable solution is really a valid solution to the engineering problem, even if it is a solution to the mathematical problem.

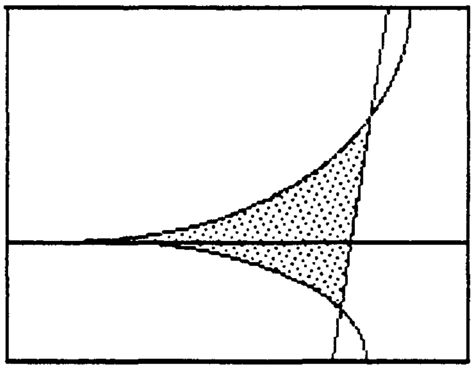


FIGURE 8. The leftmost feasible point is on a cusp

Third, the branch-and-bound strategy relies on the *existence* of solutions; interval arithmetic, however, only guarantees *possible* existence. As a practical matter, if a solution is known to 20-digit accuracy and all constraints are satisfied to such accuracy, one does not hesitate to accept it, and the question of the theoretical existence of real numbers exactly satisfying the Kuhn-Tucker equations is purely academic. It would be nice, mathematically speaking, if we could provide such a theoretical guarantee, but, philosophically speaking, it is unclear what such a guarantee could possibly mean, since one's actions are not affected by it.

7.0 Conclusions

We have presented a general and apparently practical algorithm for solving constrained optimization problems by using relational interval arithmetic. It is based on using Kuhn-Tucker conditions to convert the problem into one with a discrete solution set (viz. find all possible Kuhn-Tucker critical points), and then using branch-and-bound to find the minimum such critical point. The complementary slackness conditions are used in part to drive the enumeration of solutions. Another important part of this algorithm, in terms of efficiency, is the use of pivoting operations to provide systematic redundancy in the constraint network, and it is noted that this is a general technique of interval arithmetic and not limited to linear problems.

There are several preliminary conclusions which may be drawn from this exercise. One is that the formulation of mathematical problems in interval arithmetic is quite different from traditional implementation techniques. It is not just a matter of shorter programs with a different structure and a declarative reading; deeper issues such as "what is mathematically relevant?", "what is difficult?", and even "what is the meaning of a solution?" are all transformed in significant ways. There is, or can be, a very close relation between the implementation and the abstract mathematical theory it embodies.

We do not know what the practical limits of this technology are, or may be in the future. There has been such rapid progress in the interval arithmetic technology recently that problems, such as this one, which we would have considered beyond our capabilities a few months ago, now appear very approachable.

8.0 References

[1] W. Older and A. Vellino, "Constraint Arithmetic on Real Intervals" to appear in *Constraint Logic Programming:Selected Research*, F. Benhamou and A. Colmerauer (eds), MIT Press, 1993.

[2] W. Older and F. Benhamou, "Programming in CLP(BNR)", BNR Research Report, 1993.

[3] R. Fletcher, *Practical Methods of Optimization*, (2nd Edition), John Wiley & Sons, 1987.

[4] C.A. Floudas and P.M.Pardalos, *A Collection of Test Problems for Constrained Global Optimization Problems*, Springer-Verlag,?.

Application of Relational Interval Arithmetic to Ordinary Differential Equations

William J. Older
March 1995

wolder@bnr.ca

Abstract

Relational interval arithmetic can be used in the numerical solution of ordinary differential equations, although the overheads can be substantially higher than for conventional arithmetic. In return for the increased overheads, relational interval arithmetic can offer several advantages which are difficult to achieve using traditional approaches. The first and most fundamental of these is correctness, in the sense that the interval solution when properly implemented is guaranteed to contain the true trajectory; that is, both conventional truncation error and roundoff error can be replaced by an interval of uncertainty in the solution. The second advantage is symmetry : the same formulation solves not only initial value problems, but also the final value problem, as well as problems in which there is partial information at both ends.

1. Introduction

The conventional approaches to the numerical solution of ordinary differential equations have several problems. The most fundamental of these is that these techniques introduce errors at every step of the integration: truncation error which results from using a finite order approximation to a Taylor series, and roundoff error due to floating point arithmetic. Another set of problems is due the limitations of functional language: one can directly solve the initial value problem (or the final value problem) but other boundary conditions require iteration of the integration procedures, with a great increase in the complexity of the code and the cost of solution. Iteration is also used to investigate structural stability issues when the equations contain parameters whose value is not known precisely.

Thus although it is possible to implement something like the traditional integration algorithms in interval arithmetic, there is no real incentive to do so as the results would still suffer from at least some of these problems. One is therefore led to reformulate the integration algorithms in ways which can better take advantage of the formal properties of interval arithmetic. We will develop a family of such algorithms in the remainder of this paper, using the language of CLP(BNR).

2. Ordinary Differential Equations - General Framework

First we establish some notation. The differential equation will be written generically as:

$$dy/dx = f(x,y)$$

where y may be a vector (as indicated by the boldface). We assume that we are interested in this equation within a box (an approximate "flow box") defined by closed intervals for x and each element of y. We also assume some degree of differentiability of f (with respect to x and each y) within this box; the precise degree required will depend on which algorithm is being used. As usual, if f does not depend on y, the algorithms reduce to simple quadratures.

As a Prolog predicate, we then get something of the conceptual form:

```
integrate(ODE,Initial,Final,Flowbox,Control,Output).
```

We want all the variables of the problem to be represented by uninstantiated (and unconstrained) logic variables. Furthermore it is necessary that we be able to match these problem variables with their corresponding initial and final value terms and their flow box bounds. The simplest way to accomplish this is to express the ODE explicitly (i.e. with the left hand side present), using the syntax:

```
X |> [Y1,...Yn] := [ F1,...Fn]
```

where X and {Y1,...Yn} are free logic variables, {F1, . . . Fn} are arithmetic expressions in {X,Y1, . . . Yn}, and we have introduced the "|>" infix operator to denote differentiation (of the right argument(s) by the left argument(s)). It is convenient to represent points along the curve (e.g. initial and final values) in the form [X,Ys] where Ys is the list of y values. In terms of control strategy, we will limit ourselves to those based on binary subdivision of the range of X, so `Control` is an integer indicating the number of doublings of the interpolation points; in particular, `Control`=0 indicates no subdivision. (Note that this assumes that the initial and final values of x are known precisely in advance, so they can be represented as ordinary floats rather than intervals.) Other control regimes, such as a variety of adaptive techniques, would be possible but are outside the scope of this paper. Finally, we assume that `Output` takes the form of a list of points along the curve, which we will treat internally as a difference list of form "`Head/Tail`".

With these assumptions in place, the code takes on the following general shape:

```
integrate(ODE,Initial,Final,Flowbox, Control, Out):-
      setup_ode( ODE, Method),
      $integrate(Control,ODE2,Initial,Final,Flowbox, Out/[]).

$integrate(0,Method,Initial,Final,Flowbox,[P,Ps..]/Ps):-
      Method( Initial, Final, Flowbox, P).

$integrate(Control,Method,Initial,Final,Flowbox, L/E):- Control>0,
      Cn is Control - 1,
      $interpolate( Initial, Final, Flowbox, Middle),
      $integrate(Cn,Method,Initial,Middle,Flowbox, L/M),
      $integrate(Cn,Method,Middle, Final, Flowbox, M/E).
```

The function of the second clause of `$integrate` is to insert a new point midway between the initial value of X and the final value of X, and to construct a vector of new Y values at this point; the flow box is used to initialize the ranges of these Y values, and is assumed to be of the canonical CLP(BNR) form `[real(Y1L,Y1H),...real(YNL,YNH)]`.

```
$interpolate( [X0,Y0s],[X1,Y1s], Flowbox, [XM,[YMs..]]):-
      XM is 0.5*(X0 + X1),
      YMs : Flowbox.
```

Finally, `setup_ode` must create/select the appropriate code for the algorithm being used, and the base clause for `$interpolate` must execute it. Since a separate instance of the basic integration routine will be needed at each subinterval, the appropriate code will be a predicate call and we must arrange for it to be called with the proper arguments. Hence we have assumed that the variable `Method` is bound to a symbol naming this dynamic predicate, and its calling sequence is

```
Method( Initial, Final, Flowbox, Output).
```

This completes the general framework for our family of algorithms. It remains to fill in the various versions of `setup_ode` for the specific members of the family.

3. Interval Eulerian Integration

In order to keep things as simple as possible to begin with, we start with the Eulerian algorithm, which replaces the differential equation by the difference equation, which has the initial value form

(3.1) $y_1 - y_0 = f(x_0,y_0)\ dx$

and a symmetrical final value form

(3.2) $y_1 - y_0 = f(x_1,y_1)\ dx.$

Both are, of course, fundamentally incorrect and only provide an approximation to the solution. Both can be thought of as arising from approximating the correct integral form:

(3.3) $y_1 - y_0 = \int_X f(x,y)\ dx,$

where the X denotes the interval $[x_0,x_1]$. An interval form can be derived from (3.3) since

(3.4) $m\ dx \leq \int_X f(x,y)\ dx \leq M\ dx$

whenever we have $m \leq f(x,y) \leq M$ over interval $X_{01}=[x_0,x_1]$ and so we have the inclusions:

(3.5) $f(x,y)\ \varepsilon\ f(X_{01},Y)$ and

(3.6) $\int_{X01} f(x,y)\ dx\ \varepsilon\ f(X_{01},Y)dx$ and

with Y *any* valid range estimate for y over interval X_{01}. In effect, we are using the natural interval extension of f to compute Lipshitz bounds, from which we can generate an inclusion for Y1. This leads to the following interval equations:

(3.7a) Dx is X1 - X0, DX : real(0,Dx),
(3.7b) $\mathbf{F}$ <= $f(X_{01},\mathbf{Y_{01}})$ % inclusion
(3.8a) $\mathbf{Y}_{10} - \mathbf{Y}_0$ == $\mathbf{F}$ * Dx, % - vector subtraction
(3.8b) $\mathbf{Y}_1 - \mathbf{Y}_0$ == $\mathbf{F}$ * DX. % .* scalar multiplication

The first step is to create the "dx" variables: Dx spanning from X0 to X1, and DX spanning from X0 to any point within the [X0,X1] interval. The first inclusion equation (symbolized by "<=') evaluates the rhs of the ode in the box $(X_{01},\mathbf{Y_{01}})$, but blocks any attempt to backwards narrow from its output. The next equation creates the fixpoint computation to narrow Y_{01} , and hence the interval slope $f(X_{01},Y_{01})$. Then the last extrapolates the resulting slope to the end of the interval in order to bound Y_1.

Equations (3.7) and (3.8) lead to the following code for the integration step. (For expository simplicity we are assuming that the CLP language has been augmented to handle vector operations.)

```
euler( [X0,Y0], [X1,Y1], Flowbox, [X1,Y1]):-
            X01: real(X0,X1),
            Y01 : Flowbox,         % range of Y in interval X01
            Dx is X1 - X0,         % assumed dx>0
            DX: real(0,Dx),        % or: U:real(0,1), DX is U*Dx,
            f( X01,Y01, F),        % range f over interval X01
            {Y01 - Y0 == F*DX},    % "forward predictor"
            {Y1  - Y0 == F*Dx}.    % extrapolation
```

The integration step assumes that the rhs of the original equation has been converted into a predicate f(X,Y,F) and then uses (3.8) to set up an implicit equation for Y01 in line 6 (the unknown range of f in the interval DX), and an explicit equation in line 7 for Y1. Lines 5 and 6 establish a "feedback loop" which narrows Y01 to some fixed point. Note that the last two lines are vector relations, with "-" denoting vector difference and * scalar multiplication (DX,dx being the scalars). The output is chosen to be the pair X1 and the (interval vector) Y1 which is guaranteed to contain the correct value, but other choices like [DX,Y] which bound the entire curve are sometimes useful. (Line 4 as written assumes that x1 > x0, but an alternate form given in the comment will work in either direction.)

The setup code which is responsible for creating predicates for f/3 is then simply:

```
setup_ode(  X |> Y := Exp ,   euler):-
      assert( f(X,Y,F):- [F <= Exp ]).
```

Note that the (also vector) inclusion operator <= has been used in this predicate.

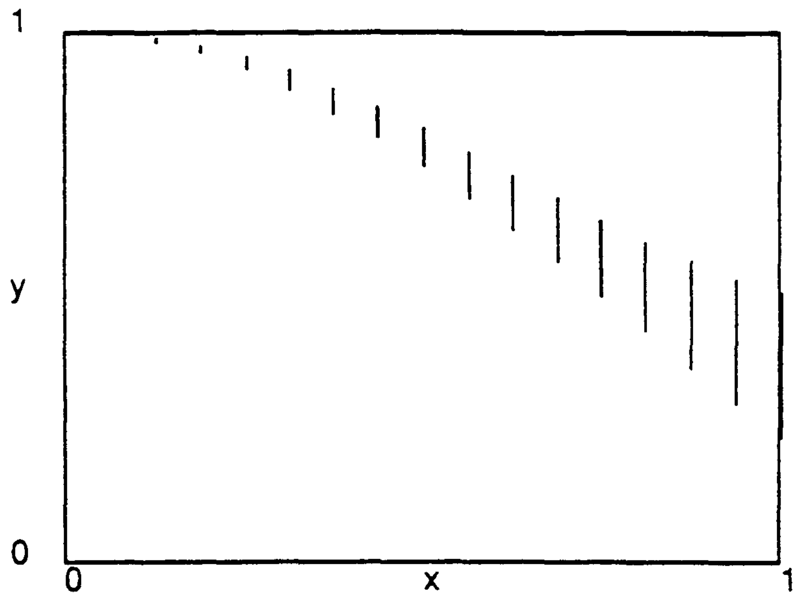
4. Examples

As an example we consider the nonlinear ode

4.1 $dy/dx = -2xy$

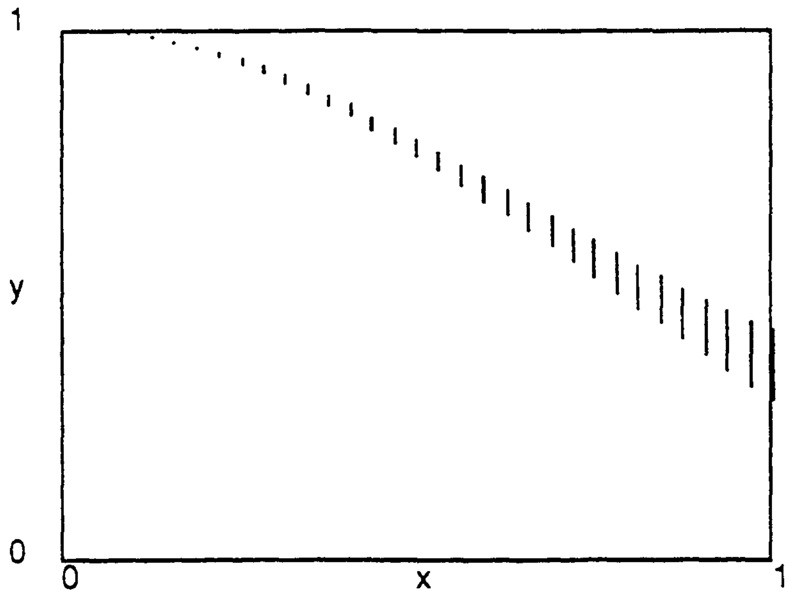
with known solution

4.2 $y= C \exp(-x^2)$,
over the interval [0,1.0]. By inspection y is decreasing and bounded below by the singular solution y=0, so the flow box is just [[0.0,y_0]], but we will use the estimate [[0.0, 1.0]]. Note that the expression on the right hand side of 4.1 is interval convex (since x and y are nonnegative) so the bounds estimates given by the natural interval extension would be tight if x and y were independent; however, as y(x) is decreasing in x, the extreme values achieved (at the NW and SE corners of the XY rectangle in the first quadrant) will be much closer than the interval bounds estimates based on the NE and SW corners. For this reason the uncertainty estimates will be quite pessimistic as we will see below.



dy/dx := ((-2 * x) * y)

Case 1a: Initial Value problem, Y0=1.0, Control=4.



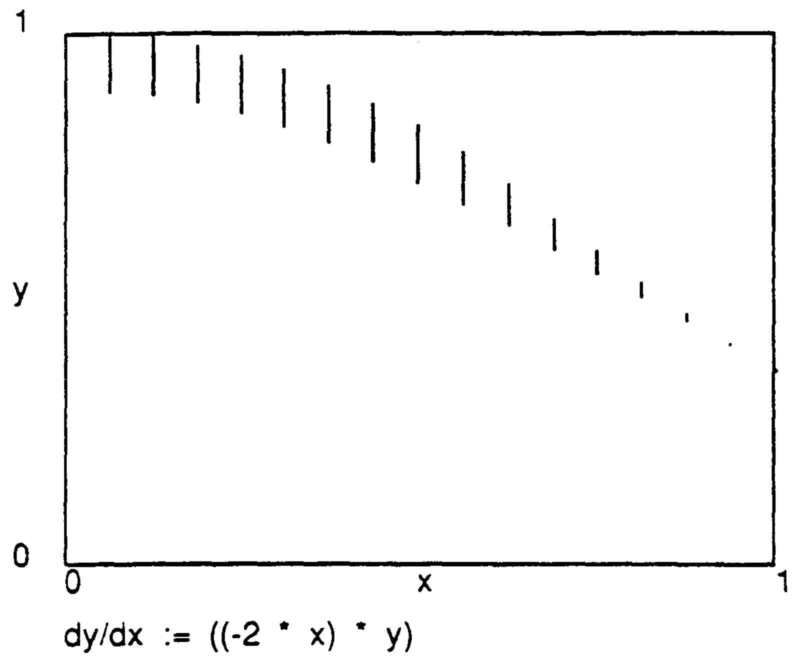
dy/dx := ((-2 * x) * y)

Case 1b: Initial Value problem, Y0=1.0, Control=5.

The width (delta) of the final y value depends on the control parameter and the number of subdivisions N used as follows:

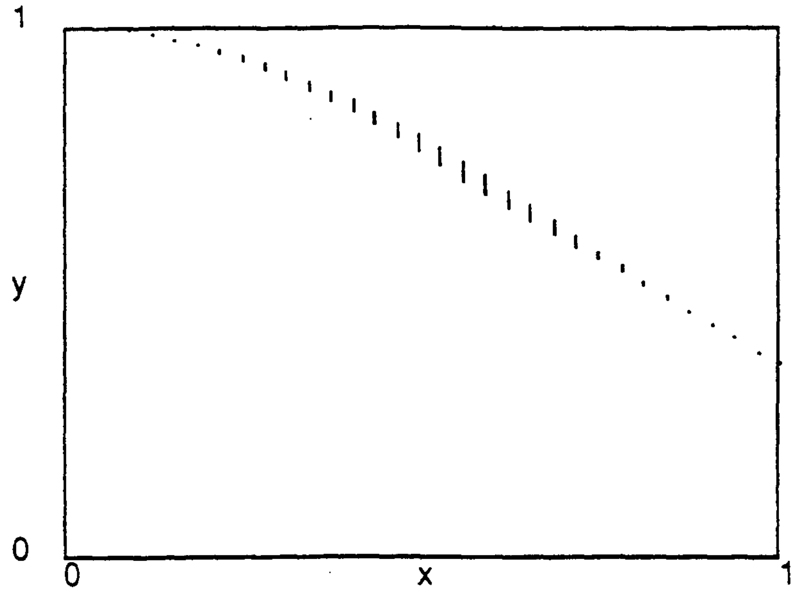
Control	N	delta(Y1)
2	4	0.85205
3	8	0.55412
4	16	0.28066
5	32	0.14079

One can see that each doubling of the number of subdivisions reduces the uncertainty by about a half, although this rate will eventaully diminish for large enough N as roundoff error begins to dominate.



Case 2. Final value problem, Y1 = 1/e=0.36788 .

The symmetry of the formulation and the relational nature of the interval arithmetic permits the known final value to propagate backwards through the equations, as expected.

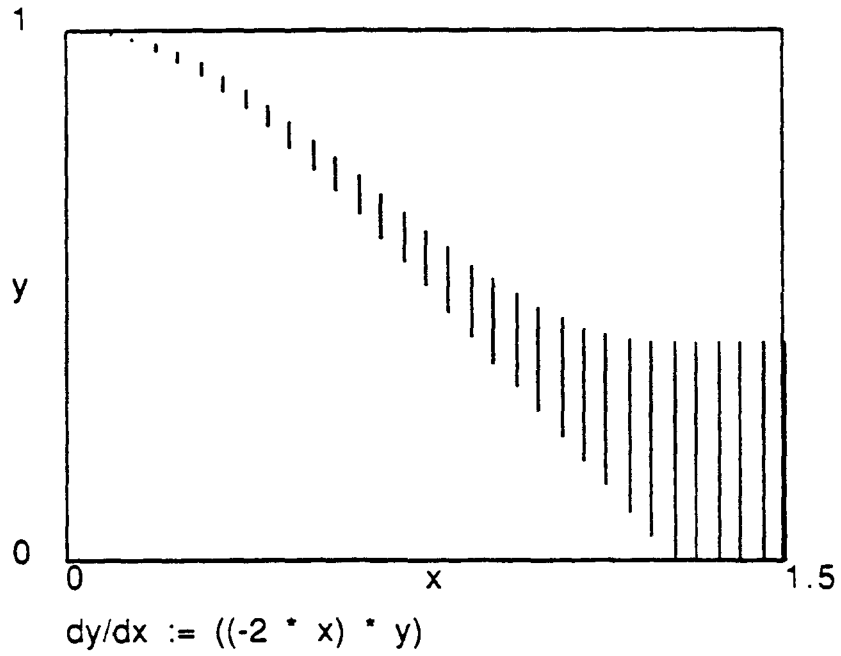


dy/dx := ((-2 * x) * y)

~~Case~~ 3. Both together, Y0=1.0, Y1=1/e = 0.36788 .

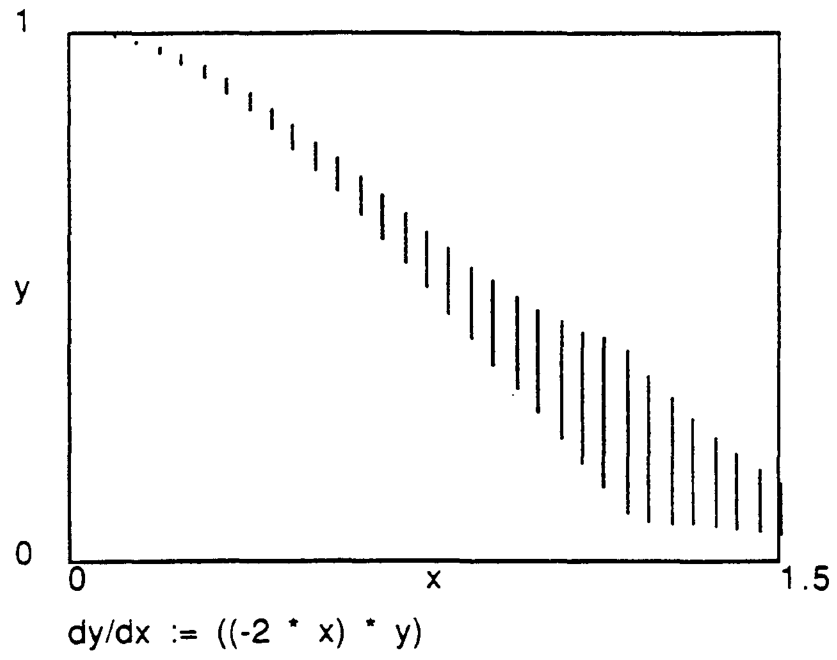
This would, of course, be considered an overdetermined problem classically speaking. Note that as one might expect the uncertainty is largest in the middle, away from the boundary conditions. Since the integration routine is correct, the imposition of a final value actually on the correct trajectory will always succeed, while those sufficiently far off the trajectory will cause failure for sufficiently high number of subdivisions. (Points sufficiently close to the true trajectory will not cause failures because with very fine subdivisions the effect of tracking rounding errors will keep the intervals from vanishing.)

It is apparent from these cases that one is getting a correct trajectory, and that the algorithm works symmetrically in both directions and even both directions at once. However, it is also clear that the interval of uncertainty is very large and only improves slowly with subdivision as one would expect with a first order method. The next few examples explore other aspects- or problems- of the technique.



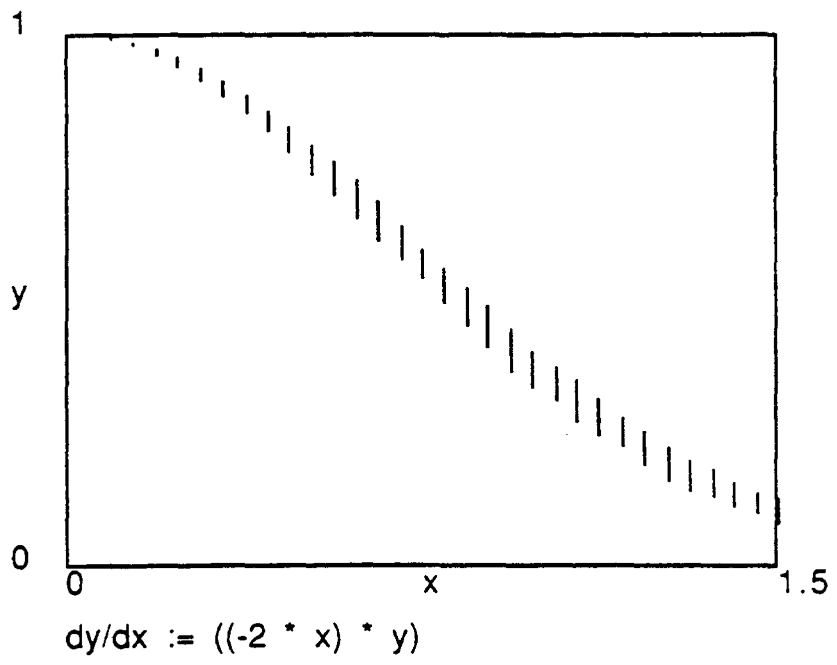
Case 4a: (a pathological case) initial value problem with $x_1=1.5$.

Here we have tried to extend the solution farther along the x-axis, but the solution becomes relatively useless shortly after x=1.0 . The problem here is due to the flowbox on y: the initial estimate of [0,1] refuses to narrow (at the low end), since the lower bound of the extrapolated y1 remains at 0; as a result, the possibility of a leveling off is not eliminated either. Reducing the step size postpones this effect only slightly; another strategy is needed.



Case 4b. Use of `absolve(YF,3)` on final Y value in case 5.

The overly large fixed point for the previous case contains points which when extrapolated backwards will be unable to match up with the forward extrapolation. These can be trimmed by using absolve; in this case a modest absolve(YF,3) done on the final value eliminates some of the problem.



Case 4c. Use of `absolve(Y1,2)` at *each* step.

In this case, we have modified `euler` to use `absolve(Y1,2)` after each step. This is very expensive in time, but has trimmed the error considerably (especially in the second half of the interval where we note the graph is convex upward) as well as overcoming the lower bound "stuck" at 0. Note that there appears to be a slight oscillation in the size of the error bars: this can be explained by noting that error bars expand until their non-viable fringes are large enough for `absolve(_,2)` to eliminate. The prohibitive cost of this approach makes it impractical generally, but it does suggest that it might be useful to employ `absolve` whenever there is insufficient narrowing of Y.

5. Trapezoidal integration- Geometric version.

The first order Eulerian integration algorithm has a problem with accuracy--although correct, the intervals of uncertainty are large, and- what is worse- they only improve slowly with subdivision. To improve matters we can use a second order inclusion in the integration step. Thus we return to equation (3.3)

(3.3) $y_1 - y_0 = \int_X f(x,y)\ dx,$

and try to refine our estimate of the definite integral. In this section we will use elementary geometric arguments (with y a scalar), and in the next section we will develop a more formal approach that can be generalized more easily. If we look at a graph of f (regarded as a function of x alone along the true trajectory) we see that the Eulerian interval approximation computes the area under it as the interval [mdx,Mdx] as shown in Fig 5.1a , with an "uncertainty" (M-m)dx where m and M are bounds generated by the natural interval extension of the original formulation of f.

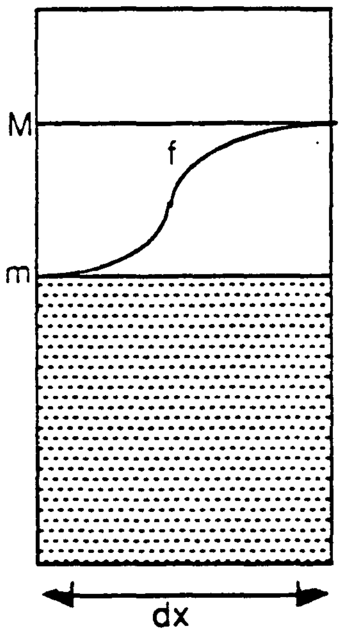


Figure 5.1a
Eulerian approximation

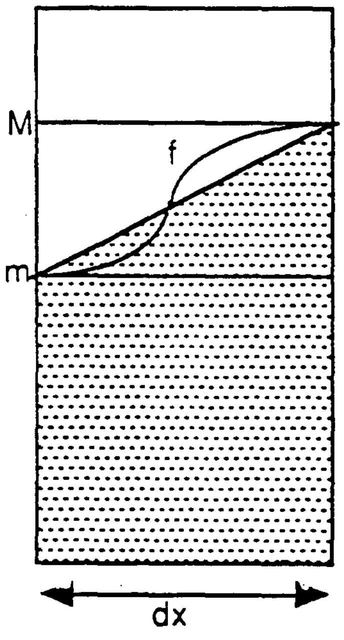


Figure 5.1b
Trapezoidal approximation

Geometrically speaking, trapezoidal integration would use the approximation given in Figure 5.1b, which uses the average of the values of f at the beginning ($f_0=f(x_0,y_0)$) and end ($f_1=f(x_1,y_1)$) of the interval. (For simplicity let us assume that we know both of these values exactly; this assumption will be relaxed later.) If the total derivative of f with respect to x is constant in the interval, this result is obviously exact. But in general- as shown- there is an error represented by the area between the graph of f and this straight line interpolation.

If we have bounds [b,B] for the total derivative of f, we get a picture as in Figure 5.2, where the graph of f is now constrained to lie within the parallelogram P formed by the linear equations:

$$m(x - x_0) \leq y(x) - x_0 \leq M(x - x_0)$$
$$m(x_1 - x) \leq y(x) - x_1 \leq M(x_1 - x)$$

for $x_0 \leq x \leq x_1$. Since the average slope line segement from y_0 to y_1 line bisects the parallelogram, the true value of the integral is bounded by $(1/2)*(f_0+f_1)*dx \pm 1/2$ area(P).

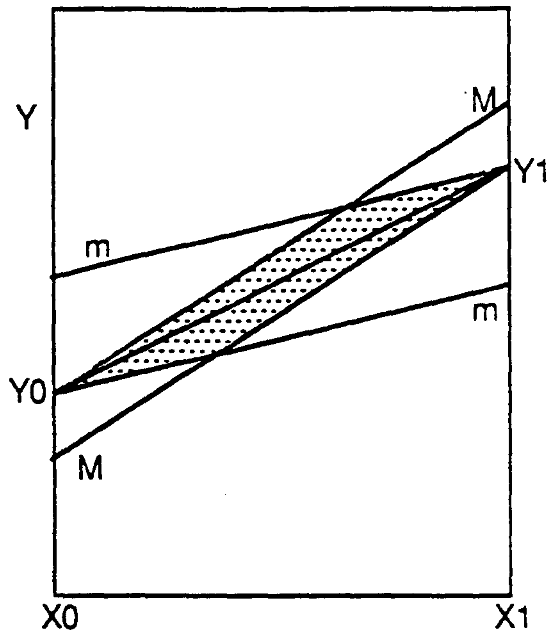


Figure 5.2
Graph of f lies in shaded region,
which represents the uncertainty in the integral

It remains to calculate the area of the paralledgram P. It is shown in Figure 5.3 that P can be rearranged by slicing a bit off the top and sticking it on the bottom, so as to make the tops and bottoms level. The area can then conveniently computed as the product of the base and height as shown. With $\mu = (y_1 - y_0)/dx$ being the average slope, and assuming that μ is greater than or equal to the average of M and m, the height h can be computed as

$$h = M\,dx - \mu\,dx = (M - \mu)\,dx.$$

(The reverse case leads eventually to the same formula.)
Similarly the base b can be computed from

$$y_0 + Mb = y_p = y_1(\,dx - b)$$

so $\qquad Mb = \mu\,dx - m(dx - b)$

so $\qquad (M - m)b = (\mu - m)\,dx$

and finally $\qquad b = dx\;(\mu - m)/(M - m).$

Hence the area of P is given by the pleasantly symmetric form:

$$\text{area} = dx^2 (\mu - m)(M - \mu)/(M - m).$$

This can be expressed more conveniently by introducing ρ $(0 \le \rho \le 1)$ defined by $\mu = \rho m + (1-\rho)M$, so

$$\text{area} = \rho(1-\rho)\,(M - m)\,dx^2 .$$

Note that the error vanishes whenever $\rho=0$ or $\rho=1$, since the parallelogram degenerates into a line segment, implying that the graph of f is a straight line. This formula gives a maximum error (half of the area) , which occurs at $\rho = 0.5$, of

$$\text{error} = (1/2)(1/4)\,(M - m)\,dx^2 .$$

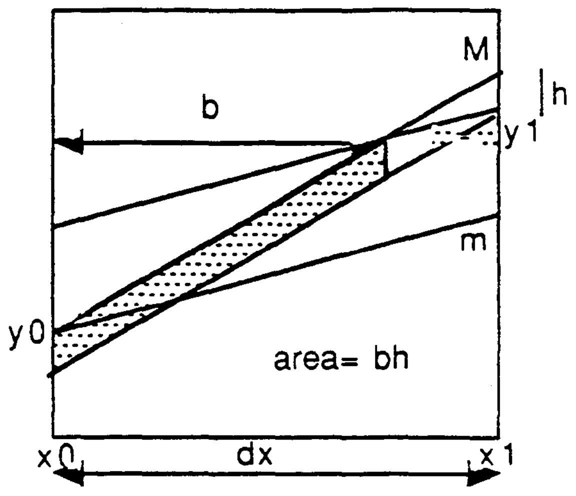


Figure 5.3
Bounding the error

6. Trapezoidal Integration by Taylor Series

In this section we will develop the second order error estimate by analytical means. We assume that f has a continuous total derivative with respect to x. in the interval $[x_0,x_1]$. If we expand y in a Taylor series with remainder about $x=x_0$ we get exactly

(5.1) $\qquad y(x) = y_0 + f_0 (x - x_0) + {}_0R_2$

about $x=x_1$ we get exactly

(5.2) $\qquad y(x) = y_1 + f_1 (x - x_1) + {}_1R_2$

where ${}_0R_2$ and ${}_1R_2$ are the respective remainder terms. Taking the difference of (5.1) evaluated at $x=x_1$ and (5.2) evaluated at $x=x_0$ gives the identity

$$2(y1 - y0) = f_0 (x_1 - x_0) - f_1 (x_0 - x_1) + ({}_0R_2 - {}_1R_2)$$
$$2(y1 - y0) = (f_0 + f_1)(x_0 - x_1) + ({}_0R_2 - {}_1R_2)$$

yielding

(5.3) $y1 - y0 = (1/2)(f_0 + f_1)(x_0 - x_1) + (1/2)({}_0R_2 - {}_1R_2)$.

We recognize the first term on the right as the truncated trapezoidal formula. So it remains to generate an interval estimate of the remainder "error" quantity $E=(1/4)({}_0R_2 - {}_1R_2)$.

To do this, it is helpful to review the derivation of (5.1) to see the origin of the remainder term. We start with:

(5.4) $y(x) = y_0 + \int y'(t)\, dt$

where the integral is from x_0 to x, and apply integration by parts to get

$$y(x) = y_0 + \left[\, y'(t)(t-x) - \int (t-x)\, y''(t)\, dt \,\right]$$

evaluated from t=x0 to t=x, giving

$$y(x) = y_0 - y'(x_0)(x_0 - x) - \int_{[x0,x]} (t-x)\, y''(t)\, dt$$

(5.5) $y(x) = y_0 + y'(x_0)(x - x_0) - \int_{[x0,x]} (t-x)\, y''(t)\, dt$.

Since $y'(x_0) = f(x_0, y_0) = f_0$, this agrees with (5.1) if we set

(5.6) ${}_0R_2 = \int_{[x0,x1]} (x_1 - t)\, f'(t)\, dt$.

Similarly, we get

(5.7) ${}_1R_2 = \int_{[x1,x0]} (x_0 - t)\, f'(t)\, dt$.

Then

$$E=(1/2)({}_0R_2 - {}_1R_2) = (1/2)\int_{[x0,x1]} (x_0 + x_1 - 2t)\, f'(t)\, dt$$
$$= \int_{[x0,x1]} ((x_0 + x_1)/2 - t)\, f'(t)\, dt.$$

The integrand is a product of f' and the ramp shown in figure 6.1.

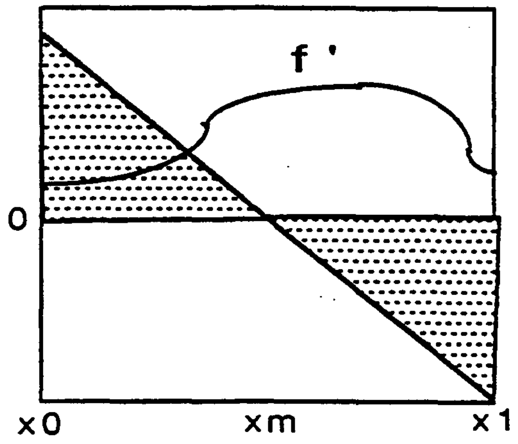


Figure 6.1
Error term is integral of the product of f' and a ramp

We estimate this integral by breaking it into its two parts at the midpoint $x_m=(x_0 + x_1)/2$ and estimating each separately:

$$E = \int_{[x0,xm]} (x_m - t)\, f'(t)\, dt \; - \int_{[xm,x1]} (t-x_m)\, f'(t)\, dt$$

so $\quad E \; \varepsilon \quad (1/8)[m,M]\,(x_1 - x_0)^2 \; - (1/8)\,[m,M]\,(x_1 - x_0)^2,$
since $x_m - x_0 = x_m - x_0 = (x_1 - x_0)/2$, where $[m,M]$ are bounds for f' over $[x_0,x_1]$. Thus the interval error term can be expressed as

$$E= (1/8)\ [m-M,M-m]\ (x_1 - x_0)^2$$

or as

$$E= (1/4)\ \delta(R)\ (x_1 - x_0)^2$$

where the symmetric interval $\delta(R)$ (:= (R - R)/2) represents the width of interval R=[m,M]. Note that this simplified formula agrees with that of the previous section, although the processes by which they were arrived at are quite different.

A more precise treatment of this one would be to use e.g. separate estimates of f' over the two halves of the interval. On the other hand, the geometric argument suggests looking at where the average slope falls between the bounds, as a way of improving the estimate.

7. Trapezoidal Implementation

In this section we develop an implementation for the trapezoidal integration routine, which will require some symbolic differentiation. Using the same ideas and notation as was used earlier in in `euler` we can implement the integration step based on the previous two sections as:

```
trapezoidal( [X0,Y0], [X1,Y1], Flowbox, [X1,Y1]):-
            X01: real(X0,X1),
            Y01 : Flowbox,       % range of Y in interval DX
            Dx is X1 - X0,
            DX : real(0,Dx),
            f( X0,Y0, F0),       % slope at x0
            f( X1,Y1, F1),       % slope at x1
            {Fm is (F0 + F1)/2}, % average slope
            df( X01,Y01, R),     % aprox f' over interval DX
            {Y01 - Y0 == Fm*DX + (1/4) d(R) DX**2 },
            {Y1 - Y0 -  Fm*Dx == (1/4) d(R) Dx**2 }.
```

Here we used the notation d(R) to represent the continuous symmetric "delta" of interval R. Note that if f' is constant over the interval, d(R)=0. Also, we are here using a (nonexistent) vector CLP language for readability.

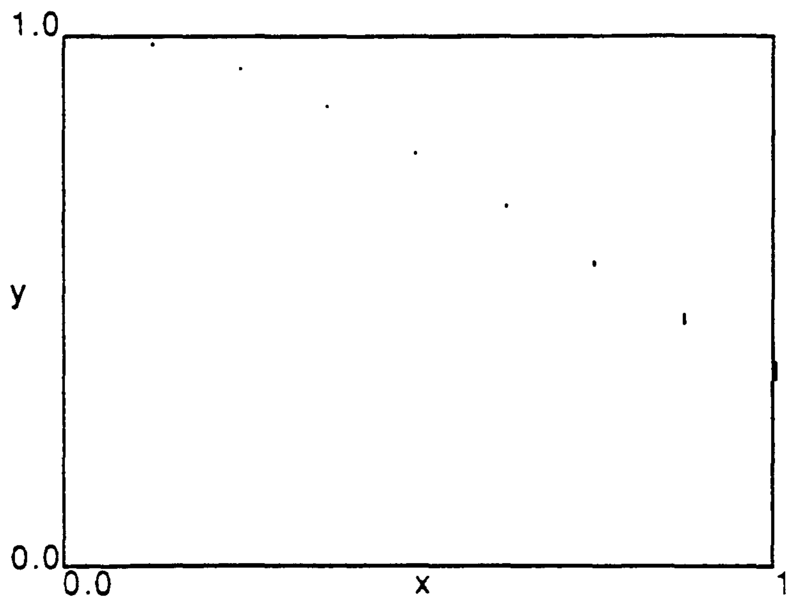
The only new feature that this adds is the creation of a predicate df to compute the derivative of f with respect to x along the trajectory, i.e. the convective derivative:

$$D_x (y'_j) = D_x (f(x,y)_j) = \partial_x(y_j) + \Sigma_i \, y'_i \partial_{y_i} f_j$$
$$= \partial_x(f_j) + \Sigma_i \, f_i \partial_{y_i} f_j \, .$$

This requires the use of a symbolic differentiation package; the one used here uses '|>' as a general infix partial differentiation operator in harmony with our previous use. Also we have used the inclusion "<=" in computing the output. Using this we can implement the necessary operations as:

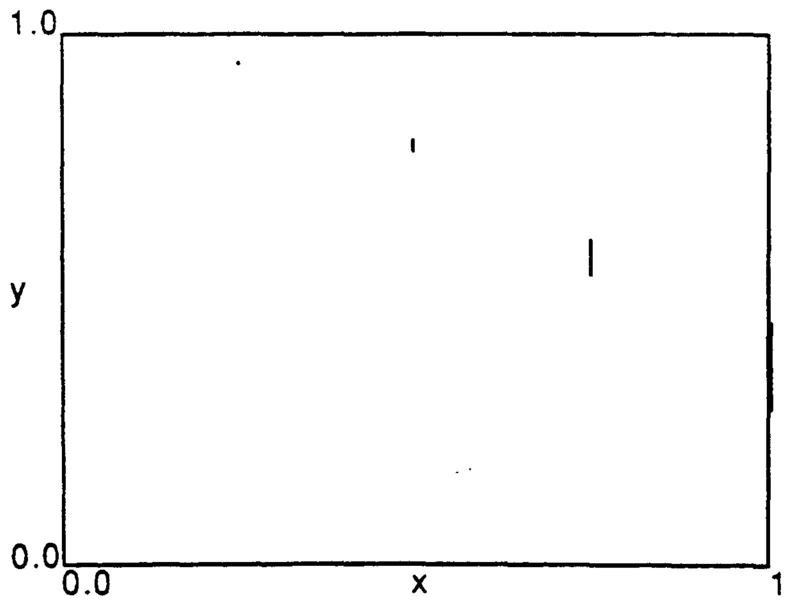
```
setup_ode( X |> Ys := Exps, trapezoidal):-
      retractall( $rhs(_..)),
      assert( $rhs(X,Ys,Fs) :- [Fs := Exps] ),
      retractall( $df(_,_,_)),
      second_derivatives( X |> Ys := Exps, Ds, Body),
      assert( $df(X,Ys,Ds) :- Body).

second_derivatives( X |> Ys := Es, Ds, Body):-
      convective_derivative( Es, X |> Ys := Es, Ds, Body).

convective_derivative( [], X |> Ys := Es, [], []).
convective_derivative( [E,Es..],X|>Ys := Fs,[D,Ds..],[D<= Ex,Bs..]):-
      {DD ::=  X |> E,          % compute direct derivative
       J  ::= Ys |> E,          % compute gradient
       ID ::= Fs +* J,          % dot with velocity
       Ex ::= DD + ID },        % add to direct derivative
      convective_derivative( Es,X |> Ys := Fs,Ds,Bs).
```

Using this implementation we can repeat the earlier examples.

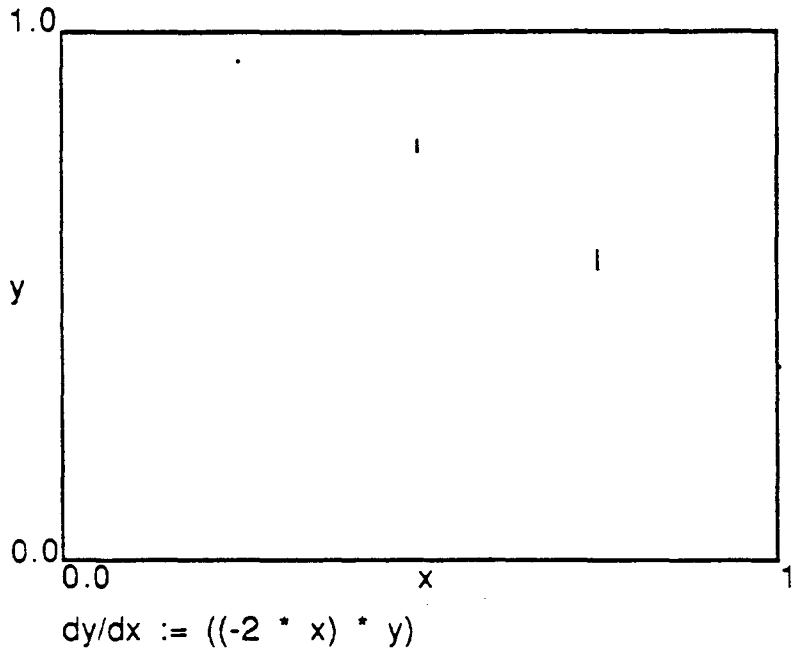

dy/dx := ((-2 * x) * y)

Case 1a: Initial Value problem, Y0=1.0, Control=3.

Even with half as many points, the uncertainties are under much better control. In order to see the error bars, we will drop to using only four points (Control=2), as shown below:


dy/dx := ((-2 * x) * y)

Case 2. Final value problem, Y1 = 1/e=0.36788 , Control=2.



Case 3. Both together, Y0=1.0, Y1=1/e = 0.36788 ., Control=2.

8. Concluding Remarks

In this paper we have taken the first small steps towards developing a relational interval arithmetic approach to the integration of ordinary differential equations. We have explored very slightly the issue of containing truncation errors, and demonstrated with simple examples some of the qualitative properties possessed by this approach: the formal correctness of the solution and its trade-off against growing uncertainty in initial value problems, the symmetry of use, and the use of redundant boundary conditions.

More sophisticated techniques, such as higher order and adaptive algorithms, and more sophisticated applications, such as two-point boundary conditions as well as the more complex boundary conditions that often appear in control theory problems, will be the subject of a subsequent paper.

A modified version of this paper appeared in the Proceedings of the CLP Workshop of the International Logic Programming Symposium held in Ithaca, N. Y. in November, 1994.

Bibliography

BNR Prolog User's Guide, 1988.

BNR Prolog Reference Manual, 1988.

Older, W.J., and Vellino, A., Constraint Arithmetic on Real Intervals, in *Constraint Logic Programming:Selected Research*, ed. Colmerauer and Benhamou, MIT Press, 1993.

Press,W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., *Numerical Recipes in C*, Second Edition, Cambridge University Press,1988.

Scheduling

Scheduling and related resource allocation problems are among the technically hardest problems which the average person is likely to encounter in ordinary circumstances. Such problems occur almost everywhere, and most of them are at least NP hard or NP complete, and the most important ones are very large. It is not surprising then that there is a large literature on such problems, nor that much of the practical focus has been on heuristic algorithms which aim to give merely "good" results rather than complete or optimal ones. This area has also been a fruitful one for the application of CLP techniques (mostly of the finite domain variety), although the practical focus here also has been on heuristics for the larger problems.

The emphasis in this section, however, will be on the exact solution of small (but not necessarily easy) problems. There are several points which provide justification for this emphasis. First, some important and hard problems are in fact now within reach of exact methods, given the right approach and adequate computing resources. Second, large problems usually consist of a large and intertwined set of small problems of the sort dealt with here; in this case it is plausible that a good formulation of the large problem can be constructed directly from good and complete formulations of the subproblems. Third, only exact/complete solutions can be rigorously compared with one another. Finally, many of the large practical problems appear to be very heavily constrained, sometimes so much so that it is possible that some exact methods may in fact be practically useable.

It appears that much of the scheduling work done using CLP techniques has been in the context of CLP technology of the finite domain sort such as CHIP and its derivatives, such as Charme and COSYTEC, and the CLP approach has been quite successful in this area. In this framework the scheduling medium variables (time or space) are discretized and eventually enumerated like everything else. This discretization of time and space is usually problematic, being both a conceptual distortion of the problem and a source of added decisions concerning the resolution needed. More important, it has also tended to obscure the very nature of scheduling problems in important ways. *In this discussion we will treat the scheduling medium, and* *sometimes resources as well, as continuous quantities, which cannot be enumerated.* This would seem to make the problems much more difficult, if not impossible, but in fact leads to significant simplifications.

Critical Path Scheduling

The Critical Path Method (CPM) is one of the most important traditional scheduling methodologies for the simple reason that it is one of the few subclasses of scheduling problems for which efficient (linear time) classical algorithms exist. It was also one of the earliest problems to be tackled by interval-based CLP techniques.

The problem is described as a set of activities, where each activity has a start time, a duration, and a finish time (= start + duration). The constraints are generated by a set of precedence rules which state that one task must finish before another starts. As with most scheduling problem there is a feasibility variant and an optimization variant. In the feasibility variant, all activities start after some initial time T0 and must finish before some deadline TF. In the optimization variant, the object is find the minimum span or distance between initial time and deadline, usually by minimizing TF with T0 constant. With TF at its minimal value, the start time (and finish time) intervals of some activities become point values; these activities are said to form the "critical path" and give the method its name. The residual intervals at non-critical activities (traditionally called "floats") represent some leeway or latitude in the starting/finishing/duration of the non-critical activities; small changes (within float) of timing of non-critical activities will not impact the total span.

Given the following problem specification:

activity	duration	follows
a	10	start
b	20	start
c	30	start
d	18	a,b
e	8	b,c
f	3	d
g	4	e,f
finish	-	g

a direct CLP(BNR) encoding of the problem is:

```
op(700,xfx,   before).

activity(  Name,   Duration,   task(Name,Start,Finish)):-
     { Finish == Start + Duration}.

task(_,_,F) before task(_,S,_) :- { F =< S }.

% specific data for a project:
project(  Start,   Finish,   [A,B,C,D,E,F,G]):-
      activity(a,10,A),  Start before A,
      activity(b,20,B),  Start before B,
      activity(c,30,C),  Start before C,
      activity(d,18,D),  A before D,B before D,
      activity(e, 8,E),  B before E,C before E,
      activity(f, 3,F),  D before F,
      activity(g, 4,G),  E before G,  F before G,
     G before Finish.
```

The feasibility variant, with a deadline of 60, is then posed as:

```
?-   project(task(start,0,0),task(finish,60,60),L).
     ?- project(task(start,  0,  0),
                 task(finish,  60,  60),
                  [task(a,  [0.0000,  25.001],  [10.000,  35.001]),
                   task(b,  [0.0000,  15.001],  [20.000,  35.001]),
                   task(c,  [0.0000,  18.001],  [30.000,  48.001]),
                   task(d,  [20.000,  35.001],  [38.000,  53.001]),
                   task(e,  [30.000,  48.001],  [38.000,  56.001]),
                   task(f,  [38.000,  53.001],  [41.000,  56.001]),
                   task(g,  [41.000,  56.001],  [45.000,  60.001])]).
YES
```

The optimization variant can also be done very easily as:

```
?-   project(task(start,0,0),task(finish,E,E),L),   lower_bound(E).
     ?- [project(task(start,  0,  0),
                    task(finish,  [45.000,  45.001],  [45.000,  45.001]),
                   [task(a,  [0.0000,  10.001],  [10.000,  20.001]),
                     task(b,  [0.0000,  3.0518e-5],  [20.000,  20.001]),
                     task(c,  [0.0000,  3.0001],  [30.000,  33.001]),
                     task(d,  [20.000,  20.001],  [38.000,  38.001]),
                     task(e,  [30.000,  33.001],  [38.000,  41.001]),
                     task(f,  [38.000,  38.001],  [41.000,  41.001]),
                     task(g,  [41.000,  41.001],  [45.000,  45.001])]),
          lower_bound([45.000,  45.001])].
YES
```

In general problems, optimization may be more difficult to do than feasibility and require, for example, the branch-and-bound technique discussed earlier, so the simplicity of this needs both explanation and justification. One way to justify it is simply to note that this is the way the classical algorithm does the problem, so we can recycle whatever the classical justification was: the problem decouples into a problem of calculating lower bounds of all intervals from T0, and a separate problem of calculating upper bounds back from TF, and if lower bounds are always less than upper bounds the problem is feasible, and finally because of the additivity of the delay constraints, a shift of TF translates into an equal shift of all upper bounds, hence setting TF back to equal the computed lower bound of the final activity generates the minimal solution.

The classical argument is quite problem specific and also does not involve CLP explicitly, so applying it here tacitly assumes that the CLP calculation of the lower bound of the final activity is in fact the same as the classical computation (modulo rounding differences). Showing this rigorously then requires formulating both the classical algorithm and the internal CLP mechanism in ways that can be compared usefully. This can be done, but is not really easy, because the CLP framework does not imply any particular sequence of operations.

It is useful therefore to find another way to tackle this problem, one that is both more general and more CLP oriented. This leads to the theoretical notion of interval convexity, which is explored in the next section.

Interval Convexity

The concept of interval convexity was introduced very informally by J. Cleary in the first paper on relational interval arithmetic, but was not formalized nor explored in any depth until Benhamou & Older (1992) and Older (WCLP, Marseille, 1993), which is the source for much of the material presented here. The context used for theory is that of relational interval arithmetic with intervals defined with real bounds, i.e. the infinite precision case.

Definition: A relation R is *interval convex in a state X* iff for every substate Y of X, $R \cap Y$ has all of its projections intervals.

That is, the interval hull closure operation can be omitted, and the interval representation is the same as the set representation. When the state X is the top state of the lattice, the state qualification is usually omitted. From this definition it is clear that interval convexity is persistent, that is, if true in a state it is true in every substate; it is also preserved under projections (since they are special cases of subtates).

There are a couple of useful theorems for recognizing interval convex relations:

Theorem: Convex relations and solvable relations are interval convex.

A relation is convex if for every two points in it, the line segment between them is also in it; it is said to be solvable if it can be solved (uniquely) for each of its variables as a continuous function of the remaining variables.

Many of the primitive relations are interval convex everywhere: ==,=<, +-, max, min, the odd powers, exp/ln. Most of the rest of the primitives are interval convex over slightly smaller states: abs and even powers on either the + or - half line, */ when the sign of at least one factor is known, trigonometric relations when confined to a region in which they are strictly increasing or decreasing.

A narrowing operator is said to have the *sequential instantiation property* (SIP) if for any choice of order of its variables, any *sequential* choice from the current domains of those variables will succeed. Here sequential choice means that one variable is instantiated, and only after the system has computed a new stable state, one picks the next variable from the new domain. The connection between interval convex relations and narrowing operators satisfying the sequential instantiation property is:

Theorem: **R** is the canonical narrowing operator for an interval convex relation R if and only if **R** has the sequential instantiation property.

This only holds completely in the infinite precision case; for floating point implementations the forward implication still holds, but of course nothing can be implied about R below the precision limit. Because of this theorem, there is a tendency to confuse the interval convexity of the relation with the SIP property of its narrowing operator, and speak loosely of interval convex narrowing operators, which is permissible so long as the other condition (canonical, i.e. minimal) holds.

Another sometimes useful theorem is:

Theorem: If two narrowing operators over different sets of variables each have SIP, and a single variable from one set is linked to a single variable from the other by a narrowing operator with SIP, then the whole network has SIP. Furthermore, if it fails it does so after at most one primitive operation, and if it succeeds it will do so in time linear (or less) in the size of the combined network.

Since most primitives are interval convex and their operators therefore have SIP, constraint networks which are trees are likely to have SIP. To be more precise, a network of primitive relations is said to be *locally interval convex in state X*, if each primitive in the network is interval convex in X. Then: a locally interval convex tree has SIP, by recursive application of the above result.

One way to show that an operator has SIP is to first show that its relation is interval convex (because e.g. it is convex), and then show that the operator is *tight*, i.e. that the bounds of its stable states are in fact solutions. Hence it must be the canonical narrowing operator, and then one can apply this theorem to derive SIP. In particular, this applies to the CPM problem: since it is composed of additions and inequalities only, its solution set is convex; it can be shown to be tight by instantiating to lower bounds sequentially from the earliest task start and instantiating to upper bounds in order of latest finish, so that there is no propagation possible from the instantiation.

Finally we note that if a feasibility problem can be reduced to a residual problem known to have SIP, *we can regard the residual problem itself as a generalized solution*, since it contains all the particular solutions (uncountably many of them in general), and any one of them can be generated easily (in quadratic time or less). For optimization problems which can be reduced to an SIP residual, one need *only* instantiate the appropriate bound of the objective variable to form the most general solution.

Serially Reusable Resources- Disjunctive Scheduling

The activities to be scheduled often require the exclusive use of some resource, and this requires serialization of the tasks using the same resource. One way of handling this is by using booleans to force serialization: for each pair of activities P and Q using the same resource we require that either P precedes Q or Q precedes P. This can be done with a constraint of the form:

```
$disjunct(task(P,S1,F1),  task(Q,S2,F2),  B):-  B:boolean,
         { B == (F1=< S2), ~B== (F2=<S1)}.
```

where S is the start time and F the finish of a task. The code to serialize all pairs from a list then follows the pattern of the `distinct` predicate:

```
serialize( List, Bs):- $serialize(List,Bs,[]).

$serialize( [X|Xs], Bs, EndB):-
        $serialize_1( Xs,X, Bs, E),
        $serialize( Xs, E, EndB).

$serialize_1([],_, B, B).
$serialize_1([X|Xs],Y,  [B|Bs],EndB):-
        $disjunct(X,Y,B),
         $serialize_1(Xs,Y,Bs,EndB).
```

Enumeration of the booleans then determines the service order of the tasks; when all the booleans have been enumerated, the residual problem is of the CPM type, and the interval convexity results can be applied to determine the minimal span for that particular task order.

Example: Job-Shop Scheduling

As an example we will consider a classical Job-Shop Scheduling problem: this consists of a set of tasks, each involving a sequence of operations, where each operations needs the use of a machine of a certain type, and there is only one machine of each type available. The problem is find an ordering of operations at each machine such that all the tasks complete before a given time, or to find the minimal time to complete all tasks. The particular example here has 5 tasks of 10 operations each and 5 machines, with a raw search space (all possible orderings) of about (10!)**5 or about 10**31.

The simplest direct specification of the specific problem (deadline/feasibility version) is given below:

```
jss(Deadline,F):-
     {S==0, F=<Deadline},          % S is start time
     A=[A0,A1,A2,A3,A4,A5,A6,A7,A8,A9],
     B=[B0,B1,B2,B3,B4,B5,B6,B7,B8,B9],
     C=[C0,C1,C2,C3,C4,C5,C6,C7,C8,C9],
     D=[D0,D1,D2,D3,D4,D5,D6,D7,D8,D9],
     E=[E0,E1,E2,E3,E4,E5,E6,E7,E8,E9],
     sequence(S,F,a,A,[29,78,  9,36,49,11,62,56,44,21]),
     sequence(S,F,b,B,[43,90,75,11,69,28,46,46,72,30]),
     sequence(S,F,c,C,[91,85,39,74,90,10,12,89,45,33]),
     sequence(S,F,d,D,[14,  7,23,61,35,18,52,29,11,69]),
     sequence(S,F,e,E,[37,84,  6,43,22,61,27,17,  9,73]),
     serialize([A0,A5,B0,B5,C1,C7,D4,D7,E1,E9],  Bs0,  m0),
     serialize([A1,A6,B3,B8,C0,C6,D0,D9,E2,E8],  Bs1,  m1),
     serialize([A2,A7,B1,B6,C4,C8,D1,D6,E0,E7],  Bs2,  m2),
     serialize([A3,A8,B4,B9,C2,C9,D3,D5,E4,E6],  Bs3,  m3),
     serialize([A4,A9,B2,B7,C3,C5,D2,D8,E3,E5],  Bs4,  m4),
     enumerate(Bs0),
     enumerate(Bs1),
     enumerate(Bs2),
      enumerate(Bs3),
      enumerate(Bs4),
      lower_bound(F).
```

The first block of code just sets up the start and deadline times and defines the lists of variables which will be the operations. The calls to sequence create the operation data structure, impose the sequencing constraints for the task, and ensure that the first operation occurs after the start time and the last finishes before the the value F. The code is:

```
sequence(Start,Finish,Label,List,Durations):-
      $sequence(List,Durations,Label,Start,Finish,0).

$sequence([],[],_,PreF,Fin,_):-  {  PreF  =<  Fin}.
$sequence([task(Id,St,Fin)|Ts],[D|Ds],Label,PreF,F,N):-  N1  is  N  +1,
      swrite(Id,Label,N),
      { PreF =< St,Fin== St + D },
      $sequence(Ts,Ds,Label,Fin,F,N1).
```

Then come the calls to serialize to set up the disjunctive constraints, followed by enumeration of the 275 boolean variables, and finally the use of lower_bound to calculate the actual time required by the interval convex residual problem. Considering the raw size of the search space and the apparent simplicity of this approach, the performance is surprising: the constraint setup (prior to enumeration) requires (on a 25Mhz 68030; Sun4 times are roughly half of these) 6.2 seconds in the current version of CLP(BNR). (This would be expected to drop by an order of magnitude in the foreseeable future.) Enumeration time for the first solution in the worst successful case (with deadline set to the minimal span of 634) is 1.383 seconds (with only 23 backtracks!). (A careful choice of enumeration order (busiest machines first) can reduce this to about 0.6 seconds.) More surprising even is that the worst failing case (with deadline=633, which requires searching the entire space) only required 2.783 seconds of enumeration time.

In order to compute the minimal span one can use the branch-and-bound technique. Because there can be a very large number of different schedules with the same span, the continuation method is not very useful, and it is better to restart the search after each solution (constraint setup needs to be done only once of course) with a deadline *strictly* lower than the best seen so far. Also, since there are many solutions with only slightly better spans, it helps to retarget agressively (e.g. 90% of previous span), and then use bifurcating search as soon as a failing value is discovered. With this approach the minimal span was discovered (and proved) in 9.9 seconds of search time (statistics after setup are accumulative):

```
setup costs: [90530, 6712, 1357, 1224, 6200]
target : 720
[1588, 150, 5395, 53, 616] : [719.0, 720.0]
target : 647
[3763, 396, 20468, 172, 2366] : [642.0, 647.0]
target : 577
[4054, 426, 21992, 173, 2633] : fail(577)
target : 609
[4767, 457, 25574, 174, 3133] : fail(609)
target : 625
[5921, 573, 40854, 229, 4850] : fail(625)
target : 633
[8947, 961, 69997, 448, 7983] : fail(633)
target : 637
[11538, 1208, 86645, 566, 9916] : [634.0, 637.0]
succeeds at:634.0
failed at:633
```

Serially Reusable Resource - Non-deterministic Sort Method

An alternative approach to the resource scheduling problem is based on sorting. The idea here is that if we knew the start times of the tasks and the finish times of the tasks *in order of time*, we could ensure feasibility by requiring that the start times and finish times interleave: first start followed by first finish followed by second start etc. If this is done the nth finish is preceded by n starts for all n, and by induction one can then show that the nth start and nth finish must belong to the same task. (This formulation may seem slightly odd, but we shall see later that it has its uses.) For sorting intervals we can use a declarative version of Quicksort, in which the binary comparison imposes an inequality constraint down each branch; with intervals, of course, both branches may succeed. Once we have an output from each of the sorts, we can impose the interleaving constraints; arrangements that survive are then again in CPM form and we can apply interval convexity to minimize the span.

This algorithm just described is very inefficient because one must do two non-deterministic sorts (worst case (N!)**2) before applying the interleaving constraints which prune the space, and it is obviously better to do both the sorts and the interleaving constraints together. When this is done the basic sort decision becomes that of ordering two tasks *just as in the disjunctive constraint version.* The code looks like:

```
serialsort(X,Y):- ssort(X,Y,[]).   % uses difference list

ssort([],A,A).
ssort([X|Xs],A,C):-
     part(Xs,X,Low,High),
     ssort(Low,A,[X|B]),
     ssort(High,B,C).

part([],_, [],[]). % partition on U, last 2 args are dif list
part([X|Xs], U, [X|Low], High):- before(X, U), part(Xs,U,Low,High).
part([X|Xs], U, Low, [X|High]):- before(U, X), part(Xs,U,Low,High).

before( task(_,S1,F1), task(_,S2,F2)):- { F1=<S2 }.
```

This was tested on a single resource benchmark (with 12 operations) and found to be more than an order of magnitude *slower* than the disjunctive approach given earlier. This would imply that on a problem such as the 50-operation job-shop problem given above it is likely to be at least 10**5 times slower and probably *much* worse. Yet, in an abstract sense, it is the "same" algorithm (expressed partly in Prolog rather than in CLP }. This underscores the importance of being able to postpone choices until they are forced, and reminds us that measurement of actual code of algorithms is absolutely necessary.

Serially Reusable Resource - Deterministic Sort Method

The basic sort method was not very efficient because of the non-determinism and the fact that it was necessary to postpone the interleaving inequalities until the sorts were done. We have seen one way to partly get around this. Another, very different, way is to do the sort *deterministically* using constraints. The key here is to realize that a 2-element sort function can be constructed using `min` and `max` in ordinary arithmetic. A standard recursive algorithm based on bitonic sequences can be used to define a general sort (of a power of 2 elements) from the 2-element sort. By changing the low level `min/max` to a constraint primitive, we can "lift" this algorithm into the CLP realm and it will still be a sort, an *interval sort.* The output list of intervals gives the bounds for the first, second, third, etc event in time order for every possible instantiation of the original list.. But, since this sort is now deterministic, we can sort the start and finish times and express the interleaving inequalities between the sorted lists, as in:

```
iserialize( Tlist ,SS, SF):-
      $split(Tlist, Starts, Fins, Durations),
      isort(Starts,SS),
      isort(Fins,   SF),
      $interleave(SS,[0|SF]).
```

```
$split([],[],[],[]).
$split(  [task(_,D,S,F)|Ts],   [S|Ss],[F|Fs],[D|Ds]):-   $split(Ts,Ss,Fs).

$interleave([],_).    % each start must be preceded by a finish
$interleave([S|Ss],[F|Fs]):-  {F=<S},  $interleave(Ss,Fs).
```

However, the interval sort is *not* interval convex, so there is a price yet to pay, as we cannot just instantiate the lower bound of the last finish to find the miminal span, and we need an instantiation method, and non-determinism will be reintroduced during instantiation. (A really good instantiation method for optimal schedules is still being sought.)

The possible advantage of using this interval sort approach is that given intervals for the time ordered start and finish times, it is possible to impose constraints on these ordered times which can help prune the space during instantiation. As an example, the simple interleaving inequalities can be augmented by recognizing that the sum of the previous idle time of the resource plus the amount of work done so far must be equal to the finish time of the current operation (a"conservation of time" law):

```
$conservation([],[F],IdleSum,   [Work],Finish):-
      { F == IdleSum + W,  F=< Finish}.
$conservation([S|Ss],[F|Fs],   IdleSum,   [W|Ws],Finish):-     Idle:real(0,_),
      {Idle == S - F,                 % this is equivalent to {F=<S},
      { Sum==IdleSum + Idle },
      { F == IdleSum + W},
        $conservation(Ss,Fs,   Sum,Ws,Finish).
```

The work estimate vector can be approximated from the list of durations (here assumed to be ordinary numbers) by sorting it in both ascending order WL and descending order WH,so the work done at the jth step is bounded between WL(j) and WH(j), and then defining a the estimated work vector by partial summation:

```
estimate_work_done(Durations,   [0|Ws]):-
      sort( Durations,  WL),
      reverse(  WL,WH),
      $est_work(  WL,WH,0,   Ws).

$est_work([],[],     Sum,   []).
$est_work([L|Ls],[H|Hs],   Sum,   [W|Ws]):-   W:real(L,H),
      {Sum1==Sum  +  W},
      $est_work(Ls,Hs,Sum1,Ws).
```

Note that the last value is the total work (the sum of the durations) and is a point value, so that the base clause of `$conservation` is very strong: the last finish time will be pushed out past the total amount of work to be done initially, and pushed out further everytime a idle definite idle period is created during instantiation. This is important: since the last finish time in every serial sequence must be less than the span, pushing out the final finish time makes the lower bound of the span a more accurate estimate and can lead to much earlier failure with tight deadlines. (Compare with Traveling Salesman.) The conservation equations above serve therefore much the same function as the global Carlier-Pinson inequalities in the traditional OR approach to Job-Shop Scheduling; it is apparently weaker in the middle operations (in terms of forcing local orderings of activities) than the local Carlier-Pinson inequalities (which use the specific durations of operations in each cluster instead of best/worst case bounds), but is stronger for final activities because it includes the effects of any forced idle periods, which seem to be common in real job-shop problems.

Narrowing of the outputs of an interval sort do not in general percolate back to the inputs (except at the two ends) since narrowings will not propagate backwards very well through min and max primitives of *overlapping* intervals. A partial exception occurs at the first and last intervals in a cluster in the ordered list: if instantiated, the propagation will percolate back to the original operations whenever there is a unique solution (i.e., when only one of the original operations could have had such an early/late value). This suggests the possibility that there may be an effective time-ordered instantiation strategy. However, once a choice of ordering is made during enumeration of the original operations, their intervals may become disjoint, and then the narrowings stuck in the sort network can trigger failure earlier than would otherwise be the case. Overall, however, this approach has not been very effective(compared with disjunctive scheduling) on benchmarks with busy machines. It may be more useful (especially at detecting infeasibile problems) in cases where the operations are of about the same duration and where there are many external constraints on time which produce many idle periods.

Serially Reusable Resources - Multiple Identical Resources

The method of disjunctive scheduling does not apply when there are multiple equivalent servers of the same type, since the several servers can perform overlapping operations. However, the method of deterministic interval sorting, interleaving equations, and conservation laws, does generalize fairly easily. (Older and Van Emden 1994-95) All that is required is that the shift (the prepended 0 on lists of finishes and work estimates) be changed to reflect the number of servers. For example:

```
     $interleave(SS,[0|SF])
```

becomes:

```
     shift(Nservers,SF,ShF),
     $interleave(SS,ShF),
```

where

```
shift(0,F,F):-!.
shift(Nservers,F,[0|Fs]):-  N  is  Nservers  -1,  shift(N,F,Fs).
```

Note that failure during the setup of these equations implies the infeasibility of the scheduling problem (just from the correctness property of CLP), so it becomes feasible to adjust the number of servers *during setup* to values for which a solution may be possible. The choice of a good enumeration strategy to be used with the interval sort-based algorithm is still open. Based on the single-server case, the method described in the next section may be a good candidate.

General Resource Scheduling - Cumulative Constraint

The previous method handles the common case of each task requiring one instance of a resource, or more generally, where each requires the same amount of the resource. But for the general case where different tasks make different resource demands, one needs a more general formulation. A useful interface to this general case is the so-called "cumulative constraint" introduced by COSYTEC.

The general form of the cumulative constraint is

```
cumulative(  Ss,Ds,Ws,Limit)
```

where `Ss,Ds,Ws` are lists of length N of numeric values representing the start times, durations, and resource requirements respectively (of a certain type) for N tasks, and `Limit` is the total amount of resource available. The interpretation is that at all points in the schedule the total resource usage must be less than `Limit`. (We use the language of time scheduling here for convenience, but any one dimensional space will do.) Usually the starts are intervals and durations and weights and limit are all constants, but all may be intervals if desired, although this does sometimes complicate the interpretation of the results. When durations and weights are intervals there may also be side constraints, e.g. that the product of duration and weight be constant. The starts, durations, weights etc. are here taken usually as real values, but can be used with integer variables also if some slight modifications are made.

The form of the cumulative constraint, i.e. with separate lists for the arguments, makes it convenient to apply it in more complex situations. Where there are several sorts of resources used by the same tasks, each will need a separate cumulative call in which the Ss and Ds are the same, but Ws and Limit are different (corresponding to the different resource usages) on each call. In a two dimensional placement problem formulated with cumulative, there will be a call for placement on the X-axis and one for the Y-axis with different starts (X-starts and Y-starts) and with duration and weights (i.e. heights) interchanged.

In the COSYTEC implementation, cumulative is a complex "black-box" primitive, the implementation of which is not described in detail. For CLP(BNR), it can be implemented in terms of the more basic facilities as a utility, and this can have advantages in terms of increased flexibility to adapt to problem demands.

Simple Formulation in CLP(BNR)

The semantics of this constraint is that at each point along the time axis (from the earliest to latest time mentioned), the sum of the resources in use at that time must be less than the limit. The geometrical picture is the familiar one of the Gantt chart, with weight shown as height. This can be expressed as:

(1) for each T: $\text{Limit} >= \text{sum}_k(W_k*(T \varepsilon [S_k,S_k+D_k])$.

That is, at each point we sum the Ws of just those tasks currently holding resources. Of course, this is impractical (for continuous time) since it represents an uncountable infinity of summations. However, we note that the resources being used only change at start and finish times of the tasks, so it suffices to check at only these values.

Hence we reformulate (1) in two steps: first define the finish times

(2a) $F_k = S_k + D_k$

then require:

(2b) for each T in $\{S_k\}$ or $\{F_k\}$:

$$\text{Limit} >= \text{sum}_k(W_k*(T \varepsilon [S_k,F_k]).$$

Altogether there are N constraints in 2a and 2N in 2b.

We can compute the boolean expression $T \varepsilon [S_k,F_k]$ in CLP(BNR) as:

```
{ Bk == (T >= Sk) and (T =< Fk )}
```

or more explicitly as:

```
{Bk1 == (T >= Sk),
 Bk2== (T =< Fk ),
 Bk is  Bk1 and Bk2},
```

where `Bk,Bk1,Bk2` are booleans. Note that with the substitutions envisioned for T, the low level booleans become

```
Bssik:=(Si > Sk),Bsfik:=(Si < Fk )
```

and `Bfsik:=(Fi > Sk),Bffik:=(Fi < Fk )`

so there is some overlap between them (`Bfsik=Bsfki, Bssik=Bsski,Bffik=Bffki`), which should be exploited by the common subexpression folding feature in CLP(BNR). There are also various dependencies such as `Bssik-> Bfsik` and `Bffik->Bsfik`, which also help reduce the effective complexity.

A couple of special points are important. First, the "diagonal elements" such as $S_k \varepsilon [S_k,F_k]$ and $F_k \varepsilon [S_k,F_k]$ should be replaced by1, which helps to get the algorithm started. Second, usually one permits a new task to start at the same time instant that an old one finishes, but this is prohibited by the formulation above. To fix this, we must alter the formulation slightly so that we exclude cases where a start and finish are equal. This can be done using (for T a start time S)

```
{Bk1 == (S >= Sk),
 Bk2== ~( Fk =< S),
 Bk is  Bk1 and ~Bk2},
```

and for T a finish time as:

```
{Bk1 == ~(Sk >= T),
 Bk2== (T =< Fk ),
 Bk is  ~Bk1 and Bk2}.
```

Now consider the effect of the sum constraint in eq. (2b). Depending on the extent to which start and finish times are known, we may know that some start or finish time of necessity occurs during some task, thus setting the booleans to 1, or alternatively, that they follow or precede serially thus setting the booleans to 0.

In the other direction, suppose we focus on one term, say:

```
        Limit >=  Wk*(Bssik and Bsfik) + Rest
```

for a case where `Rest` is so large (because of existing placements) that adding W_k to it would exceed `Limit`. Then the boolean value (`Bssik and Bsfik`) would neccessarily narrow away from 1 and hence become 0. If either `Bssik` or `Bsfik` is 1, the other must become 0, i.e. the task k will be forced either to strictly follow task i (`Bsfik` =1) (if it already starts later) or to start before thus narrowing a start time and ensuring that one of the booleans associated with start k is 1. In this way a tight resource constraint can deterministically force serialization of tasks.

The remaining case forces (`Bssik and Bsfik`) to 0, but both constituents are indeterminate. If either becomes 1, then the other is forced to 0 deterministically. But otherwise, we need to introduce a choice point of the form:

```
            (Bssik=0);(Bsfik=0).
```

These are the only choice points required; physically they correspond to the classical disjunctive scheduling form of serial resource use in which one must choose a serial order. Since N tasks have N! possible serial orders, one expects that there can be at most about $\log_2(N!) \approx N\log_2(N)$ such choices to make.

When all the choices have been made so that the resource summations are determined, then the remaining active constraints (since the booleans have become "dead") are all either the addition constraints from (2a) or a coherent set of inequalities as in a CPM problem, and hence possesses the sequential instantiation property. Narrowing of a time normally propagates only a short distance in such a network, but will be at worst of order N when successful. However, the creation of a directed cyclic graph, which corresponds to a cyclic precedence relation and hence eventually fails, may at worst require several iterations to do so, but in fact is usually quite fast.

CLP(BNR) Implementation

The implementation is now quite straightforward. This version is for a single resource problem with continuous variables; the booleans are exported for separate enumeration.

```
cumulative( Ss,  Ds,  Ws,  Limit,  Bs):-
      $add( Ss,  Ds,  Fs),
      $resource_constraints(Ss,Fs,1,  Ws,  Ss,  Fs,Limit,  Bs).

$add( [], [], [])
$add( [S|Ss],[D|Ds], [F|Fs]):- {F==S + D},$add( Ss, Ds, Fs).
```

It is more efficient to do the sums for both starts and finishes in one pass since the overhead can be shared; this also keeps the booleans dealing with the same two items together. We have also changed the encoding of the booleans so that the standard enumeration (which tries 0 first) will attempt to force tasks to be concurrent first.

```
$resource_constraints([],      [],        K, Ws, S1, F1,   Limit,[]).
$resource_constraints([S|Ss],[F|Fs],  K,  Ws,  S1,  F1,    Limit,  B):-
      K1 is K + 1,
      $sum_at( Ws,1,  S1,F1,  S,F,  K,  Sum1,  Sum2,  B,EB),  % print(Sum),
      {Sum1  =<  Limit,  Sum2=<Limit},
      $resource_constraints( Ss,  Fs,   K1,  Ws,  S1,  F1,  Limit,  EB).

$sum_at( [],      J, [],      [],        S1,F1,K,  Sum1,Sum2,EBs,EBs):-
      {Sum1==0,Sum2==0}.
$sum_at(  [W|Ws],J,  [S|Ss],[F|Fs],  S1,F1,J,  Sum1,Sum2,  Bs,EBs):-!,
      J1 is J + 1,
      {Sum1  ==  W  +  Sum_1,  Sum2==W  +  Sum_2},
      $sum_at( Ws,J1,   Ss,Fs,  S1,F1,    J,  Sum_1,Sum_2,Bs,EBs).
$sum_at([W|Ws],J,[S|Ss],[F|Fs],S1,F1,  K,   Sum1,   Sum2,
                                        [Q,Q1,Q2,P,P1,P2|Bs],  EBs):-!,
      {  Q1==  (F1=<S),  Q2  ==(F1=<F),
         Q ==(Q1  or  ~Q2),
         Sum2==  W*(~Q)  +  Sum_2,
         P1==  (S=<S1),  P2  ==(F=<S1),
         P ==(~P1  or  P2),
         Sum1==  W*(~P)  +  Sum_1  },
      J1 is J + 1,
      $sum_at( Ws,J1,   Ss,Fs,        S1,F1,  K,  Sum_1,Sum_2,Bs,EBs).
```

For many, and perhaps for most, purposes a good enumeration order is the one which enumerates the largest demands (duration*weight) first. This can be done by preordering the arguments.

Application to the Squares Puzzle

A classic benchmark which illustrates the use of cumulative is the squares problem: arrange a list of squares within a large square of given size. As a two dimensional layout problem, this will use cumulative along both the X- and Y- axis. This has been done with a version modified to work with integer variables (and integer summations), but we will not be doing any enumeration on the integer variables at all. Since the total area of the little squares just equals that of the big square in this problem, it is also very advantageous to be able to modify the code for cumulative to make the resource limit constraint an equality:

```
$resource_constraints([],       [],       K, Ws, S1, F1,   Limit,[]).
$resource_constraints([S|Ss],[F|Fs],  K,  Ws,  S1,  F1,    Limit,  B):-
      K1 is K + 1,
      $sum_at( Ws,1,  S1,F1,  S,F,  K,  Sum1,  Sum2,  B,EB), % print(Sum),
      {Sum1  ==  Limit,  Sum2==Limit},  % exact  fit
      $resource_constraints( Ss,  Fs,   K1,  Ws,  S1,  F1,  Limit,  EB).
```

The compatibility connection between the X and Y solutions is that no squares overlap. This can be forced by imposing some additional constraints between the enumeration maps of the X and Y subproblems: by checking which booleans in each cluster of 6 encode finish =< start disjointness conditions, we can ensure that at least one of the four conditions is true:

```
meld([],[]).
meld([_,Qx1,_,_,_,Px2|As],[_,Qy1,_,_,_,Py2|Bs]):-
      { 1== Qx1  or  Px2  or  Qy1  or  Py2},  % non-overlapping
      meld(As,Bs).
```

Here is the code for a *small* problem of this type:

```
squares(9,   [5,4,4,3,2,2,2,1,1,1]).

pack(  Size):-  stats,
      squares(  Size,  Sqs),
      location(  Sqs,  Size,  Xs,  Ys),
      cumulative(  Xs,  Sqs,  Sqs,  Size,  B1),
      cumulative(  Ys,  Sqs,  Sqs,  Size,  B2),
      stats(L,P,O,I,DT),  nl,  write([L,P,O,I,DT]),  %  setup  time
      stats,
      meld(B1,B2),
      enumerate(   B1),
      stats(L1,P1,O1,I1,DT1),   nl,   write([L1,P1,O1,I1,DT1]),%   1st   x   sol
      nl,write('Xs:'),print(Xs),
      enumerate(   B2),
      stats(L2,P2,O2,I2,DT2),   nl,   write([L2,P2,O2,I2,DT2]),%   1st   y   sol
      nl,write('Ys:'),print(Ys).
```

The execution trace is:

```
[291952,20705,4510,2894,15883]    %  set  up  costs

[57645,7203,259794,4322,25350]    %  time  to  get  first  solution  in  X
Xs:[4,  5,  0,  2,  2,  0,  0,  4,  0,  1]

[68318,9134,263546,5154,26584]    %  total  time  for  solution  in  X  and  Y
Ys:[4,  0,  5,  0,  3,  3,  0,  3,  2,  2]
```

Times are for 25Mhz 68030. The enumeration time to find the first X solution is almost 27 seconds. The final enumeration list of 540 booleans for X is:

```
       001100 100111 100010 100111 100111 100111 100010 100111 100111
001010        100111 100111 100111 100111 100111 100111 100111 100111
111100 111100        001100 001100 100010 100010 111100 100010 100100
001100 111100 100010        100010 100111 100111 001100 100111 100111
111100 111100 001010 001010        100111 100111 111100 100111 100111
111100 111100 001010 111100 111100        001010 111100 100010 001100
111100 111100 001010 111100 111100 001010        111100 100010 001100
001010 111100 100111 001010 100111 100111 100111        100111 100111
111100 111100 001010 111100 111100 001010 001010 111100        111100
111100 111100 001010 111100 111100 001010 001010 111100 100111
```

Note also that only a little more than an extra second is then required to find the first right fit in the Y direction because of the constraints between the enumeration lists, even though the 540 Y booleans are initially unknown when Y enumeration begins. (Note that there will be several Y solutions for each X solution because of symmetries.)

The cross-coupling between the X- and Y-enumerations suggests that it might be a good idea to interleave the two enumerations. Doing this on a block-by-block basis (i.e. the block of 6 booleans dealing with one pair of squares), leads to much better times, requiring only 5.2 seconds to jointly enumerate both:

```
[291952,20705,4510,2894,15984]
[63905,6193,21808,2232,5216]
Xs:[4, 5, 0, 2, 2, 0, 0, 4, 0, 1]
Ys:[4, 0, 5, 0, 3, 3, 0, 3, 2, 2]
```

There are several open questions at this point. One has to do with resolution: the same problem, also expressed as integers, but with everything increased by a factor of 10, was much slower on the original formulation (227 secs enumeration), suggesting that (although integer variables are not enumerated) integer rounding is playing a major hidden (and not understood) role. Curiously, the interleaved version showed only a slight increase (to 5.7 secs).

Another question is whether the use of cumulative is necessary at all, or would the systematic enumeration of just the no-overlap conditions be sufficient and perhaps even better? A step in this direction is to eliminate half of the sums from cumulative, retaining just the sums at the start of each task, since the rest are redundant. At the same time, half of the enumeration booleans are dropped, the revised code looking like:

```
$sum_at( □,     J, □,      □,       S1,F1,K,  Sum1,EBs,EBs):-
      {Sum1==0}.
$sum_at(  [W|Ws],J,   [S|Ss],[F|Fs],  S1,F1,J,   Sum1,   Bs,EBs):-!,
      J1 is J + 1,
      {Sum1 == W + Sum_1},
      $sum_at( Ws,J1,    Ss,Fs,  S1,F1,    J,  Sum_1,Bs,EBs).
$sum_at([W|Ws],J,[S|Ss],[F|Fs],S1,F1,K,Sum1,[P,Q1,P2|Bs],EBs):-!,
      {  Q1== (F1=<S),
         P1== (S=<S1),  P2  ==(F=<S1),
        P is (~P1 or P2),
         Sum1== W*(~P) + Sum_1 },
      J1 is J + 1,
      $sum_at( Ws,J1,   Ss,Fs,        S1,F1,  K,  Sum_1,Bs,EBs).
```

This version-- *with all real variables*-- cut setup costs almost in half and produced slightly better enumeration times as well:

```
?-  pack(9).
[167154,12067,2184,1628,9184]
[53520,4378,23368,1497,4600]
Xs:[4.0,  3.0,  0.0,  0.0,  7.0,  7.0,  0.0,  3.0,  2.0,  2.0]
Ys:[4.0,  0.0,  5.0,  2.0,  0.0,  2.0,  0.0,  4.0,  0.0,  1.0]
```

It is possible that even simpler and better versions may exist.

CLP(BNR) Algorithms for Traveling Salesman

William J. Older

BNR Computing Research Laboratory

August 3, 1993

ABSTRACT

This paper describes the development of several CLP(BNR) exact algorithms for the well-known traveling salesman problem. Although none of these algorithms are efficient enough for serious practical use, their development illustrates several principles of constraint logic programming: the difficulties posed by optimizations, the need for intrinsic characterizations of optima, the use of branch-and-bound techniques, and the occasional advantages of solutions which use a mixture of Prolog and CLP techniques.

This paper describes in detail how relational interval arithmetic can be applied to give a complete solution to a classical NP-complete problem, the famous "traveling salesman" problem. This problem is of intrinsic interest because of its numerous practical applications as well as its theoretical significance. But it also easy to describe and understand, so it is an interesting example to illustrate the application of CLP methods. Because it is usually thought of as an essentially combinatorial problem, a useful formulation of this problem in interval arithmetic terms is not completely obvious. It appears that this problem is not a "natural" candidate for the application of relational interval arithmetic. This has several advantages when it is viewed as a teaching example. First, most real-world problems will also not be entirely "natural" for this technology, and will sometimes require some ingenuity to formulate in CLP terms, or will require hybrid solutions. Also, the difficulties are themselves illuminating; it is just as important to see what cannot be conveniently done in CLP as it is know what can be done.

We will begin by describing the problem and then give two versions of a traditional Prolog solution, and then describe the practical performance difficulties with these solutions. In section 2 we will describe a reasonable (deterministic) sub-optimal heuristic solution, and examine its shortcomings. In section 3 we will extend the heuristic solution to a complete (nondeterministic) solution using interval arithmetic. Finally, in section 4, we extend this solution to get an effectively prunable branch and bound algorithm.

1. Traveling Salesman Problem

The general problem may be stated as: given N points in a metric space, find the shortest *tour* which visits all the points, where a tour is a path where the initial point is the same as the terminal point and which visits each point exactly once. In the specific case dealt with here, the metric space will be a region of the Euclidean 2-plane.

It will be convenient to take the input data as consisting of the locations of the N points, in the form of a relation:

```
          point( Label, X, Y ).
```

The output can be represented most conveniently as a list of labels. The initial (and terminal) point will be labeled 0 by convention.

We will need a relation to compute distances:

```
     distance( J, K, D) :-
          point(J,   XJ,YJ),
          point(K,   XK,YK),
          D is sqrt( (XJ-XK)**2 + (YJ-YK)**2).
```

If a tour is represented as a list of point labels, the total length of a tour is easily computed by:

```
tourlength(  Tour, 0,   0.0,   Length)    % 0 is standard origin
```

where

```
tourlength( [ ],    Last, Dist,    Total ):-
          distance( Last, 0,  D),
          Total is Dist + D.
tourlength( [X,Xs..], Last,   Dist,  Total) :-
          distance( Last, X  , D),
          Newdist is Dist + D,
          tourlength( Xs,  X,  Newdist,  Total).
```

A naive Prolog program (in the traditional declarative style) for computing the lengths of all paths is then easily written:

```
tour_and_length( Tour   ,  Length):-
          findall( P, point(P,_,_)  , List ),
          permutation( List,  L),
          tourlength( L, 0  , 0.0, Length).

permutation(  [],  []).
permutation( List,  [X,Xs..]  ):-
     delete( List,  X,  Newlist),
     permutation(  Newlist,   Xs).

delete( [X,Xs..], X,    Xs ).
delete( [Y,Xs..], X,    [Y,Ys..] ):-  delete( Xs,  X,  Ys).
```

The time required to generate all solutions is then aproximately proportional to (N-1)! * N. The extra factor of N (for computing the length of the tour) can be largely eliminated by interleaving the length computation with the permutation generation. This eliminates the recalculation of the the cost of the early steps of the tour for each tour. This can be done easily by "splicing" the two programs together with an editor (and possibly relabeling some variables).

```
tour_and_length(  Tour   ,  Length):-
      findall( P, point(P,_,_) , List ),
      perm_length( List, L, 0, 0.0, Length).

perm_length( [], [], Last, Dist, Total ):-
      distance( Last, 0, D),
      Total is Dist + D.
perm_length( List, [X,Xs..], Last,   Dist, Total) :-
      delete( List, X, Newlist),
      distance( Last, X , D),
      Newdist is Dist + D,
      perm_length( Newlist, Xs, X, Newdist, Total).
```

Both of these procedure can be used with a general minimization search algorithm which records in state space any solution which is better than any previous solution:

```
min_search(    Generator( MinResult, Solution, Parms..) ):-
           forget_all( Generator(_,_)),
           Generator( Res, Sol, Parms..),
           update_best( Generator, Res, Sol ),
          fail.
min_search(  Generator(MinResult,  Solution,  Parms..)  ):-
           recall( Generator( MinResult, Solution) ).

update_best(  G,R,S):-
           once(recall( G(OR,_) )),
          OR =< R,!.  % worse, so cut
update_best( G,  R,  S):-
           remembera( G(R,S) ).
```

The result is, of course, still much too slow to use for problems with more than a few points. With N=11, (N-1)! is already over three and half million, and the running time of this algorithm would be about an hour. Then 12 points would require about a work day; 13 about a week, 14 a season, and 16 about a half century.

One problem is that interval arithmetic, like Prolog, is naturally oriented towards determining feasibility, i.e., existential propositions of the form

exists(x) such that P(x)

for some vector of variables x and some predicate P. Optimization problems, however, have quite a different structure of the form:

exists(x) such that P(x) & forall(y) { P(y) => x<<y}} .

A search program of the above sort is the direct translation (assuming the comparison relation is transitive) of this formulation, and, in general, such a search will be necessary. The best approach in such a case is usually to discover intrinsic properties Q that any optimum point must have, and then restrict the search to the space of points satisfying Q as well as P:

exists(x) such that {P(x) & Q(x) & forall(y) { P(y) => x<<y}} .

Later we will see how we can apply this idea to the traveling salesman problem, but first we need to look at a somewhat different algorithmic approach.

2. Heuristic Algorithm

Faced with this combinatorial intractability, one can relax the problem and seek only a "good" tour. There is no formal definition for what constitutes a "good" tour, nor generally is there any way to judge just how far from the optimum it may be. There are many ways to find good tours- a human being with a pencil and a large sheet of paper often does very well! In this paper we will consider only one such heuristic algorithm, one probably rediscovered many times.

The starting point is the observation that the shortest tour will tend to be made of the shortest point-to-point segments, or "legs." An obvious "greedy" heuristic is therefore to construct a tour by picking legs in order of size. The most economical way to do this is to construct all the M=N(N-1)/2 legs and sort them by length to form a list. Items can then be sequentially picked from the list, subject to the constraint that the picked items must form a valid tour. Since this produces a list of selected legs in order of size, a step is needed to assemble the legs of the tour into the proper sequence. This yields:

```
heuristic(   Tour,     Length ) :-
       construct_legs(   Leglist  ,  N),
       sort(   Leglist,   Ascending ),
       select( N,  Ascending,    Selected ),
       sequence( Selected,  Tour,  0,  0.0,  Length).
```

We will see that these steps will have execution times roughly proportional to M, M*ln(M), M, and N, respectively, where M=N*(N-1)/2.

In order to use the standard sort predicate, we must construct the term for a leg so that the sort key (distance) is first; we will want to know which points are on each leg as well, so we consider the term with structure:

```
leg(   Distance,    PointA,  PointB  ).
```

(Note that legs are unoriented; the leg AB is to be regarded as the same as leg BA. This cuts the number of legs in half, but means that the proper orientation must be done as part of the sequencing.)

With these ideas in mind we can code the first predicate thus:

```
construct_legs(  Leglist  ):-
      findall(   pt( Label,  Component,  [Next,  Prev],  [X,Y]),
                  mk_pt( Label, Component,[Next,  Prev],[ X,  Y]),
                 Point_list ),
      list_length(  Point_list,N),
      distance_list(  Point_list,  Leglist).

mk_pt( Label,  Component,  [Next,Prev],  [X,Y]):-  point( Label,X,Y).

list_length( [],  N,  N).
list_length( [X,Xs..],M,N):-  M1  is  M  +  1,  list_length(Xs,M1,N).

dist(  pt(_,_,_,[X1,Y1]),   pt(_,_,_,[X2,Y2]),   D):-
      D  is  sqrt(  (X2-X1)**2  +  (Y2-Y1)**2.

      % form  the  list  of  N*(N-1)/2  pairs
distance_list( [],  [] ).
distance_list( [C,Cs..],  DL  ):-  dist_from(  Cs,C,  DL,EL),
      distance_list(  Cs,  EL).
      %  legs  from  list  of  points  to  a  point
dist_from( [],    P,          L, L ).
dist_from(  [X,Xs..],Y,   [leg(D,X,Y),Ls..],   E):-
      dist(X,Y,D),
      dist_from(Xs,Y,Ls,   E).
```

The key issue is to ensure that only collections of legs forming valid tours are selected. One constraint is then that each point must be visited exactly once, i.e. used in exactly two legs. This can be accomplished most easily in Prolog by associating two unbound variables `[Next,Prev]` with each point, and then binding them as legs are selected.

The second constraint is to ensure that the tour forms a single cycle rather than several smaller cycles. This can be restated as a rule that a leg should never be chosen if it will connect to points which are already connected, unless this is the last leg to be selected. A Prolog implementation of this is to add an unbound variable `Component` to each point, and unify these variables for the points connected by a leg (so the variables represent connected components of the graph), but only if they are not already identical (@=), except for the last leg.

The main algorithm now has a very simple structure:

```
select( 0, List,    [] ):-!.% stop  when  N  are  selected
select(  N,   [L,Ls..],   [L,Rs..]):-   check_constraints(L,N),!,
         N1 is N - 1,
           select(N1,Ls, Rs).
select( N,[L,Ls..], R):- select(N, Ls, R).           % else skip

check_constraints(leg(_,  pt(P,  C1,  Plink,_),  pt(Q,  C2,  Qlink,_)),N):-
           different_components( C1,C2,N),
           link(P, Qlink), link(Q,Plink).

link( X,  [X,  _]).
link( X,  [_,  X]).

different_components(C1,C2,N):-   N<>1,   C1@=C2,!,fail.
different_components(C,  C,  N).   % succeed  and  unify  components
```

The final sequencing is also straightforward:

```
sequence( □, □, 0 , D, D):-!.   %last leg returns to origin
sequence( List,  [P,Ps..],  P,  Length_so_far,  Length):-
      delete_leg( List,  leg(D,  pt(P,_..),pt( Next,_..),  Rest),!,
      D2 is Length_so_far + D,
      sequence( Rest,  Ps,  Next,  D2,  Length).

delete_leg(  [X,Xs..],  Y,  Xs):-  match_leg(X,Y),!.
delete_leg(  [X,Xs..],  Y,   [X,Ys..]):-  delete_leg(Xs,Y,Ys).

match_leg(  leg(D,A,B),  leg(D,A,B)).
match_leg(  leg(D,A,B),  leg(D,B,A)).
```

If we run this algorithm on our test problem, we get a tour length of about 84.2, not far from the optimal tour of 77.547 . If we examine the tour itself, we find that it agrees with the optimal tour (or its reverse, which is also optimal) almost everywhere. However, unlike the optimal tour, its path crosses itself. This suggests that we take a closer look at self-crossings.

A Geometrical Lemma

Recall that a metric d satisfies the axioms:

d(X,Y) >= 0
d(X,Y) = d(Y,X)
d(X,Z) =< d(X,Y) = d(Y,Z).

In any metric space we can define the 3-ary relation
between(X,Y,Z) iff d(X,Z) = d(X,Y) + d(Y,Z).
(There may not be any points between two given points, depending on the particular metric space.)

Lemma: If points A,B,C,D in a metric space are such that there is a point W where between(A,W,C) and between(B,W,D), then
d(A,D) + d(B,C) =< d(A,C) + d(B,D)
and d(A,B) + d(C,D) =<d(A,C) + d(B,D) .
Proof: From the triangle inequality we have
d(A,D) =< d(A,W) + d(W,D)
and d(B, C) =< d(B,W) + d(W,C),
so d(A,C) + d(B,D) =< d(A,W) + d(W,C) + d(B,W) + d(W,D)
= d(A,C) + d(B,D),
since W is between A and C, B and D. The other inequality is proved similarly.

In ordinary two-dimensional Euclidean space, for any four points, no three of which are colllinear, there will be at most one of the three pairs of segments (with distinct endpoints) which has a crossing. The situation can be more complex with non-Euclidean metrics, but this theorem remains valid.

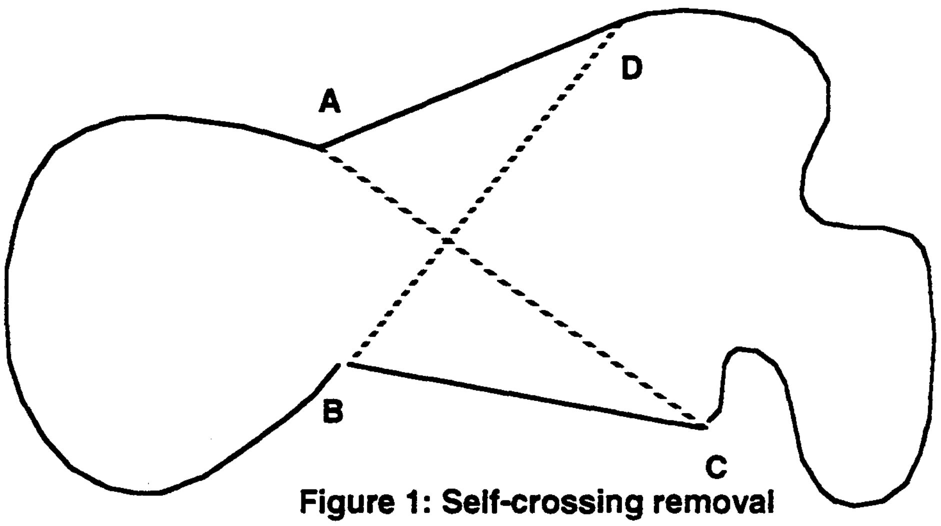


Figure 1: Self-crossing removal

Now suppose we have an optimal tour in the Euclidean plane with a self-crossing, for example leg A-C crosses leg B-D. Removing these two legs breaks the tour into two pieces, and suppose A and B are in one piece and C and D in the other. We can then construct a new tour by adding legs from A-D and B-D, and by the above the result will be better (or the same) as the original tour. Thus we infer that a minimal tour (but not neccessarily all minimal tours) are non-self-crossing; so "non-self-crossing" is a possible intrinsic property that can be used to reduce the search space.

One way that we might try to use this property is this: whenever we select a leg, any leg which crosses it becomes inelgible for selection. In this way we would only generate tours free of crossings. This requires that the selection algorithm be non-deterministic and we must remove the cut in the second clause.

3. Basic Constraint Algorithm

In this section we begin formulating a constraint based algorithm. The first problem is to find a representation of the problem in terms of "arithmetic" variables, including integer and boolean. Since the problem involves a permutation, one thinks naturally of a *permutation matrix*, that is, a matrix p(i,j) (denoting "from i goto j") with boolean entries such that row and column sums are equal to 1. However, proper tours have additional constraints: (a) the diagonal entries p(i,i) are 0, and (b) if p(i,j)=1 then p(j,i)=0. These added constraints can be used to reduce the number of variables by more than a factor of two, if we use instead of the p(i,j) the new variables p'(i,j) := p(i,j) or p(j,i) for i<j. These are easily seen to be essentially equivalent to the unoriented "legs" used above. The row and column sum constraints are replaced with: the sum of all leg variables attached to a point is 2, as shown in fig. 1. (This replaces the link fields used earlier.)

The same strategy that was used before for ensuring only single-cycle permutations are generated can be used here also. In fact, there does not appear to be any practical way of achieving the same result using CLP techniques at all. Here is one of the places where a hybrid aproach is neccessary.

The previous algorithm can now be transformed easily into an equivalent constraint- based algorithm (changed parts are shown in italics, some secondary simplification has also been done):

```
salesman1(  Tour,    Length ) :-
      construct_legs(   Leglist  ,  N),
      sort(   Leglist,   Ascending ),
      select( N, Ascending,   Selected ),
      sequence( Selected, Tour, 0, 0.0, Length).

construct_legs(  Leglist   ,N):-
      findall(  pt( Label,  Component,  [X,Y]),
                mk_pt( Label, Component,   X,  Y),
                 Point_list ),
      list_length(  Point_list,N),
      distance_list(  Point_list,   Leglist),
      M is N - 1,
         setup_dst_constraints( M,   Leglist). % nl, print( Leglist).


mk_pt( Label, Component,   X,  Y):- point( Label,  X,Y).

dist( pt(_,_,[X1,Y1]),  pt(_,_,[X2,Y2]),  D):-
      D is  sqrt( (X2-X1)**2  +  (Y2-Y1)**2.

      % form  the  list  of  N*(N-1)/2  pairs
distance_list( [],  [] ).
distance_list(  [C,Cs..],   DL  ):-  dist_from( Cs,C,   DL,EL),
      distance_list(  Cs,   EL).
      %   legs  from  list  of  points  to  a  point
dist_from( [],         P,          L, L ).
dist_from(   [X,Xs..],Y,    [leg(D,P,X,Y),Ls..],    E):-
        P:boolean,    % make  boolean  constraint  variable  to  record  choices
      dist(X,Y,D).


setup_dst_constraints(  -1,  L  ):-  !.
setup_dst_constraints(  N,  L  ):-  N1  is  N  -  1,
          incident(L,  N,  LN ),
         sum(  LN,  S ),
          S == 2,    % total  degree  in  tour  is  exactly  2
           setup_dst_constraints(  N1,  L ).

incident(  [],  _,  []).
incident(  [X,Xs..],   N,   [P,Ys..]):-   incid(X,N,P),!,
           incident(Xs,N,Ys).
incident(  [X,Xs..],   N,    Ys):-   incident(Xs,N,Ys).


incid(  leg(D,P,  pt(N,_..),_),   N,   P).
incid(  leg(D,P,  _,pt(N,_..)),   N,   P).


sum(  [X],   S):-S==X,!.
sum(  [X,Xs..],  N ):-  S:integer,  N == X + S,  sum(  Xs,S  ).


select(  1,  [leg(D,1,P,Q),_..],   [leg(D,P,Q)]  ):-!.
            % stop  when  last  one  is   selected
select(  N,  [leg(_,1,P,Q),Ls..],  [leg(D,P,Q),Rs..]):-
          check_components(P,Q),!,
         N1 is N - 1,
          select(N1,Ls, Rs).
select(  N,[leg(_,0,_..),  Ls..],  R):-
         select(N,  Ls,  R).        % else skip

check_components(pt(_,C1,_),   pt(_,C2,_)):-
           different_components(  C1,C2).

different_components(C1,C2):-     C1@=C2,!,fail.
different_components(C,  C  ).   %  succeed  and  unify  components
```

This constraint-based algorithm can now be extended to use the crossing-blocking strategy discussed above. For this we need to first move the cut out of select; for a heuristic algorithm we can keep the cut around the call for select. Secondly, we need to impose an additional constraints for each pair of crossing legs. These changes transform the main predicate into:

```
salesman2(   Tour,     Length  ) :-
      construct_legs(    Leglist   ,   N),
      sort(    Leglist,    Ascending  ),
      crossover_constraints(   Ascending    ),
      select(  N,  Ascending,    Selected  ),!,
      sequence(  Selected,  Tour,  0,  0.0,  Length).
```

where we have added

```
crossover_constraints(   []   ).
crossover_constraints(    [L,Ls..]):-
      cross_constrain_all(  Ls,   L),
      crossover_constraints(   Ls).

cross_constrain_all(   [],_).
cross_constrain_all(  [L,Ls..],   K)   :-
      cross_constrain(  L,   K),
      cross_constrain_all(  Ls,   K).

cross_constrain(  L,  K  ):-  crosses(  L,  K),!,  notboth(  L,  K).
cross_constrain(  _,  _  ).

notboth(  leg(_,P,_..),  leg(_,Q,_..):-   P+Q  =<  1.
```

The crossing test, once adjacent legs are eliminated, can also make effective use of interval arithmetic:

```
crosses(  leg(_,_,pt(I,_,[X0,Y0]),  pt(J,_,[X1,Y1])  ),
           leg(_,_,pt(K,_,[X2,Y2]),  pt(L,_,[X3,Y3])  ) ):-
       not( adjacent_legs( I,J,K,L) ),
      [S,T]:   real(0,1),
      S*X0 + (1-S)*X1 = T*X2 + (1-T)*X3,
      S*Y0 + (1-S)*Y1 = T*Y2 + (1-T)*Y3,
      solve(S,T).

adjacent_legs(X,J,X,L).
adjacent_legs(X,J,K,X).
adjacent_legs(I,X,X,L).
adjacent_legs(I,X,K,X).
```

Note, however, that the setup costs for these constraints are fourth order in N, so that computing all these crossovers becomes quite expensive.

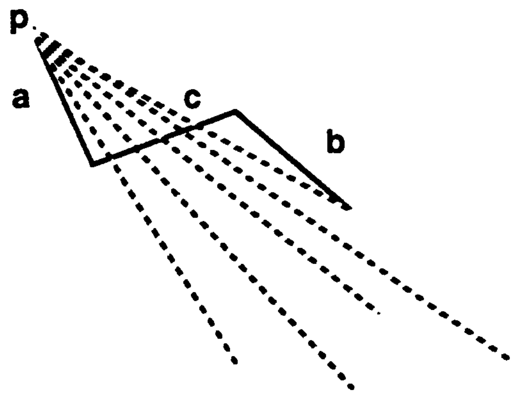


Figure 2 : Leg c is skipped because of the interaction between constraints

With crossover prohibition constraints in place one finds that there is sometimes a significant amount of pruning of the search space which appears surprising at first. For example, in Figure 2. above, we have a case where the algorithm first chooses leg a, then leg b, and suppose leg c would then be the next choice. However, when leg c is chosen, the constraints trigger an immediate failure, so leg c is skipped. The reason for this failure is that choosing leg c requires that all the legs crossing c must become inelgible, but since this includes all the remaining legs incident at point p, this leads to a contradiction with the requirement that we must choose two legs incident at p.

Because of this interaction between crossover constraints and incidence constraints (and in general with other constraints to be added later) the algorithm begins to exhibit new and sometimes surprising properties. This synergy phenomenon, in which separate (and often weak) constraints interact to produce a strong and surprising result, is a novel characteristic of CLP programming.. Note that, had we not moved the incident restrictions into the constraint network, this synergy would not have happened, but that after choosing leg c, all branches of `select` would have eventually failed.

In view of this, and the geometrical lemma above, one might expect that the first solution of the crossing-blocked algorithm would be better than the first solution with crossovers. However, generally the reverse is true: the additional constraints make the first solution worse rather than better. So, as a heuristic algorithm, we are now worse off. The benefits of blocking crossovers only appear when we run the algorithm non-deterministically, because it reduces the search space.

4. Branch and Bound

At this point we would like to remove the cut and return to a complete, non-deterministic algorithm. However, the number of crossover-free tours (which is the set we are searching with this algorithm) is still too large to be practical for any but very small values of N.

In order to prune the search space further we introduce a constraint version of the branch and bound strategy. This requires that we maintain an interval estimate of the length of a tour and a global bounding value based on the best tour seen so far with the intent that we thereby cut off the search whenever the estimate's lower bound is above the global bound. Each new (and better) solution must then alter (as a side-effect) the global bound.

For this purpose we add two additional fields SP (an integer variable) and SPD (a real variable) to the leg structure: `leg( D, P, SP, SPD, X, Y ).`

The variable SP represents the partial sum of the P booleans for this and all previous legs, while SPD is the partial sum of P*D for this and all subsequent legs. This additional constraint structure is constructed by a call to `bnb_constraints( Leglist, Number, Nsofar, Total_Dist).`

The mainline of the algorithm is then changed to call this additional setup routine, set up the global bound inequality, and to update the global bound for each new solution. It uses a special CLP(BNR) primitive (set_upper_-bound) for performing this last action as a side-effect, which will not be undone by backtracking. At the end, the optimal (i.e. the last) solution is retrieved from state space.

```
salesman_bnb(  Tour,     Length ) :-
      construct_legs(   Leglist  ,  N),
      sort(   Leglist,    Ascending ),
         crossover_constraints(  Ascending),
       bnb_constraints(  Ascending,   N,   0,   Total)),
        Bound:  real(0,_),
       Total  =<  Bound,
         foreach(  select1(  N,   Ascending,   Selected,   Ct)
            do    [ remembera( tour( Selected) ),
                    range(Total,[_,U]),
                    set_upper_bound( Bound,U)         ]),
        recall( tour( Selected)),    % get final answer
        sequence( Selected, Tour, 0, 0.0, Length).
```

One possibility for the setup routine would be this:

```
bnb_constraints( [], _, _, 0.0).
bnb_constraints( [ leg(D,P,SP,SPD,_,_), Ls..], Number, OSP, SPD):-
        SP == OSP + P,
          bnb_constraints( Ls, Number, SP, OSPD),
        SPD == OSPD + P*D,
         SPD >= (Number - SP)*D.
```

Notice that the first line in the second clause computes SP as a partial sum of previous P's, while the third line computes SPD as a partial sum of subsequent P*D's.

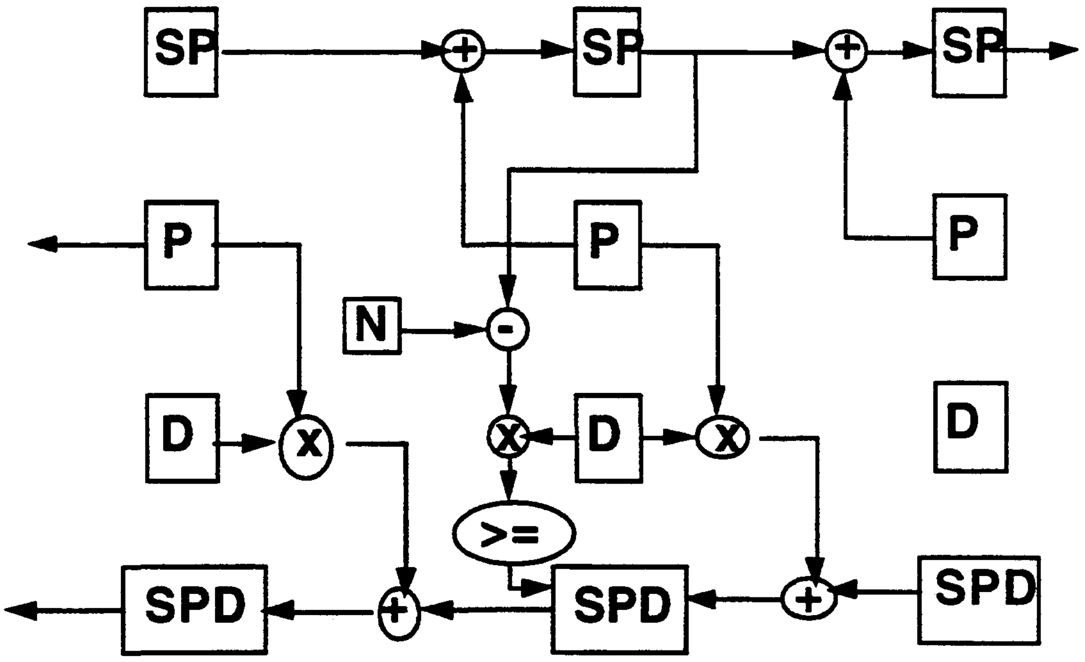


Figure 3.A portion of the constraint network

This is not good enough, however, since the lower bound of SPD will be 0.0 so long as no subsequent P's are equal to 1; for this reason the total value returned by bnb_constraints has a lower bound which just reflects the currently chosen legs. This lower bound is too low to get much pruning from the global bound. For this reason, the last line in the second clause adds a *bounding* *constraint*: the rest of the choices (of which there are : Number - SP) are each at least as big as D, so SPD >= (Number - SP)*D. Note that this estimate is applied at every stage so that many lower bounds are being computed "in parallel" and the largest of these becomes the lower bound of the total cost of the tour. The largest ofthese bounds would come from legs with large D, except that past the current choosing point of the select algorithm, the other factor (Number - SP) becomes increasingly uncertain because of SP. Usually, the effective bound will comes from at or just ahead of the choosing point.

With this mechanism in place the number of choices K1 drops rather dramatically. Direct measurements yield:

N	7	8	9	10	11
(N-1)!	720	5040	40320	362880	3628800
K1	18	55	230	448	1378

These mechanisms have thus reduced the complexity of the problem significantly, from (N-1)! to a mere approximate 2**(N-1). Practically speaking, however, the cost of the constraint processing, which seems to be roughly proportional to N*N, means that the running times to do not show the same improvement. For the N=11 case, for example, the improvement is only about a factor of 10. Thus despite the fact that on larger problems one might be able to reduce the running time from say a century to a mere few months, this is not a practical solution in the usual sense.

Therefore, in order to improve the pruning in the branch and bound, and reduce the constraint costs for doing estimates, in our last version we will move the bounding estimation back into Prolog. We no longer need to have the SP variables, so we can simplify:

```
bnb_constraints( [],     0.0).
bnb_constraints(  [  leg(D,P,SP,SPD,_,_),  Ls..],  SPD):-
        bnb_constraints( Ls,    OSPD),
       SPD == OSPD + P*D.
```

The select algorithm, however, becomes more complex:

```
select( 1,  [leg(D,1,_,_,P,Q),_..],  [leg(D,P,Q)]  ):-  !.
select( N,   [leg(D,S,SPD,P,Q),Ls..],   R):-
        var(S) -> [ estimate(N, Ls,D, TD),
                      SPD>= TD   % connect estimate to constraint net
                  ],
        sel( S, leg(D,P,Q),  N, N1, R,R1),
        select1( N1, Ls, R1).

sel(  1,  leg(D,P,Q),  N,  N1,[leg(D,P,Q),Ls..],Ls):-
        check_components(P,Q),
      N1 is N - 1
sel( 0, Leg,   N,N,  L,L).
```

In this hybrid solution we compute an explicit lower bound by summing over the next n possibly chooseable legs; since this ignores any constraints between them, it is obviously a lower bound.

```
estimate(1,_,   D,D):-!
estimate(N,   [leg(D,S,_..),Ls..],   D1,   TD):-   S@=0,!,   estimate(N,Ls,D1,TD).
estimate(N,[leg(D,S,_..),Ls..],D1,TD):-
      N1 is N - 1, DD is D1 + D,
       estimate( N1, Ls, DD, TD).
```

This lower bound is then connected directly into the SPD variable, where it propagates back to the comparison against the global bound. With this version, which computes a slightly tighter lower bound, we get a further reduction in the number of choices K2.

N	7	8	9	10	11	12	13
(N-1)!	720	5040	40320	362880	3628800	39916800	479001600
K1	18	55	230	448	1378	?	?
K2	16	46	166	287	911	3403	9805

This version is also somewhat better in time because of the reduction in the amount of constraint activity devoted to lower bounds, with the search time for N=11, for example, reduced to a couple of minutes versus the hour required by the naive algorithm. The lower bound estimator is still very crude, however, and an investment in finding a tighter lower bound estimator would most likely produce large benefits.

Conclusion

In this paper we have gone through the first steps of an iterative development cycle for a partially constraint-based approach to a difficult problem. We have examined several strategies for pruning search spaces and looked at the mechanism for implementing branch-and-bound, and seen how these act to reduce the amount of non-determinism by large factors. The resulting algorithm is, of course, still exponential as it must be (unless P=NP). In any particular problem, depending on the details of the formulation, the resulting algorithm may or may not be useable for realistic-sized problem instances; only by doing the experiment can we tell what the practical limits will be. CLP(BNR), which makes the job of writing correct and complete solutions very easy, makes such empirical investigation feasible. Each improvement in the algorithm permits larger sample problems to be handled, and these serve as test cases for the next generation algorithm. Finally, and most important, each advance in theoretical knowledge (such as the geometry lemma in this example) translates directly into additional constraints and hence a reduced search space.