Search BYTE:

Search

BYTE Categories
Columns
Features
Audio

Print Archives
By Issue
By Topic

Resources
History Of Byte:
Part I
Part II

Java Books
More Java Books
Java One Audio Report

BYTE Humor
Ian Shoales' Page

Newsletter
Sign Up Now

News
TechWeb
Stocks/Finance

Off Hours
Games
Family
Cartoons

Horoscopes

Free E-mail
Sign Up Now

About Us
Editorial Staff
Feedback
Subscriber Inquiries
CMP Media Inc.
Link To Us
Privacy Policy

Sponsored by:

ARTICLES

Constraint Logic Programming

February 1995 / Core Technologies / Constraint Logic Programming

A child of Prolog finds new life solving programming's nastier problems

Dick Pountain

Which programming paradigm will likely gain the most in commercial significance over the next five years? I'd bet on CLP (constraint logic programming), even though few programmers today understand it well. CLP's potential lies in its power to tackle difficult combinatorial problems--such as those encountered in job scheduling, developing time tables, and routing--that stretch conventional programming techniques beyond their breaking point. Though CLP is still the subject of intensive research, it's already being used by large corporations, including the manufacturers Michelin and Dassault; the French railway authority SNCF; the airlines Swissair, SAS, and Cathay Pacific; and Hong Kong International Terminals, one of the world's largest privately owned container terminals.

Children of Prolog

As its name suggests, CLP is descended from logic programming, which shot to fame via the Prolog language, widely used in the Japanese 5th Generation project and the expert-systems boom of the mid-1980s. Its relatively poor efficiency (compared to procedural languages like C) hindered Prolog's commercial acceptance, and its use has declined in recent years. Now, by focusing on a particular problem domain, CLP languages make logic programs execute efficiently.

Prolog is based on first-order predicate logic, and the objects that it manipulates are pure symbols with no intrinsic meaning. For example, in the Prolog proposition ‘‘likes (jim, baseball)’’ the constants ‘‘jim’’ and ‘‘baseball’’ have no deeper interpretation beyond syntactic identity (i.e., jim = jim). Execution of a Prolog program proceeds by a process called unification, which searches a database of such facts and finds those values that will satisfy a user’s query. Unification is based on syntactic identity. Since Prolog tries to find the set of all solutions to a query, during this search, a program may encounter many dead-ends to explore and then abandon by backtracking to an earlier state and trying a different branch. For complex problems, this search process can become greedy in both space and time, which is the root of Prolog’s inefficiency.

In a CLP language, objects that have meaning in an application domain--for example, the integers or the real numbers, with their associated algebraic operations (e.g., addition and multiplication) and predicates (e.g., =, <, and >)--supplement this purely abstract logical framework. Hence, there isn’t a single CLP language but a whole family of them defined for different application domains. A CLP programmer introduces arithmetic expressions called constraints (e.g., $X > 0$ or $Y + Z < 15$) into program s, which have to be satisfied for successful execution of the program. (For a more formal explanation of how CLP works, see ‘‘Theme: Prolog,’’ August 1987 BYTE).

In such a CLP system, the simple unification algorithm that lies at the heart of Prolog must be augmented by a dedicated solver for the particular domain of application. The solver can decide at any moment whether the remaining constraints are solvable. For efficiency’s sake, solvers for CLP systems need to be incremental so that adding a new constraint to an already solved set does not force them all to be solved a second time. Constraint-solving algorithms are quite well understood from other branches of computing; you’ll have used one if you’ve ever done goal-seeking in your Excel spreadsheet. For example, a useful solver for linear rational constraints is the well-known simplex method.

Another significant way in which CLP differs from Prolog is that it’s perfectly happy to do mathematics with uninstantiated variables; therefore, in the absence of complete information, the answer might be a symbolic expression like $10 - X$ or even a constraint like $X > 23$.

Constrained Search

A CLP program still needs to search a database of facts, but it can use constraints to rule out many possible outcomes and prune away large parts of the search tree. The improved efficiency that results is comparable to custom solutions written in C.

We all use facts as constraints to guide reasoning as a key part of everyday common sense. For example, a few minutes ago, a public-relations person called to ask if I'm interested in document management and to alert me to a press briefing next Wednesday in London. A glance at my calendar revealed that I'll be in Cambridge all next Wednesday--end of conversation. We no longer needed to explore my interest (or lack thereof) in document management because an absolute geographical constraint had lopped off that branch. Without such constraints, every little decision might set off an av alanche of philosophical speculation.

Herbert A. Simon, Nobel laureate and theorist of heuristic problem-solving, has used popular word-for-number puzzles to illustrate this pruning process. For example, in the puzzle DONALD + GERALD = ROBERT, there are 3,628,800 possible assignments of digits to letters, and it would take you several years to solve the problem by unconstrained search. Yet most of us can solve it in just minutes by incrementally applying constraints (e.g., T must be even) to rule out more and more options. ''An Eclipse program to solve the DONALD + GERALD = ROBERTWord Puzzle'' shows a typical CLP program to solve this puzzle. (Mark Wallace of IC Parc wrote the solution.)

Slaying NP-Hard Dragons

This constrained-search ability makes CLP languages good at precisely those problems that conventional programming techniques find hardest: NP-hard search problems where the time needed for an unconstrained search increases exponentially (or worse) with the problem size.

Consider the simple problem of a commercial harbor that needs to schedule the loading and unloading of 10 ships using only five berths. There are many criteria for choosing the berth for a particular ship: Some berths are too small for some ships, some ships need to be turned around faster than others, some berths cost more than others, ships' intended cargoes are stacked nearer to certain berths, and so on.

You can find the optimal schedule by trying all permutations of ships in berths and calculating the cost of each, which means considering 510 (or around 10 million) alternatives. Assuming that your computer can try an alternative every millisecond, it can solve the whole problem in around 3 minutes. Now imagine it's a decade later, and business has been good and the harbor has expanded to 10 berths, with 20 ships to unload. Determining the optimal schedule now means trying 1020 alternatives, which will take 3000 million years on the same computer (of course, you can a nte up for an accelerator card and cut that to 300 million years).

There are many other problems in planning and scheduling that exhibit this kind of unreasonable scaling behavior for which an exhaustive search is not a feasible strategy. So how do you solve these problems? A naive but tempting approach is to divide the harbor in two and schedule each half using the old program, taking 6 minutes in all. Unfortunately, such a schedule is unlikely to be anywhere near optimal, and worse, you won't even know how far from optimal it is and how much money you are wasting. Actually running the 3000-million-year program for 6 minutes and choosing the cheapest alternative so far would give just as good (or bad) a result.

Where CLP languages score for this class of problem is that you can explicitly employ all the real-world constraints that are particular to the problem and so reduce the search space enormously. In our harbor example, adding a constraint like ‘‘shipLength <berthLength’’ might immediat ely remove millions of possibilities.

Languages like CHIP (Constraint Handling in Prolog) and Eclipse offer direct control over the search strategy (via the ‘‘deleteff’’ function in the word-puzzle solution). If this still doesn't yield an optimal solution in reasonable time, you must then deploy approximation algorithms to reach a solution that lies close to the optimum with a high degree of probability. Researchers are working hard to integrate algorithms like hill-climbing, simulated annealing, and genetic algorithms into the newer CLP languages.

Don't get the idea that CLP can perform magic. You need a great deal of experience before you can choose the correct algorithms and correct expression of the constraints to get a good solution for big problems. Nevertheless, the interactive nature and highly expressive power of CLP languages makes it easy to experiment with different combinations. This results in much shorter and more maintainable programs than when using a procedural language.

CLP Implementations

The founding work on the CLP scheme was done at Monash University in Melbourne, Australia, by J. Jaffar and J. L. Lassez around 1987. They created the CLP(R) system, which works on the domain of real linear arithmetic. This system, extended as CLP(X), is still being developed at Monash as well as in the U.S. at IBM's Yorktown Heights research facility and at Carnegie Mellon University.

In Europe, CLP research was originally concentrated at the ECRC (European Computer-Industry Research Centre) in Munich, and it led to what is probably the most widely known CLP system at present: CHIP. The French company Cosytech commercially markets CHIP, and two of ECRC's industrial sponsors, Bull (as Charme) and ICL (as Decision Power), sell CHIP derivatives. CHIP provides constraint solvers over finite arithmetic, linear rational, and Boolean domains.

In 1990 in Marseilles, France, Alain Colmerauer (one of the founding fathers of Prolog and CLP) created Prolog III , a CLP language over the domains of linear rational arithmetic, Booleans, and finite strings or lists. More recently, London's Imperial College has set up IC Parc, a university/industry collaboration that uses CLP techniques to tackle hard planning problems, such as work-force management, routing, and resource allocation. IC Parc's language, Eclipse, began life at ECRC and shares many features with CHIP; but where CHIP's constraint solvers are hard-coded in C, Eclipse's are written in itself for easier modification. Eclipse also employs the powerful generalized propagation technique, and a parallel version of Eclipse is currently under development.

Interesting non-Prolog-based CLP languages include Trilogy, from the Vancouver-based Complete Logic Systems, and Oz, an object-oriented concurrent CLP being developed at DFKI (German Research Center for Artificial Intelligence) in Kaiserlautern.

An Eclipse program to solve the DONALD + GERALD = ROBERT

```
word puzzle.


l
ib(fd).
addNum(Num1,Num2,Result) :-
 addwithcarries(Num1,Num2,0,Result).
addwithcarries(Num1*Digit1,Num2*Digit2,Carryin,Num3*Sum) :-
 Carry::0..1,
 addDigit(Carryin,Digit1,Digit2,Sum,Carry),
 addwithcarries(Num1,Num2,Carry,Num3).
addwithcarries(Digit1,Digit2,Carryin,Digit3) :-
 is_domain(Digit1), is_domain(Digit2),
 addDigit(Carryin,Digit1,Digit2,Digit3,0).
addDigit(C,N1,N2,Sum,CIn) :- Sum #= C+N1+N2-10*CIn.
```

```
solve :-
 Letters=[D,O,N,A,L,G,E,R,B,T],
 Letters::0..9,
 alldistinct(Letters),
 addNum(D*O*N*A*L*D,G*E*R*A*L*D,R*O*B*E*R*T),
 label(Letters),
 writeln([D,O,N,A,L,D]),
 writeln([G,E,R,A,L,D]),
 writeln([R,O,B,E,R,T]).
label([]).
label(List) :-
 deleteff(Digit,List,Rest),
 indomain(Digit),
 label(Rest).
```

Dick Pountain is a BYTE contributing editor based in London, U.K. You can reach him on the Internet or BIX at dickp@bix.com
.

Up Level Previous Next Search Comment Subscribe