Problem Solving with Constraints and Programming

Gert Smolka
DFKI and Universität des Saarlandes
Postfach 15 11 50, Geb. 45
D-66041 Saarbrücken, Germany
smolka@ps.uni-sb.de
http://ps-www.dfki.uni-sb.de/~smolka/

I sketch a general model of constraint-based problem solving that is not committed to a particular programming paradigm, show that Prolog in particular and logic programming in general do not provide a satisfactory framework for constraint programming, and outline how constraint programming is realized in Oz, a general-purpose language for symbolic processing.

Since 1990, I have led a group at DFKI that is designing, implementing and applying the concurrent constraint programming language Oz. Oz is a general-purpose language for symbolic processing. It is distinguished from other languages by providing both for concurrent programming and constrained-based problem solving. An Oz program can create any number of concurrent agents, each equipped with its own inference engine for constraint-based reasoning.

An efficient public domain implementation of Oz has been released by DFKI in January 1995, and the language is now in world-wide use for research and educational purposes. The constraint facilities of Oz 1 are experimental. This has changed in Oz 2, which provides fully developed constraint facilities for solving combinatorial problems over integers (i.e, scheduling). We expect the beta release of the DFKI Oz 2 system for early 1997.

Since 1995, SICS has joined DFKI in developing and promoting Oz. We have started an ambitious joint project to design and implement a distributed version of Oz. Other projects at DFKI and SICS explore the use of Oz for scheduling, natural language processing, constrained-based planning, configuration, and multi-agent systems.

In the following, I sketch a general model of constraint-based problem solving that is not committed to a particular programming paradigm. I will then argue why programming is needed and consider which features a programming language implementing the model should offer. From that it will become clear that Prolog in particular and logic programming in general do not provide a satisfactory framework for constraint programming (although they have been good starting points for the development of constraint programming). The discussion will outline the main ideas of the way constraint programming is realized in Oz.

1. WHAT IS CONSTRAINT PROGRAMMING?

Constraint programming stands for a wide spectrum of software technology that is characterized by embedding computation with constraints into a programming language. One area where the constraint programming approach has been particularly successful is solving combinatorial problems over integers (e.g., scheduling). Another successful area is natural-language processing (where constraint programming in fact originated). Here the constrained variables range over feature trees rather than integers. A third and rather different area related to constraint programming is the design of high-level concurrent programming languages, which take the notion of logic variable and constraint store as a simple and powerful means to provide for communication and synchronization of concurrent computation.

Historically, constraint programming grew out of the idea of combining logic programming as realized in Prolog with constraint-based problem-solving methods originating in artificial intelligence, operations research, and computational logic. Three languages pioneering this combination in the 1980s were CHIP (constraints over integers and rationals), Prolog III (constraints over rationals and infinite trees) and LOGIN (constraints over feature trees). Taking Prolog as the programming language was a natural choice since it provides three essentials of constraint programming: search, logic variables, and a well-defined relationship between computation and predicate logic. No other programming model provided these crucial features.

However, it has become clear that Prolog is neither the only possible nor a fully satisfactory platform for constraint programming. Ilog's Solver is a commercially successful C++ library providing constraint programming over integers. Oz is a concurrent constraint language that integrates constraint computation into a framework much richer than logic programming.

2. PROBLEM SOLVING WITH CONSTRAINTS

I now sketch what I think is the basic method underlying constraint-based problem solving. At this level of abstraction there is no immediate need for programming and hence no commitment to a particular programming model. Why programming is needed is discussed in the next section.

To start with, we assume a fixed first-order structure and call the formulas over its signature constraints. For different applications we may have different structures with different constraints.

A problem must be represented as a set of constraints such that the solutions of the constraints in the structure are the solutions of the problem. To solve the problem, we need a method that computes the solutions of the given set of constraints.

For reasons of computational feasibility, we are not interested in a fully automatic method. Rather, we direct the inference engine implementing the method with additional operational information specified together with the constraints representing the problem.

For computational reasons, we distinguish for each constraint structure between basic constraints and nonbasic constraints. For basic constraints we require the existence of efficient and incremental methods for checking satisfiability and entailment. We also require that basic constraints be closed under conjunction and existential quantification.

A set of constraints is solved by a process that transfers information from nonbasic constraints to basic constraints. To find one solution, the process will exhibit one satisfiable basic constraint that entails the initial constraint set. To find all solutions, the process will exhibit finitely many satisfiable basic constraints whose disjunction is equivalent to the initial constraint set. Obviously, the process transforms implicit information stated by nonbasic constraints into explicit information stated by basic constraints.

Nonbasic constraints are implemented as propagators. A propagator implements a function that maps basic constraints to one of four possible answers: succeeded, failed, no propagation possible, and advance to a basic constraint D. More precisely, the answer a propagator for a nonbasic constraint C yields for a satisfiable basic constraint B must satisfy the following conditions:

(1) If the answer is "succeeded", then B must entail C.
(2) If the answer is "failed", then $B \wedge C$ must be unsatisfiable.
(3) If the answer is "advance to D", then D must be a basic constraint that is entailed by $B \wedge C$, that entails B, and that is not entailed by B.
(4) If B determines unique values for all free variables of C, then the answer must be either "succeeded" or "failed".

The basic computational setup employed by the method for constraint-based problem solving is a space consisting of a number of propagators connected to a constraint store:

```
Propagator   . . .   Propagator
          \             /
        Constraint Store
```

The constraint store is a satisfiable basic constraint. Computation advances by applying propagators to the constraint store that do not yield the answer "no propagation possible". If the answer of a propagator is

(1) "succeeded", the propagator is deleted.
(2) "failed", then computation stops since the set of constraints corresponding to the space is unsatisfiable.
(3) "advance to D", the constraint store is updated to D.

Certain assumptions on the propagators ensure that the constraint propagation process terminates with a constraint store that is unique up to logic equivalence. In practice, application of a propagator typically has polynomial-time complexity.

If the constraint-propagation process terminates with a failed propagator or with no propagator left, the solution method has done its job. Otherwise, a distribution step is necessary. A distribution step corresponds to an exhaustive case analysis.

For a distribution step, we choose a constraint C and proceed from the unsolved space to two new spaces, the first obtained by adding a propagator for C and the second obtained by adding a propagator for $\neg C$. If the constraint C is well chosen, the two new spaces can proceed by constraint propagation. By iterating constraint propagation with constraint distribution, we obtain a search tree. If we are searching for just one solution and a solution in fact exists, there is no need to explore the full search tree. Rather, we are interested in heuristics telling us which alternative to explore next.

The art of constraint programming consists in using the method in a way that leads to small search trees or explores only few nodes of the search tree in case only one solution is needed. There are three basic techniques for avoiding combinatorial explosion:

(1) Have as much constraint propagation as possible. This means choosing the right constraint representation of the problem, choosing the right propagators for the nonbasic constraints, and finding redundant constraints for which additional propagators will increase propagation.

(2) Find a smart distribution strategy for choosing the constraints needed for distribution steps. The size of search trees depends very much on the distribution strategy employed. A smart distribution strategy will carefully analyze the constraint store and cook up a constraint expressing a decision as strong as possible.

(3) If only one solution is needed, find good heuristics to decide which open node of the search tree should be explored next.

3. WHY PROGRAMMING IS NEEDED

We now discuss the most important reasons why the solution method outlined above should be embedded into a programming language. For each reason we mention the particular features the programming language should have.

3.1 Compilers

Given the data specifying an instance of a problem, one needs a compiler that generates the initial propagators and constraint store representing the given instance of the problem. Obtaining the initial constraint representation from the data is often a complex process that requires nontrivial symbolic programming. The compiler in fact implements a model for the problem to be solved. Ideally, one would like to write the compiler in a higher-order language with rich abstraction facilities. For this a language like Scheme is better suited than Prolog.

3.2 Distributors

One needs a program called distributor to implement the distribution strategy. The distributor is invoked when a distribution step is necessary. It analyzes the current constraint store and then generates the constraint to distribute with. For some problem (e.g., scheduling), nontrivial distributors are required. The fact that distributors need first-class access to the current state of the constraint store conflicts with the philosophy of logic programming.

3.3 Propagators

New applications sometimes require new propagators implementing certain constraints with nontrivial algorithms. Sometimes these algorithms come from operations research. For efficiency, one would like to implement these algorithms in a hardware-oriented language such as C. A high-level constraint programming system should provide a simple interface that makes it possible to import new propagators.

3.4 Inference Engines

Different applications require different inference engines. One needs inference engines that search for one, for all, or for a best solution. It may be necessary to start with one solution and obtain further solutions when needed. It may also be necessary to cancel a search process if it does not deliver in time. Moreover, it may be desirable to explore different subtrees of the search tree in parallel, for instance, by delegating their exploration to different computers on a local area network. For some applications (e.g., constraint-based chart parsing or type inference), specialized inference engines rather different from the ones outlined here are needed. This means that a constraint programming language should not come with a few built-in inference engines but rather should provide the right primitives to program inference engines at a high level. This is again in conflict with the philosophy of logic programming, since primitives for programming inference engines are inherently operational.

3.5 Need for Experimentation

Developing constraint-based problem solvers requires a lot of experimentation. For this reason one would like to have a high-level and interactive programming system providing for rapid prototyping. To understand a problem and a solver, tools are required that visualize the search tree and provide information about the constraint propagation obtained.

4. A FEW WORDS ABOUT OZ

Oz was designed to meet the requirements mentioned above. It combines logic variables and constraints with first-class procedures and rich possibilities for constructing programming abstractions. Rather than providing some built-in inference engines, it comes with a few primitives (e.g., first-class spaces) from which a rich class of inference engines can be constructed. Standard inference engines are of course predefined. The DFKI Oz System provides a C++ interface through which new propagators can be imported. All predefined propagators are provided through this interface. Oz makes it straightforward to organize a program into concurrent agents, some of them performing constraint-based reasoning using encapsulated inference engines. See `http://ps-www.dfki.uni-sb.de/oz/` for more information about Oz.