CP as a general-purpose constraint-language

Vijay A. Saraswat
Computer Science Department
Carnegie-Mellon University
Pittsburgh Pa 15213

Carnegie Group Inc
Station Square
Pittsburgh Pa 15209

Abstract

In this paper we present the notion of *concurrent, controllable constraint systems*. We argue that purely declarative search formalisms, whether they are based on dependency-directed backtracking (as in Steele [Steele, 1980] or Bruynooghe [Bruynooghe and Pereira, 1985]) or bottom-up breadth-first (albeit incremental) definite clause theorem provers (as in deKleer's ATMS approach [deKleer, 1986]) or built-in general purpose heuristics (as in Laurier's work [Lauriere, 1978]) are unlikely to be efficient enough to serve as the basis of a *general purpose* programming formalism which supports the notion of constraint-based computation. To that end we propose the programming language **CP**[↓, |, &], based on the concurrent interpretation of definite clauses, which allows the user to express domain-specific heuristics and *control* the forward search process based on eager propogation of constraints and early detection of determinacy and contradiction. This control follows naturally from the alternate metaphor of viewing constraints as processes that communicate by exchanging messages. The language, in addition, allows for the dynamic generation and hierarchical specification of constraints, for concurrent exploration of alternate solutions, for pruning and merging sub-spaces and for expressing preferences over which portions of the search space to explore next.

I. Introduction

> ... *a constraint is a* declarative statement of relationship... *(and) a* computational device for enforcing the relationship... *[From Steele [Steele, 1980]]*[1]

In this paper we examine the concurrent logic programing (CLP) language **CP**[2] which *strongly supports* the notion of *concurrent, controllable constraint-based* programming in the sense of [Steele, 1980]. Steele presented the programming model of constraint-based computation, and raised the possibility that some day a general purpose programming language may be constructed based on such a model. He noted that *'The constraint model of computation is not supported by any programming language in existence today; the closest approximation is probably* **Prolog.**' We show that a *concurrent* interpretation of definite clause programs provides a computational paradigm that strongly supports constraint-based computation, and, in fact, naturally extends it to the notion of *controllable* constraint-based systems.

[1] All quotations in this paper are from [Steele, 1980], unless otherwise noted.

[2] More specifically, the language discussed in this paper is CP[↓, |, &], which is one of a family of languages, all based on the concurrent interpretation of definite clauses. See e.g. [Saraswat, 1987c] for more details.

II. An introduction to CP

> *'The difficulty with general theorem provers is the combinatorial explosion which resuls from simply trying to deduce all possible consequences from a set of statements. There must be some means of limiting this explosion in a useful way. The challenge is to invent a limiting technique powerful enough to contain the explosion, permissive enough to allow deductions of useful results in most cases of interest, and simple enough that the programmer can understand the consequences of the limiting mechanism.'*

This quote captures our attempt to design control structures to allow the programmer to control (limit) potential combinatorial explosions. To the list above, we only wish to add that, in some appropriate sense, the control structures should be *sound*. The soundness of the control structures we present with respect to the logical interpretation of the clauses underlying the program has been proven in the author's thesis ([Saraswat, forthcoming]), in which the language is studied more extensively (this paper is an abstract of a chapter in it). A formal operational semantics may be found in [Saraswat, 1987c].

We recapitulate the essentials here.

A. Syntax

We take as primitive the usual logic programming notions of *variables*, *functions*, *predicates* and *atomic formulas* ('atoms').

For the purposes of this paper, a **CP** program consists of a sequence of *clauses*. A clause consists of a *head* and a *body* separated (in sequence) by the 'if' symbol ('←'), a *guard* and a *commitment operation* (which is one of '|', the *don't care commit* or '&', the *don't know commit*). The *guard* is a conjunction of atoms for *built-in* predicates. For syntactic convenience, the guard and the commitment operation may be omitted: they default to the special goal `true` and '|' respectively.

The *body* is a *goal system* g, which is either an atom (notated a), a simple goal system (g_1.g_2), an isolated atom ([a]) or an isolated goal system ([g_1.g_2]). A *query* is syntactically the same as the body of a clause.

The *head* of a clause is a *well-annotated* atom in a language whose non-logical symbols contain a special unary function-symbol '↓'. If t is a term in the language, we say that t↓ is an *annotated term* (if t is a functional term, then its function symbol must be different from '↓'.) A term is *well-annotated* if either it does not contain an annotated subterm or every super-term of an annotated term is annotated. For simplicity of syntax, we *assume* that every super-term of an annotated term is annotated: hence only the innermost annotations need be explicitly written.

B. Informal semantics

Very roughly, the operational semantics of CP programs is the same as the operational semantics of pure Horn clauses (using SLD-refutation) except that in every step of the refutation process, the **mgu**$_\downarrow$ of two atoms is used, instead of the **mgu** (most general unifier), and it must be possible to satisfy the (built-in) goals if any, in the guard.

The **mgu**$_\downarrow$ of two terms, when it exists, is the same as the **mgu** of the two terms. To compute the **mgu**$_\downarrow$ of two terms, one follows the same algorithm as for computing the **mgu** except that an annotated term t↓ can unify against another term t_1 *only if* the term t_1 is not a variable. In case t_1 is a variable, unification is said to *suspend* until such time as some event causes t_1 to be instantiated. Decorating a term in the head of a clause with a '↓', then ensures that the clause can be used only for goals in which the argument in the corresponding place is a *non-variable*.

As an example consider the clauses, which may be taken to define a 'plus' constraint:

```
X↓ + Y↓ = Z   ← Z is X+Y.
X↓ + Y = Z↓   ← Y is Z-X.
X + Y↓ = Z↓   ← Z is Y-X.
```

For a given goal A + B = C, these clauses are applicable only if at least two of the variables A, B, C are instantiated.

The only 'built-in' predicates we consider in this paper are is/2 and ==/2. A goal A is B) succeeds if B can be 'evaluated' as an arithmetic expression; the value is unified with A. The goal A == B suspends until either A is unified with B or A and B are (top-level) instantiated, whence it succeeds if A and B can be unified and fails otherwise. Consider the following clause:

```
X * Y = Z↓  ← X == Y | X is sqrt(Z).
```

A goal A * B = C succeeds (with this clause) only if A and B are either the same variable or are instantiated, and C is instantiated; as a result of the execution of this clause X (*and* Y) are unified with sqrt(Z).

We now consider the execution cycle in more detail. Computation is initiated by the presentation of a query ← $a_0 \cdots a_{n-1}$. Each goal a_i will try to find a proof by '↓'-unifying against the head of a clause, and finding proofs for first, the goals in the guard and, after committing, for the goals in the body of that clause. Unification results in *bindings* for the variables in the goal, which are *communicated* (applied) to sibling goals at *commit-time*. A goal can simultaneously attempt to unify against the heads of all the clauses; however different clause invocations for a goal must commit in some serializable order.

Commitment involves three operations: the *atomic publication* of the answer bindings, action on other OR-siblings and promotion of body goals.

Atomic publication of bindings means that the bindings are (conceptually) instantaneously applied to all the goals in the body of the clause and to *all* AND-sibling goals of the committing goal a_i. The *extent* of this publication is determined by goal system boundaries: the bindings are published only upto the smallest enclosing goal system boundary.

Together with atomic publication, both the commit operations also cause the goals in the body goal-system to be executed as AND-siblings of the goals that were the siblings of the committing goal (i.e. their uncles).

The commit operators differ in the actions they take with respect to other OR-siblings of the committing guard system, either in the same clause or in other clauses. The don't care commit *kills* them all. The don't know commit allows OR-siblings to keep on computing, in effect allowing a goal to commit *multiple* bindings. Each of these bindings is committed to a *different* copy of the rest of the goals in the smallest enclosing GoalSystem. Thus, if a goal '&'-commits bindings θ, all the sibling goals in the smallest enclosing goal system are split into two at commit time. Goal system boundaries are *one-way walls*: they allow bindings committed by a sibling of the goal system to enter the block, but prevent the bindings committed from within the goal system from leaving.

As computation proceeds, the goal system within a block may thus repeatedly split. Two adjacent blocks (isolated atom or isolated goal systems) are combined when there is no more progress to be made in either one, i.e. either each has terminated successfully or suspended for lack of input. The process of merging two blocks b_1 and b_2 is concerned with creating another block b which has one OR-branch for every *compatible pair* of OR-branches, one each from b_1 and b_2. An OR-branch contains a sequence of suspended blocks and goals, together with (possibly vacuous) substitution, which is the composition of all the substitutions that have been committed internal to the goal system. Two OR-branches are compatible if their substitutions are compatible. Informally, two substitutions are compatible if they do not assign ununifiable values to the same variable. The substitution associated with an OR-branch of b is obtained from these two substitutions, and the sequence of blocks obtained by concatenating the two sequences. Conceptually, all the branches of one block may be merged with all the branches of another block in parallel.

Finally, computation succeeds when a branch finds a so-

lution (no more goals left to prove); it fails when all branches terminate in failure.

C. CP as a constraint-based language

The use of CP as a constraint language should now be clear: goals in the current resolvent correspond to *constraints* and the program axioms correspond to the rules of behaviour for a constraint.

The versatility of CP as a constraint language arises from its simple solution for the *control problem.* The control problem in the context of constraint-based languages is: given an under-constrained system, which of a possible set of *assumptions* to make next? Another useful metaphor for programming in CP is to think of a goal as representing a *process*, processes communicating with each other by instantiating shared variables to *structures* which may contain other terms, called *messages.*[3] (Recall that variables may be instantiated not just to constants — actually, just integers in Steele's language — but also to arbitrary terms.) This allows the user to solve the control problem by programming *negotiations* between various constraints. For example, in the case of a discrete constraint satisfaction problem, it is possible for the user to write constraints such that if local propagation does not yield a solution then the constraints cooperate to determine the problem-variable which is the most constrained (the so-called 'fail-first' heuristic) and have the constraint corresponding to this variable *make assumptions* about the possible values for the variable. (We discuss a specific example in the next section.) The elegance of the CP solution to the control problem lies in that *such heuristic rules are expressed in the same language, using the same concepts and techniques as the constraint propagation rules.*

The process metaphor provides another important benefit: in CLP languages, process behaviours are naturally specified in a recursive form. Such rules can describe in a succinct fashion arbitrary recursively constructed topologies and inter-connection patterns for constraints. For example, *one* constraint system may be specified for solving the N-queens problem, where N is an input: this system uses the value of N to spawn a network of appropriate size. Moreover constraint definitions (and not just connection structures) may recursively depend upon each other.

We now consider the design criteria Steele lays down for a constraint language.

> *Design Goal 1: As far as possible, the computational state of a constraint system should depend only upon the relationships stated so far, and not on the order in which they were stated.*

In our language constraints are represented by means of goals in the current resolvent. All these goals are treated as AND-parallel siblings: hence any one goal can reduce at any given time, provided that it can find a matching behaviour.

[3]The versatility of CP as a concurrent programming language is demonstrated in e.g. [Saraswat, forthcoming], and [Saraswat, 1987a]. An overview of programming techniques in the related CLP language Concurrent Prolog may be found in [Shapiro, 1986].

> *Design Goal 2: As far as possible, a constraint-based system shall perform its computations on the basis of locally available information only.*

In our language the only information that can influence the behaviour of a constraint is information contained directly in the constraint in the form of variables, and their current bindings. Hence this design goal is trivially satisfied.

> *Design Goal 3: A constraint-based system should, so far as possible, be* monotonic. *The more is known, the more can be deduced, and once a value has been deduced, knowing more true things ought not to capriciously invalidate it.*

The language CP is monotonic in a very important sense: as computation progresses, only bindings that are consistent with the ones already generated are produced. Moreover, if a constraint is known to be true, then providing *more information* (in the form of bindings) *cannot* invalidate the constraint.

Steeele goes on to discuss the use of the term 'capriciously'. He wants to be able to allow the system to make *assumptions* which may later be retracted in the light of more information in a reasoned way.

In CP, *assumptions* are made when a goal reduces using a rule with the don't know commitment operation: the bindings associated with this resolution step constitute the assumptions made in this inference. In a sequential language such as **Prolog**, such bindings may be undone on backtracking: in CP, such bindings are always assumed to be made to a *copy* of the current resolvent. Hence taking a tentative step in CP always corresponds to *splitting* the current query into two disjoint queries. If future processing results in a contradiction being discovered, the current copy is merely discarded; meanwhile the other copy is free to make other derivations, and thus pursue other contexts.

The presence of the other control structures may also be motivated naturally. The '$\downarrow$'-annotation is essential: without some such annotation on unification it is impossible to specify (efficiently) that a highly non-deterministic constraint should suspend until more bindings are available which reduce the number of possible solutions for the constraint. The don't-care commit is necessary to allow the user to specify that alternate solutions to the constraint are to be eschewed; thus pruning portions of the search-space. Both the '$\downarrow$'-annotation and the '|' commit introduce incompleteness.

Finally, *blocks* allow the user to provide control information which may be quite important in solving *loosely connected* constraint systems efficiently. There are two important computational savings that blocks may introduce. First, in a system such as `[b.[g1].[g2]]`, any determinate bindings introduced by `b` are shared by all the sub-contexts in `g1` and `g2`. (This is quite analgous to pushing sub-contexts in Conniver-style languages in which changes in the original context are visible in the sub-context as well.) The advantage here is that whenever `g1` splits into two, `b` is not copied into both the sub-contexts, resulting in `b` making the same transition twice. The price paid is that no bindings that `g1` produces can be communicated to `b` until merge-time. Second, in a system in which

g1 and g2 spawn a large number (say c_1 and c_2 respectively) of alternate branches only a few of which survive at the end (say $b_1 \leq c_1$ and $b_2 \leq c_2$), the number of contexts examined are $c_1 + c_2 + b_1 \times b_2$ rather than $c_1 + c_2 + c_1 \times c_2$. If there are few interactions between two large constraint systems, it is preferrable therefore to solve the constraint-systems in isolation and then combine the results. (See [Saraswat, 1987c] for a discussion of an example used in [deKleer, 1986] to illustrate pathological behaviour by chronological backtracking systems.)

To sum up, our language design exhibits the following charactersitics: it allows the user to express control over the constraint-propagation as well as the constraint-selection phase using naturally motivated concurrent programming idioms, it allows a natural notion of user-definable, hierarchical, mutually-recursive constraints, and provides a problem-solving framework in which multiple solutions are possible, together with the possibility of simultaneously working in more than one context.

III. Examples: N-queens

In the following we consider a solution to the *N*-queens example. We first consider a purely declarative program (with no search control) and then consider how to improve its performance by programming various heuristics.

A. A straightforward solution

We consider a solution (first presented and discussed in [Saraswat, 1987c]) in which there is a constraint for every square on the chess-board. We imagine that in order to solve the *N*-queens problem, we have spawned an $N \times N$ chess-board with one `cell` constraint for every square on the board. Each constraint has six parameters: its `I` and `J` coordinates, and four *wires* (variables), the `H, V, L R`. All the cells on the same row have the same `H` wire, on the same column the same `V` wire, on the same left-diagonal the same `L` wire and on the same right-diagonal the same `R` wire. (Each wire could thus have a fan-in/fan-out of up to *N*.) There are just two behaviours for every cell. Each cell may either non-deterministically decide that it has a queen (in which case it sends its Id on all the four wires incident on it and terminates) or else it waits for some cell on the horizontal wire to declare that it has a queen and then it terminates. Note that as soon as a cell decides that it has a queen, no other cell that is dominated by it can decide that it has a queen (no two cells have the same ID). It should be clear that this solution is correct and complete: exactly the set of solutions to the *N*-queens problem may be obtained by following these behaviours. The specification for a `cell` is simply:

```
cell(I,J,J,I,I,I) ← true & true.
cell(I,J,H↓,V,L,R) ← true & true.
```

B. Doing local propagation before choosing

While the program given above is correct, it may not exhibit good run-time behaviour, because of two reasons. First, there is no guarantee that when a cell asserts that it has a queen, all other cells which have are dominated die immediately. If these cells remain they may be unneccesarily copied each time a new assumption is made. Second, it is preferable to detect as soon as possible when all the cells on a row or column have been dominated by queens already placed (and there is no queen on that row or column), because such a state is bound to lead to failure. Along the same lines, if a row or column has just one non-dominated cell left, then it is preferrable if that cell immediately decides that it has a queen, because given the problem formulation, it must have one for a solution to exist. In a phrase, local propagation should precede making assumptions.

We obtain this effect as follows. We assume a mechanism (discussed in the next section) for serialising *phases*. There will be *N* phases; in each phase, one queen is placed, and the next phase is not initiated until the previous phase quiesces. A phase is initiated when a cell (the *leader* for this phase) decides that it has a queen and is considered to terminate when the leader detects quiescence. We now consider a topology in which each cell, besides having its `I` and `J` coordinates and the four wires, is also connected in four *rings*, one each along the horizontal, vertical, left-diagonal and right-diagonal axes. For each process, its ring-connections consist merely of two wires, one connecting it to its predecessor (the `left` connection) in the ring and the other connecting it to its successor (the `right` connection). (To be precise, the `left` connection of a cell is the same variable as the `right` connection of the cell to its left along the given axis; similarly for the other direction.)

As before when a cell decides it has a queen, it sends its Id on the `H, V, L` and `R` wires. We would now like to force the cells that get dominated to die in the current phase. We can achieve this by using a variation of the so-called *short-circuit* technique for detecting distributed termination ([Saraswat, 1987a]).

The idea is simple. When a cell is dominated, it should die; this implies that it should remove itself from all its rings. It can remove itself from a ring by *shorting* its `left` and `right` connections on that ring: by shorting two variables, we mean unifying them. After it does this, its right neighbour will become the right neighbour of its left neighbour, and vice versa. (This is analogous to removing an item from a linked list.) However, when a cell decides it has a queen *all* the cells remaining on all its rings will remove themselves. After this occurs, the leader will find that, for each ring, its `left` and `right` connections are the same; it thus detects that the current phase has terminated.

We give a sample rule to show how straightforward this is to implement in **CP**. We assume that each cell is of the form:

```
cell(id(I,J), wire(H, V, L, R),
     rings(Hleft-Hright, Vleft-Vright),
     rings(Lleft-Lright, Rleft-Rright))
```

where the variable names should be self-explanatory. (Note:

Note that as in the previous section, the leader process can detect that the current phase has quiesced exactly when all its four rings are shorted. With this new protocol, however, it is possible that the four rings are *never shorted*: this happens exactly when, as a result of placing this queen, some row or column which does not yet have a queen has no more cells left. This results in the current context being *deadlocked* and consequently abandoned by the problem-solver.

2. Early detection of determinacy

We leave as an exercise for the reader the problem of programming a protocol such that if in the current phase a cell is detected as being the only one in a row or column, that cell is forced to have a queen (in the current phase).

C. Choosing the next queen wisely.

In the previous sections, the decision of *which* cell next decides to have a queen is still non-deterministic: *any* cell that is not yet dominated may so decide. We now sketch how the same techniques may be used to implement heuristics for making this control decision.

Recall that the problem was formulated by having a `cell` constraint for every square on the chess-board. We now add an extra constraint `enable`, one for each row in the chess board. Each `enable` constraint is linked into the horizontal ring for that row, and all the `enable` constraints are connected together in another ring. Conceptually, a *token* will flow down the links of this *privilege* ring, which will ensure mutual exclusion, i.e. sequentialisation of each phase. A cell can decide that it has a queen only if the `enable` process on its horizontal ring has the token. When this cell detects the end of its quiescence phase, the token is passed on to the next `enable` contraint in the privilege ring. This simple protocol results in the queens being placed in row-major order.

Consider now the implementation of a heuristic (which is quite useful for the N-queens problem) that in each phase only a cell with the highest weight can decide to place a queen: the weight of a cell is the number of other cells this cell can dominate if it had a queen. It is straightforward to associate with each ring a count of the number of elements in the ring, to have each cell compute its weight from the counts of the rings incident on it and to add to the cells a network of `max` devices which select in each phase the cell with the highest weight. This cell then becomes the leader for the next phase, and continues the cycle of waiting for local propagation to terminate (achieved by means of the `end` goal) enabling the selection phase for determining the next leader, and passing control to it.

IV. Comparison with other work

We very briefly consider other related work. More details may be found in [Saraswat, forthcoming].

CP differs from other CLP languages such as **GHC** and **Parlog** in using atomic commitment, together with unification (as opposed to matching) during process execution. This ability seems fundamental to obtain the dynamic dataflow that characterises constraint-based computation. **Concurrent Prolog** is also based on unification, but it introduces a problematic capability annotation which does not seem to be directly relevant to modelling constraints. An alternative viewpoint related to constraints may be found in [Lassez and Jaffar, 1987]. The techniques in [van Hentenryck and Dincbas, 1986] seem to be easily representable, and are naturally generalised, in our framework. **CP** avoids the combination of chronological backtracking and pre-determined order for instantiating problem variables that plague the use of **Prolog** as a language for constraint-based computation ([deKleer, 1986]). By making sure that the opportunity is available to propagate all the consequences of a choice to all the constraints *before* making the next choice, we ensure that it is possible to write programs such that when a contradiction is discovered, the last choice made contributes to the contradiction.

Problem solvers based on reason-maintenance systems have recently been studied (e.g. [deKleer, 1986], [McDermott, 1983]). In such systems, as computation progresses, the problem solver informs the RMS of the assumptions it makes and of the *justifications* that it discovers. The (well-known) problem here is that may be quite difficult for the problem-solver to determine which dependencies to capture in its justification of an inference and also quite difficult for the problem-solver to exercise control.

V. Acknowledgement

I am grateful to Jon Doyle for extensive discussions, to Guy Steele and to many others at CMU and CGI (particularly Gary Kahn, Dave Hornig and Mark Fox) who have discussed this work with me.

References

[Bruynooghe and Pereira, 1985] M. Bruynooghe and L.M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *Implementations of* **Prolog**, Ellis Horwood, 1985.

[deKleer, 1986] J. deKleer. An assumption based TMS. *Artifical Intelligence*, 28:127–162, 1986.

[Lassez and Jaffar, 1987] J-.L. Lassez and J. Jaffar. Constraint logic programming. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM, January 1987.

[Lauriere, 1978] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *AI*, 10:29–127, 1978.

[McDermott, 1983] D. McDermott. Contexts and data-dependencies: a synthesis. *IEEE Trans. on Pattern-directed Inference and Machine Intelligence*, 5(3), 1983.

[Saraswat, 1987a] V.A. Saraswat. Detecting distributed termination efficiently: the short-circuit technique in **FCP**(↓ , |). February 1987. To be submitted.

[Saraswat, 1987c] V.A. Saraswat. The concurrent logic programming language **CP**: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM, January 1987.

[Saraswat, forthcoming] V.A. Saraswat. *Concurrent Logic Programming Languages*. PhD thesis, Carnegie-Mellon University, forthcoming.

[Shapiro, 1986] E.Y. Shapiro. **Concurrent Prolog**: a progress report. *IEEE Computer*, pp:44–58, August 1986.

[Steele, 1980] G.L. Steele. *The definition and implementation of a computer programming language based on Constraints*. PhD thesis, M.I.T, 1980.

[van Hentenryck and Dincbas, 1986] P. van Hentenryck and M. Dincbas. Domains in logic programming. In *Proceedings of the AAAI*, pages 759–765, 1986.