PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS 1988, edited by ICOT. © ICOT, 1988

MULTIPLE REASONING STYLES IN LOGIC PROGRAMMING

Hervé Gallaire

European Computer-Industry Research Centre GmbH (ECRC)
Arabellastr. 17, D-8000 Muenchen 81
F.R.G.

ABSTRACT

Logic programming still needs to penetrate the industrial world. This paper takes the perspective that if logic programming was perceived as a tool supporting multiple reasoning styles, this process would speed up The paper discusses several such styles and the extensions through which logic programming must go to support them more completely. Such an extension is that of constraint programming. The paper argues, using this example, in favor of a very tight integration of the mechanisms that support extensions. Adopting such a view requires accepting to modify sometimes deeply the kernel of logic languages implementations, and the paper reviews why this is so. The paper finally analyses, using the same example, whether this quest for extensions has any limit.

1 INTRODUCTION

Why are there still sessions devoted to Applications in Artificial Intelligence Conferences, special issues of AI journals dealing with Applications and the like?

The same questions apply to Logic Programming (LP). The reason for this situation is obvious: while the LP community is very much active, LP is only making its first steps in the industrial world, even though there is growing evidence of its success in many varied tasks. Consequently, successful applications must still be discussed. In (Gallaire, 1987), I have partially analysed this situation and argued that applications needs were not fully addressed by the classical LP languages (e.g. Prolog), that a few new features were to be added to LP to support large classes of applications, and that these features, as well as some known ones, needed to be supported by an adequate software architecture. Recent progress reviewed encompasses:

- Advances in the theoretical understanding of logic programs, which pave the way for many important improvements (partial evaluation, program transformations, code optimisation, extension of tractable language constructs such as negation); these are the easiest to provide once the theory is understood, but their applicability to real programs which include non logical features is still difficult - a fact that leads some to propose a complete overhaul of the existing LP languages. Further results obtained here could constitute an alternative to the proposals made in the paper.
- Advances in the area of knowledge representation, particularly with respect to the use of type manipulation facilities and of object-oriented programming techniques; most of these have been provided at a high level, without efficiency concerns, and their impact has been weak.
- Advances in the area of problem-solving techniques through extending LP with constraints handling techniques as well as specialised unification algorithms (e.g. boolean unification), providing in effect a new type of LP systems which are seen to be increasingly useful.
- Advances in the use of large databases in conjunction with LP; the best way of doing such connections is being still a matter of debate, but their need is increasingly felt. Loose connections are proliferating. The question of what inference engine to use for a tight integration is a delicate one. There are three contenders starting from the LP perspective: a DB-like engine (bottom-up, extended relational operators), a Prolog-like engine (top-down, tuple at a time, suitably adapted to handle recursion, buffers, types, etc) and an LP-based engine (top-down, incorporating set operators).

Advances not reviewed in (Gallaire, 1987), which will not be discussed here either, but which are also key for large classes of applications are those linked to parallel LP languages, both the concurrent logic languages family and the parallel logic languages one. A software architecture has been proposed for developing systems supporting effectively these advances in an incremental manner: the "glass box" architecture of which an implementation SEPIA (Meier, 1988) is pending completion, showing the feasibility of this concept and its interest.

In this paper I would like to concentrate on analysing the suitability of LP for applications of a problem-solving nature

and discuss issues such as:

- Which features do we need to support multiple reasoning styles in LP?
- How far should the language designers support these features?

In order to answer these questions, the first section of this paper analyses what reasoning styles ought to be supported and at the same time investigates how current LP systems support them, taking advantage of their most advanced features.

Section 2 reflects on existing developments to make further proposals and discusses their limits. The driving force behind this analysis is that providing the user with the opportunity to use certain mechanisms is not sufficient. Efficiency is the key in applications.

Logic Programming seems to be moving - or should we say must move - in a completely new world of applications. Logic is domain independent, but LP must be adapted to take advantage of computation domains. Logic is isolated from mainstream sciences, particularly domain specific mathematics and consequently from applications which rely on them heavily; LP must start merging with them. Logic systems carry almost blind searches, but LP must allow proof searches to be more focused. The list of such changes could go on, but the idea should be clear by now: LP must become more knowledgeable, if it is to expand its domain of applications. A major drawback might however be facing us, namely that knowledgeable systems become specific: a company, a scientist is only good in some limited areas. How far should this go, and in which directions? These are some of the questions that this paper attempts to answer. In the sequel, when LP is mentioned, the conventional LP systems relying on an execution model similar to that of Prolog are meant.

2 REASONING STYLES

Reasoning is the activity that supports problem-solving. In the scope of this paper we need to single out some of the reasoning techniques which have been well studied in AI such as:

- reasoning from first principles
- reasoning on types, in particular taxonomic reasoning
- hypothetical reasoning and truth-maintenance systems
- goal-directed reasoning (backward reasoning)
- mixed-mode reasoning
- incremental reasoning

They will not be described in detail here. Rather they will be revisited from the point of view of logic programming.

2.1 Reasoning from First Principles - RFP

RFP consists in modeling and axiomatising, i.e. making explicit, the principles which govern the system being studied (Davis, 1984, Poole, 1986, Reiter, 1987, De Kleer, 1987). It has been defined for diagnosis and knowledge based systems. Queries about these systems are then answered by evaluating them against the models. There is a parallel between RFP and the trend that can be seen in software engineering today. In that world, something has changed: no one really cares about programming languages; today people are much more interested in performing (well) the modeling phase and the detailed specification phase. This is also addressed by RFP, but RFP requires as well to be able to compute something, thus extending the software engineering approach. Modeling and specifying are believed to be most influential on the success or failure of projects, although I believe the language choice to be extremely important. LP is one of the very few formalisms that can cover from one end of the spectrum to the other end, ie from modeling in the sense of RFP, or in the more classical software engineering meaning, to computing. If it proves to be useful for RFP, LP should be useful for software engineering, from modeling and specifying to programming. But while the RFP approach is not concerned with the query process efficiency, on the contrary LP should permit to develop efficient programs in order to fulfil this ambitious goal.

Logic is the ideal support for RFP and there is little doubt of the benefits of using logic for this purpose. Functional programming is also a contender (Bosco, 1988), but it does not match the versatility of LP. However RFP is not without problems: in particular, the following two questions need be answered:

- How much do we need of the first order logic and of other logics in order to support RFP?
- Given a modeling problem, how much of the real world must be made explicit and how much can be embedded in the system which runs the (thus partial) model and the queries?

For example, instances of the first question could be: how much of the logical negation do we need, or how much of a temporal logic do we need? An instance of the second question could be: is it needed to make explicit the assumption that only one component of a circuit at a time can be faulty, or can this be embedded in the fault-finding algorithm, or even in the combination of the algorithm and of the interpreter of the

language that was used to write the fault finding algorithm?

A straightforward axiomatisation of a problem domain and of its associated reasoning principles, which does not use features of the underlying mechanisms to replace large sets of plain axioms, does not ususally provide the efficiency needed to effectively reason from first principles. The need for negation as failure (Clark, 1978) to replace very many axioms (the particularisation axioms) is well known, and it is also known for the problems that it introduces. It is not known how to do much better. In case the axiomatisation applies only to a limited domain (e.g. axiomatising the unique fault assumption, as opposed to the particularisation axioms) the above question must be replaced by the following: are there sufficient commonalities between domains that could be abstracted and embedded in the evaluation mechanism so as to take into account RFP in more efficient ways? Is this a fatal flaw which takes away from logic?

Another way to consider the same questions may help finding answers. LP, which only "knows" about the Herbrand universe, is particularly suited to RFP because it does not embed knowledge about specific domains. If there is a (practical) need to handle implicitely some of the knowledge that an RFP approach to a problem requires, and to use "hidden assumptions", a way to counterbalance this generally criticised practice is to increase the power of expression of the language. This means allowing to state at a high enough level of abstraction the knowledge corresponding to the problem domain. An obvious case in relation to problem solving is to offer the opportunity to use "natural" concepts of the domains; constraints, such as equalities, inequalities, and more generally relations, etc are examples of such concepts (Montanari, 1974). Correspondingly, the power of the language interpretation or execution mechanisms must be extended to work at that level of abstraction, and to avoid inefficient simulations of these concepts. This interpretation of RFP proves to be very powerful. One does not do arithmetic from a full-fledged RFP perspective; instead one uses whatever knowledge is already available. These extensions may allow or involve real changes to the computational model of LP.

Can such an LP system extended for modeling purposes, better adapted to RFP and usable in software engineering, be efficient for a class of problems? It will be, provided the domains for which these extensions are made, have been carefully selected. The changes to be made to the computational model can be made at two levels: they can be achieved by extending and modifying the unification mechanisms and linking unification to a limited form of control of the search tree expansion; alternatively they can be achieved by adding further constructs to fully control the search tree expansion. The first way is the one adopted for example in specialised unification algorithms such as typed unification and boolean unification (Buettner, 1987). It is also the one adopted for dealing with delaying techniques (Naish, 1985) and constraints manipulation, where unification interacts with the control of the search tree in non standard ways. The systems which embody these features have a wide applicability; this points to a positive answer to the question of commonality across various domains; the needs of various domains can be abstracted into unique mechanisms. For example, what is common to a scheduling problem and to a diagnostic problem is a set of techniques for propagating or relaxing constraints. Not using such common knowledge or well known mathematical tools prevents logic from connecting to the real world. How much of it to use is still very much debatable, and this issue will be discussed in the last section.

Getting full control of the search tree expansion has received attention (Bourgault, 1984, Gallaire, 1982) but has not come into wide use. Perhaps this is due to the fact that no compiler does much to support literal selection or clause selection. Interpreters which allow this have been either coded in high level languages (eg Lisp (Robinson, 1980, Bourgault, 1984)) and easily modifiable, or are meta-interpreters. Although it is not known how to effectively get rid of the meta-interpreters overhead, recent experiments (Owen, 1988) seem to indicate the feasibility of application specific control and work on partially evaluating away the meta-interpreter is making progress. A full control of the search tree may be useful in a logic-based view of knowledge bases and deductive databases.

Before leaving RFP and the modeling techniques that it requires, we must stress the importance of symbolic modeling as a complement, rather than an alternative to numerical modeling. Going from the era of numerical applications to that of Artificial Intelligence applications has stressed the value of symbolic techniques. One of the strength of LP in applications is this ability to mix symbolic and numeric computing. However, while numerical modeling rests on a firm computational ground, delivering highly specialized algorithms, and thus providing very efficient techniques, in comparison symbolic modeling employs, by and large, very naive techniques: for instance symbolic constraints are usually only used for verification purposes, they are not actively used to produce values. It is important to develop systems where

symbolic techniques are taken more seriously; for example symbolic constraints can be used actively to prune the search space. This requires adequate techniques such as constraint propagation to exist in the underlying interpretation mechanism. It is easy to find problems where purely numeric constraints could be set but where the active use of symbolic constraints in conjunction with a different set of numeric constraints outperforms them by one or more orders of magnitude (Dincbas, 1988a). The reason for this is that numerical techniques have somehow been used in all cases because there was no other computational domain open to machines. But their exclusive use may totally change the size of the problem when they encode symbolic constraints; the case where 0-1 variables must be used to simulate symbolic constraints, ususally increases significantly the computational complexity. Symbolic computation does radically change the picture when used in combination with numeric computation.

2.2 Taxonomic reasoning

Taxonomic reasoning has been identified in Artificial Intelligence and in Theorem-Proving as a major paradigm for modeling and organizing structural knowledge and also for supporting high level inferences and organizing the inference process. Types (or sorts) are used to get small search trees, thereby increasing efficiency (Walther, 1984). Taxonomic reasoning, as a modeling tool, also fits naturally with the reasoning from first principles paradigm.

Taxonomic reasoning's interest stems from the combination of these features. In order to be used effectively, the two aspects must be tightly connected. This is for example the link between the A-Box and the T-Box in Krypton (Brachman, 1985).

It has long been known that logic could support taxonomic reasoning in particular through the use of typed logic or many-sorted logic. However, typed logic is mostly provided in LP through a reformulation of it in standard first order logic (Enderton, 1972). This is simple to do, and indeed a very large number of papers and systems have pushed the idea that such easy-to-implement rational reconstruction of knowledge representation techniques was sufficient.

I do not believe this to be true in a real application world. What is lost in such a simple translation is the ability to reason at the type level, and to develop answers which do not instantiate variables when this is not necessary. What is at stake here is not only the efficiency but also the meaningfulness of answers. An instantiated answer may be much less significant for the user than a more general answer in terms of typed variables. Such an issue is being investigated in the field of knowledge bases as well as in that of LP.

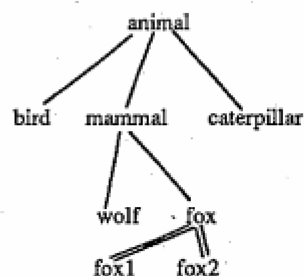
Most systems do such a systematic translation of types in LP, and lose the opportunity of reasoning on types. Among the others very few provide a reasonable handling of types.

What is meant by reasonable? Again it is an integration with the underlying logic system by opposition to developing a meta interpreter for providing the extension (using as much as possible of the underlying logic system by amalgamation (Bowen, 1982)). As noted above, eliminating the inefficiency of the meta interpreter is still very much a research issue.

An example of such an integration is the PHOCUS system (Chan, 1987). In PHOCUS, objects are constants or individuals of the logic domain, classes are types which are represented by objects, methods are polymorphic predicates. An example of reasoning on types is given in example 1.

<u>Example 1.</u>

Given the following hierarchy on animals:



given that fox1 and fox2 are instances, that:
smaller (X%fox, Y%wolf)
smaller (X%bird, Y%wolf)
(and a recursive closure on it) and that:
locomotion (X%mammal, run)
locomotion (X%bird, fly)
locomotion (X%caterpillar, crawl)

the query ?-smaller (X,Y%wolf), locomotion (X, run) would provide as an answer X = X%fox. If an explicit answer is interesting, add generate (X) to get the answer X = *fox1*

It is also possible to have an exception mechanism in this scheme. Furthermore, variables are treated as logical variables, methods are non deterministic and will provide more answers when required (few systems offer this capability as mentioned by (Conery, 1988)). The object part of PHOCUS is for the essentials, implemented in SEPIA (Meier, 1988). A related

approach in LOGIN (Ait Kaci, 1986) generalizes Prolog terms to Ψ-terms that can be regarded as object descriptions: these are type constraints but also structural constraints; Ψ-terms can be refined by unification. Ψ-terms always remain at the type level, they are never instantiated.

Manipulating types appears increasingly useful, for example in natural language and in program analysis. There is also promising work on higher order types (Nadathur, 1988). Nevertheless I believe LP will remain essentially an untyped programming language.

A rather different use of types is for optimisation purposes, such as generating data structures and accessors to them; this is a more conventional programming language view. It is possible that some applications will only become viable on typed subsets of LP; perhaps array manipulations fall in that category (Voda, 1988a). Apart from explicit type declarations as above, several systems do automatic type inference from the programs; the types so inferred may serve both purposes, modeling and optimisation; this is currently a very active field (Xu, 1988).

2.3 Hypothetical reasoning and TMS

Hypothetical reasoning, truth maintenance, multiple worlds reasoning, planning, are useful concepts in application developments. Such styles of reasoning have been considered to be largely beyond the scope of LP. There are reasons to this, in particular the monotonicity of logic. In fact, however, classical LP systems do support some of the usages of hypothetical reasoning. The intelligent backtracking capability of several LP systems correspond to limited TMS and planning features (Bruynooghe, 1984, Codognet, 1988). The use of constraint propagation techniques also alleviates the need for TMS as shown in applications for debugging circuits. It is worth mentioning here that LP equipped with such techniques may outperform TMS systems by an order of magnitude at least (Simonis, 1987). Additionally several LP systems have shown how to incorporate hypothetical reasoning and multiple world reasoning at the meta-interpreter level (Kaufmann, 1986, Chan, 1987); it appears possible to compile such metaprograms with only a limited amount of changes to the basic LP engine, namely the WAM (Bacha, 1988). Systems which are more TMS-like and sophisticated (Poole, 1986, Eshgi, 1988) have been developed. In summary, LP is suitable for this style of reasoning, contrary to accepted positions. Furthermore, limits of applicability of this reasoning style on large scale problems are known, no matter whether it is provided in a LP framework or in a Lisp or a rule-based framework.

2.4 Goal-Directed and Mixed-Mode Reasoning

Goal-directed reasoning is the reasoning style of LP by excellence and there is no need to describe it nor to describe how it is supported in LP. Supporting forward reasoning can be done as usual at several levels. If done at a low level, then this automatically provides mixed-mode reasoning, and this in turn can be used as an enabling mechanism for hypothetical reasoning, etc. The delaying mechanisms available in some LP systems are worth noting in this context (Colmerauer, 1981, Naish, 1985); they are often used to provide data-driven (demons) mechanisms, a feature related to forward reasoning. Typical also of forward and mixed-mode reasoning is again the use of equations and of constraints which "work both ways". Applications taking advantage of this have been successfully developed (Berthier, 1988, Huynh88, 1988) and many more will come. Attempts to directly embed data-driven techniques in LP by adding in the kernel a forward execution mechanism, can also be mentioned here (Chan, 1987). Although direct embedding appears promising, there remains much to do to understand how to control the interactions between the backward LP and the forward mechanism at that level. But the support obtained by demons and constraints is already a very useful one.

2.5 Incremental Reasoning

This is a much less documented reasoning style although it is very clear that people do solve problems in this way. Very few systems support incremental problem-solving: usually after returning to the user, the context of the search is lost; no subproblem which has been solved as part of a global problem can be easily retracted. LP appears to be amenable to adaptations which can support this style of reasoning (VanEmden, 1984, Okhi, 1986, Chatalic, 1987). There is a strong similarity between incremental programming and dependency-directed computation, and TMS. Although technical problems do not seem intractable, implementation problems do require further attention in the future in order to get efficient systems. However this will not be discussed further.

3 SUPPORTING REASONING STYLES?

It must be clear that there is no way of fully supporting all reasoning styles described in the previous section simply with one single extension of classical LP. This is why it was argued in (Gallaire, 1987) that an open software architecture must be provided, in the kernel of which it must be possible to install new features. This involves design choices, causing sometimes slight performance degradation even when the other features are not used. Because the cost of such an approach is non negligible its case must be argued strongly. The reasons depend on the type of extensions being considered, and cannot be given abstractly. In the rest of this paper, an example is chosen and detailed: the constraint programming languages; it was mentioned that they support several reasoning styles at once. The analysis to be presented is based on the work on the constraint programming language CHIP (Dincbas, 1987, Dincbas, 1988b), which is indeed being embedded in the "glass box" SEPIA. The SEPIA compiler system has an open architecture. The core system provides:

- standard Prolog, including modules
- true interrupts for real-time applications
- a delay mechanism whose semantics are simple and clear, and where delaying conditions are programmable in Prolog.

SEPIA is also extensible; this is one of its strengths. The types and objects described earlier are one such extension which already exists.

A compiler for CHIP is currently being developed and is part of the SEPIA environment as well. Up to now, all applications developments of CHIP have been done with a dedicated interpreter.

3.1 Why an embedding in LP

Interesting extensions to LP should be done by embedding them in LP, which entails modifying the interpreter or the compiler. The alternative to embedding extensions to a compiler (interpreter), while rejecting the programmed and amalgamating techniques (Gallaire, 1987), is to use packages that are called from the core language. To take a simple example one could think of CHIP as a set of procedures which are specialized for solving particular sets of constraints. This is the typical and successful approach used in numerical analysis where packages exist and where the programmer calls adequately selected procedures for each task at hand. Such an approach has been suggested, for instance, in the case of applications of CHIP to formal proofs of VLSI circuits design (where the boolean unification algorithm of CHIP is the major tool used); the idea was to use from a Lisp based program such specialized features and more generally the constraint propagation mechanism. This would permit to use powerful black boxes, such as (Lauriere, 1978). The richness of the CHIP approach comes from the very tight interaction between a programming language and a constraint solver. How to use the very tight interaction is clear from the examples which have been developed up to now. This has been argued in (Jaffar, 1987) to justify the use of an incremental simplex algorithm for solving linear equations over the real numbers. The CHIP experience certainly coincides and suggests going even further. CHIP also embeds the notion of finite domains (Van Hentenryck, 1986) and has a number of primitives associated to them. The interaction between these built-in predicates and the Prolog part of the program is essential to the kind of tuning that is useful, difficult to dispense with and to replace by high level control of a black box. This is now to be explained.

A CHIP program can be seen as having two highly interacting components: the high level decision maker (which sets up constraints), and the constraint manipulator and solver. It is agreed that the constraint solver needs to be incremental. The state of the constraint solver influences the next constraints that will be put in place and consequently the relationship between the two components is not a master-slave relationship. In other words, the (partial) constraints solving operation which takes place does not just return values: it influences the next steps of the higher level decision process. A typical example of this will be the detection of a contradiction by the solver which causes immediate backtrack in the decision process. Another example is that of a search algorithm for an optimisation problem which interleaves estimations provided by the constraint solver with the choice of the next branch to exploit (as a matter of fact this example occurs in the CHIP system rather frequently as a metapredicate provides a form of branch and bound algorithm). There are computation domains where the constraint solver embeds an algorithmic deterministic decision procedure, that is not to be controlled tightly; this is the case for example of booleans, of the reals and rationals, for the type of constraints handled. This does not apply for some other cases, e.g the finite domains of CHIP where, in practice, the constraint solver does not use a full decision procedure and it often only carries a partial task of manipulation of the constraints instead of solving the current set. Often, at a given stage, no manipulation is even possible; for example given a set of quadratic equations a constraint solver which is geared to solve or manipulate linear equations will not do much. This is where a labeling procedure, i.e. a

procedure which can generate guesses, is used. The functioning of the system is thus more accurately described by having a fully connected bi-directional graph made of the two components mentioned and of a labeling component. The more knowledgeable the guess necessary to label variables will be made, the better. To make such educated guesses, the labeling procedure must be able to share information with the current state of the constraint solver. Again, a tight connection is fundamental in order to take full advantage of the technique. One must also observe that the labeling procedure usually sets up simple constraints, in particular equalities (i.e. it assigns values to variables). The way these equalities are generated corresponds to heuristics. The labeling procedure explores the totality of potential values using a user-defined algorithm (e.g. a dichotomic search): new constraints, for instance equalities resulting from the fact that a unification has taken place, will be added on request, in a certain order. The practical importance of this generation and of the freedom in the choice of labeling procedure is that the interaction between the components of the system is not done after a complete labeling, but rather, potentially, a labeling of just one variable at a time. More complex things may happen during the labeling process, such as setting truly new constraints, for example when dealing with disjunctive constraints: the labeling procedure chooses one constraint and the process continues; if no solution satisfying all constraints is found, backtracking will occur and another constraint will be put in place. This is routinely used, and simple to implement, in many scheduling problems (Dincbas, 1988c). With traditional approaches disjunctive scheduling is considered a difficult task. All programs using AI techniques that can deal with sizable examples of disjunctive constraints seem to assume that an allocation of ressources is made first; then the scheduling takes place. It is precisely because, in CHIP, all three components have tight links and share data structures and control information, that disjunctive scheduling becomes easily and elegantly tractable, although its complexity may remain prohibitive in some cases (the general problem remains NP-complete).

To summarize the argument, there is little doubt that only a fully integrated approach can deliver the type of performance necessary to solve interesting and difficult problems; furthermore, and perhaps more importantly for most users, the programming ease which is obtained is without counterpart. Attaching a constraint solver to a Lisp program, or to a C program, presents limitations (Sussman, 1980, Borning, 1981). The fact that these are frameworks dealing with much more limited types of constraints and of applications supports the thesis of the full integration in LP. One must also note that they do not appear to be particularly more efficient than embedded systems such as CHIP, nor to solve larger problems, on the contrary. It is also interesting to see that work on ATMS, which started from the black box approach, is now been extended to improve the link between the ATMS and the inference engine (Forbus, 1988).

3.2 How to use the embedding techniques

The interface between the high level decision maker and the constraint manipulation tool is, by the very nature of the approach, programmable and controllable. There is scope for user's involvement in order to improve the behaviour of his/her programs. This interface may be decomposed in two parts due to the existence of the labeling procedure. This makes more space for fine tuning. Indeed there are many instances where radical speed improvements may be obtained by choosing better labeling procedures. This is precisely the place where heuristics will be expressed. But there are two ways to consider these heuristics as the following examples show, depending on their specificity or generality:

- As example where specificity dominates, consider a problem of allocation of warehouses to cities (Van Hentenryck, 1988); it is natural to use as labeling procedure one which assigns first the warehouses close to the biggest customers? Such application dependent heuristics are trivial to implement by writing simple LP (Prolog) programs. This is but an instance of the general class of all allocation problems. Considering the fact that constraints bearing on these problems will take quite different forms, there does not appear to be any good reason to go further and to provide for such heuristics general schemes which would only need to be instantiated for each given application. Note in passing that certain user-defined constraints may not have a counterpart in a mathematical modeling. Another example of specificity is the one of a problem of cutting stocks (Dincbas, 1988a); variables which must be allocated a configuration (of stocks) are subject to being labelled: a dichotomic search proves to be very efficient. Here again there is no need to provide primitives to support this simple programming task.
- There are however cases where general heuristics can be used. A quite different labeling procedure is necessary to solve another problem of allocation, namely to assign planes to itineraries, minimizing the overall cost of the fleet. There is a well-known technique used in Operations Research (Gondrand, 1984), the so-called maximum regret technique. Basically it is as follows: given two variables X and Y which take respectively their value among the elements of a set A and of a set B, and given some function of X and Y to be, say, maximised, then one should first assign a value to the variable which among its possible values, will have the biggest difference in its two highest values, and the value

assigned is the highest of those two. Although this is a semantic heuristics, it is frequent enough and too inefficient if programmed at the LP level, that a primitive is provided which does precisely that. This is an example of a lower level interface between the labeling component and the constraint solver component which needs to make available to the labeling component the whole state of the solver; thus this is yet another argument for integration. Such a heuristics is a basis for many other applications where classically it provides good results: for example the traveling salesman problem where it brings an order of magnitude improvement over a random labeling. There are many other such techniques that can be embedded in a system like CHIP. Another example of low-level general purpose heuristics that need to be primitives, is the heuristics that prefers to label first the variables which have the smallest domain. Another similar case is to use the most constrained variable first.

There is a significant pay-off in following this approach: a form of incremental programming, albeit a limited one. This approach supports the possibility to add much more easily new constraints to a problem, and to host constraints of a very different nature in the same problem. A black box approach will not allow at all this type of flexibility.

In summary, the labeling procedure plays a very important role in making good use of the technique of embedding constraint solver and high level decison maker. Two remarks must, however, be made:

- There are limits to this approach to constraint solving: approaches which model rather naively a problem and use general techniques guided by heuristics will not be always competitive with those using specialised modeling, specialised techniques guided by heuristics. What happens here is best understood by analogy with the comparison between conventional programming and logic or other declarative systems. The equation "algorithm = logic + control" (Kowalski, 1979) has limitations in practice as remarked by several authors (McCarthy, 1982, Bundy, 1988): starting from a given logical expression of a program it may be outside the scope of any control structure to turn it into the most efficient algorithm. It is very doubtful, to say the least, that more control will ever transform the 40 line CHIP program, which solves a traveling salesman problem of respectable complexity, into one which does as well as a very performant system (Padberg, 1986) which uses a sophisticated (and recent) mathematical technique based on polyhedral theory and complex heuristics and involves 9700 lines of Fortran, plus the use of mathematical packages. There are similar examples in particular in the domain of generating tests for electronic circuits. This however, is not a serious criticism of the approach in any way, in view of the many industrial cases where it has shown to be very competitive indeed (Dincbas, 1988c, Dincbas, 1988b). Research work on embedding new decision procedures or heuristics continues anyway.

- General heuristics have been studied for many years e.g. in the OR field; the example previously mentioned of the maximum regret heuristics is not unique. More must be learnt from these fields in order to integrate in the systems' knowledge, in one form or another (namely by a primitive or a direct ad-hoc encoding) such available techniques. For instance, the traveling salesman problem benefits from another such OR technique, which obtains lower bounds used to cut the search by examining not only rows but also columns of the connectivity matrix.

Declarativeness may suffer from using these highly specialised tools which can have complex operational semantics, since the interface on which they act may be low level (e.g. the most constrained variable first heuristics). There is no need to be worried unduly here. A simple declarative semantics for programs written in CHIP is retained. The operational semantics is of course much more difficult to grasp as just pointed out. This is not a limitation of the approach; indeed the behavior of a fully declarative approach for most of these problems is well-known: it cannot cope with them reasonably, due to performance reasons; this is unfortunately also true when one writes in LP a complete algorithm for solving the problem.

It is also remarkable that programs can be fine tuned by experience. Performances can be drastically improved by changing a labeling procedure, adding redundant constraints, etc. There is definitely a programming style to be learnt.

3.3 Going too far?

In the previous sections I have argued in favour of a very tight integration of specialised mechanisms in LP in order to support multiple reasoning styles and discussed in particular the constraint mechanism integration. The obvious counterpart to this discussion is that there must be a limit to the addition of new primitive functions or new constraint solving techniques which may require modifying in depth the system. First note that the answer depends on how the system has been built. A system like SEPIA, which is hosting CHIP, and CHIP itself, have shown that extensions are possible and need not be incompatible. Thanks to a careful analysis of this problem, it is

possible for them to coexist and even interact (types and constraints, types and delays etc.). Nevertheless, it is not costless to provide extensions and the problem of their usefulness is to be studied carefully. For example, should one provide a general decision procedure for Presburger arithmetic (Voda, 1988b), or consider that this is not worth the effort when corresponding solutions based on constraint propagation (an existing feature) and labeling can be often used more efficiently? Another such example is that of dealing with Post algebras instead of boolean algebras. Such an extension to Post algebras is proposed in (Tiden, 1988) and implemented; on the contrary CHIP provides a built-in boolean algebra only. The use of Post algebra to model circuits at the switch level or to model parts that may have more than a zero-one behavior (e.g high impedance, short circuit, etc) can be simulated with boolean algebra and state information and using the other efficient mechanisms in CHIP. In fact this corresponds to a more general technique used in the analysis of devices for which no algebraic structure (but for an artificial one) exists (Graf, 1988); it does provide generality and efficiency in such difficult problems. All these more specialised algorithms can nevertheless find a niche for applications. Other and powerful decisions procedures, e.g combining logic and algebraic programming are being considered. Another type of specialisation can be seen in Prolog III (Colmerauer, 1987) which supports a specific representation of lists which provides an easier to use model of lists, and may yield some speed-up in manipulating them.

For new extensions, the results of an analysis of the performance improvements they bring, if at all, and of their applicability scope, must be weighted against their costs of implementation. This means that engineering considerations will eventually be the major factor in the decisions to be taken to go beyond the existing set of extensions.

Thus, there is probably no good answer to the question as it was raised. General systems, not geared at a very specific class of problems, should be built around the principles that guided CHIP, where generality (for the domains of discourse chosen) ranked very high on the list, as well as efficiency. The fact that CHIP has been applied successfully to so many different domains is a clear validation of these ideas.

4 CONCLUSION

In this paper I have reviewed a number of reasoning styles and analysed how they were supported by standard LP. I have argued in favor of the use of features that extend significantly the scope of LP in order to support these reasoning styles. To successfully make available these features to the user, I have shown that they had to be provided in a tightly integrated manner. This was stated, but not discussed in (Gallaire, 1987). Throughout, one of these features, fitting the so-called constraint programming paradigm, has been used to support the argumentation. To summarize, logic programming, as any language, does not convince; only solutions making a heavy use of LP, and within the LP framework, will do. If this means we must extend it, let us do so. This proposal ends up giving more work to implementors and involves system changes. The systems mentioned in this paper [CHIP, CLP, Prolog III, Trilogy,..] indicate that the work involved is not overwhelming. Fortunately, not only implementors will be kept busy with our approach: new challenging problems are now open as we can be free to take advantage of the knowledge coming from other disciplines when it can fit with LP. And of course, all of this has to be reviewed in the light of the ever present challenge and opportunity for parallelism.

ACKNOWLEDGMENTS

The ideas expressed here stem out of the observation of the work carried out at ECRC, in particular in the Logic Programming Group. I am grateful to all in the SEPIA, ODE and CHIP teams for taking LP so seriously and making the efforts necessary to validate the ideas, implementing systems and developing applications. A. Herold global responsibility in these efforts is acknowledged. M. Dincbas and his colleagues in the CHIP team are the sources of many of the ideas expressed here. I hope I understood them well. M. Dincbas has also helped improve significantly the drafts of the paper. My thanks to all at ECRC.

REFERENCES

(Ait Kaci, 1986)
H. Ait Kaci and R. Nasr: LOGIN: a logic programming language with built-in inheritance.
Journal of Logic Programming 3.3:185-215, 1986.

(Bacha, 1988)
H. Bacha: MetaProlog design and implementation. In *5th Int. Conf. and Symp. on Logic Programming*, pages 1371-1387. Seattle, Aug, 1988.

(Berthier, 1988)
F. Berthier: Using CHIP to Support Decision Making. In *Actes du Seminaire 1988 Programmation en Logique*. CNET, Tregastel, France, Mai, 1988.

(Borning, 1981)
A. Borning: The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems* 3(4):353-387, 1981.

(Bosco, 1988)
P. Bosco, C, Cecchi and C. Moiso: Exploiting the full power of logic plus functional programming. In *5th Int. Conf. and Symp. on Logic Programming*, pages 3-12. Seattle, Aug, 1988.

(Bourgault, 1984)
S. Bourgault, M. Dincbas and J-P. Lepape: LISLOG : Programmation en Prolog en environnement Lisp. In *Actes du Congres AFCET - Reconnaissances des Formes et Intelligence Artificielle*, pages 275-289 (tome II). Paris, France, Janvier, 1984.

(Bowen, 1982)
K. Bowen, R. Kowalski: Amalgamating language and metalanguage in logic programming. *Logic Programming (eds: K. Clark, S. Tarnlund).* Academic Press, 1982, pages 153-172.

(Brachman, 1985)
R. Brachman, V. Pigman Gilbert and H Levesque: An essential hybrid reasoning system- knowledge and symbol level accounts of Krypton. In *IJCAI 85*, pages 532-539. Los Angeles, Aug, 1985.

(Bruynooghe, 1984)
M. Bruynooghe and L.M. Pereira: Deduction revision by intelligent backtracking. *Implementations of Prolog (ed: J. A. Campbell).* Ellis Horwood, 1984.

(Buettner, 1987)
W. Buettner, H. Simonis: Embedding boolean expressions into logic programming. *J. of Symbolic Logic* 4:191-205, 1987.

(Bundy, 1988)
A. Bundy: A broader interpretation of logic in logic programming (inv. speaker). In *5th Int. Conf. and Symp. on Logic Programming*, pages 1624-1648. Seattle, Aug, 1988.

(Chan, 1987)
D. Chan, P. Dufresne, R. Enders: *Phocus.* Technical Report, ECRC, Feb, 1987. TR-LP-20.

(Chatalic, 1987)
Ph. Chatalic: *Incremental techniques and Prolog*. Technical Report, ECRC, June, 1987. TR-LP-23.

(Clark, 1978)
K. Clark: Negation as failure. *Logic and Databases, (eds: H. Gallaire and J. Minker).* Plenum Press, 1978, pages 293-322.

(Codognet, 1988)
C. Codognet, P. Codognet and G. File: Yet another intelligent backtracking method. In *5th Int. Conf. and Symp. on Logic Programming*, pages 447-465. Seattle, Aug, 1988.

(Colmerauer, 1981)
A. Colmerauer, H. Kanoui and M. van Caneghem: Last steps towards an ultimate Prolog. In *7th IJCAI*, pages . Vancouver, Aug, 1981.

(Colmerauer, 1987)
A. Colmerauer: Opening the Prolog-III Universe. *BYTE Magazine* 12(9), August, 1987.

(Conery, 1988)
J. Conery: Logical Objects. In *5th Int. Conf. and Symp. on Logic Programming*, pages 420-434. Seattle, Aug, 1988.

(Davis, 1984)
R. Davis: Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence* 24:347-410, 1984.

(De Kleer, 1987)
J. De Kleer and B. Williams: Diagnosing multiple faults. *Artificial Intelligence* 32:97-130, 1987.

(Dincbas, 1987)
M. Dincbas, H. Simonis and P. van Hentenryck : Extending Equation Solving and Constraint Handling in Logic Programming. In MCC (editor), *Colloquium on Resolution of Equations in Algebraic Structures (CREAS).* Texas, May, 1987.

(Dincbas, 1988a)
M. Dincbas, H. Simonis and P. van Hentenryck: Solving a Cutting-Stock Problem in Constraint Logic Programming. In M.I.T Press (editor), *Fifth International Conference on Logic Programming.* Seattle, USA, August, 1988.

(Dincbas, 1988b)
M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier: The Constraint Logic Programming Language CHIP. In *Int. Conf. on Fifth Generation Computer Systems (FGCS'88).* Tokyo, Japan, December, 1988.

(Dincbas, 1988c)
M. Dincbas, H. Simonis and P. van Hentenryck: Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming* , 1988. (To appear).

(Enderton, 1972)
H. Enderton. *A mathematical introduction to Logic*. Academic Press, 1972.

(Eshgi, 1988)
K. Eshgi: Abductive planning with event calculus. In *5th Int. Conf. and Symp. on Logic Programming*, pages 562-579. Seattle, Aug, 1988.

(Forbus, 1988)
K. Forbus and J. de Kleer: Focusing the ATMS. In *AAAI 88*, pages 193-198. Saint Paul, Aug, 1988.

(Gallaire, 1982)
H. Gallaire and C. Lasserre: Metalevel control for logic programs. *Logic Programming (eds: K. Clark and S.A. Tarnlund).* Academic press, 1982.

(Gallaire, 1987)
H. Gallaire: Boosting Logic Programming. In *Proc. Fourth International Conference on Logic Programming*, pages 962-988. Melbourne, Australia, May, 1987.

(Gondrand, 1984)
M. Gondran and M. Minoux. *Graphs and Algorithms.* John Wiley & Sons, Chichester New York Brisbane, 1984.

(Graf, 1988)
T. Graf, P. Van Hentenryck, C. Pradelles and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. Forthcoming paper. 1988

(Huynh88, 1988)
T. Huynh and C. Lassez: A CLP(R) options trading analysis system. In *5th Int. Conf. and Symp. on Logic Programming*, pages 59-69. Seattle, Aug, 1988.

(Jaffar, 1987)
J. Jaffar and J-L. Lassez: Constraint Logic Programming. In *POPL-87*. Munich (FRG), January, 1987.

(Kaufmann, 1986)
H. Kaufmann and A. Grumbach: Multilog; multiple worlds in logic programming. In *ECAI 86*, pages . Brighton, Aug, 1986.

(Kowalski, 1979)
R. Kowalski: Algorithm = Logic + Control. *CACM* 22:424-436, 1979.

(Lauriere, 1978)
J-L. Lauriere: A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence* 10(1):29-127, 1978.

(McCarthy, 1982)
J. McCarthy: *Coloring maps and the Kowalski doctrine*. Technical Report, Stanford, , 1982. STAN-CS-82-903.

(Meier, 1988)
M. Meier, P. Dufresne and D. de Villeneuve: *SEPIA*. Technical Report, ECRC, March, 1988. TR-LP-36.

(Montanari, 1974)
U. Montanari: Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Science* 7(2):95-132, 1974.

(Nadathur, 1988)
G. Nadathur and D. Miller: An overview of λ Prolog. In *5th Int. Conf. and Symp. on Logic Programming*, pages 810-827. Seattle, Aug, 1988.

(Naish, 1985)
L. Naish: *Negation and control in Prolog*. Technical Report, University of Melbourne CS Dept, 1985. Ph. D. thesis.

(Okhi, 1986)
M. Okhi, A. Takeuchi and K. Furukawa: *A framework for interactive problem solving based on interactive query revision*. Technical Report, ICOT, June, 1986. TR-188.

(Owen, 1988)
S. Owen and R. Hull: The use of explicit interpretation to control reasoning about protein topology. In *ECAI 88*, pages 308-313. Munich, Aug, 1988.

(Padberg, 1986)
M. Padberg and G. Rinaldi: Optimization of a 532-city symmetric travelling salesman problem. In *AFCET Combinatorial Group 20th Anniversary*, pages 389-403. INRIA, Dec, 1986.

(Poole, 1986)
D. Poole: *Default reasoning and diagnosis as theory formation*. Technical Report, U. of Waterloo, , 1986. TR-CS-86-08.

(Reiter, 1987)
R. Reiter: A theory of diagnosis from first principles. *Artificial Intelligence* 32:57-95, 1987.

(Robinson, 1980)
J.A. Robinson and E. Siebert: *Logic programming in Lisp*. Technical Report, Syracuse University, , 1980. TR School of Comp. and Inf. Sc.

(Simonis, 1987)
H. Simonis and M. Dincbas: Using Logic Programming for Fault Diagnosis in Digital Circuits. In *German Workshop on Artificial Intelligence (GWAI-87)*, pages 139-148. Geseke, W.Germany, September, 1987.

(Sussman, 1980)
G.J. Sussman and G.L. Steele: CONSTRAINTS-A Language for Expressing Almost-Hierarchical Descriptions. *AI Journal* 14(1), 1980.

(Tiden, 1988)
E. Tiden: Symbolic Verification of Switch-Level Circuits Using a Prolog Enhanced with Unification in Finite Algebras. In *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*. Glasgow, Scotland, July, 1988.

(Van Hentenryck, 1986)
P. Van Hentenryck and M. Dincbas: Domains in Logic Programming. In *AAAI-86*. Philadelphia, USA, August, 1986.

(Van Hentenryck, 1988)
P. Van Hentenryck and J-P. Carillon: Generality versus specificity- an experience with AI and OR techniques. In *AAAI 88*, pages 660-664. Saint Paul, Aug, 1988.

(VanEmden, 1984)
M. Van Emden: Logic as an interaction language. In *5th Conf. Canadian Soc. for Computational Studies in Intelligence*, pages 126-128. , May, 1984.

(Voda, 1988a)
P. Voda: Types of Trilogy. In *5th Int. Conf. and Symp. on Logic Programming*, pages 580-589. Seattle, Aug, 1988.

(Voda, 1988b)
P. Voda. The Constraint Language Trilogy: Semantics and Computations. TR-Complete Logical System. 1988

(Walther, 1984)
C. Walther: A mechanical solution to Schubert's steamroller by many-sorted resolution. In *AAAI 84*, pages 330-334. Austin, Aug, 1984.

(Xu, 1988)
J. Xu and D.S. Warren: A type inference system for Prolog. In *5th Int. Conf. and Symp. on Logic Programming*, pages 604-619. Seattle, Aug, 1988.