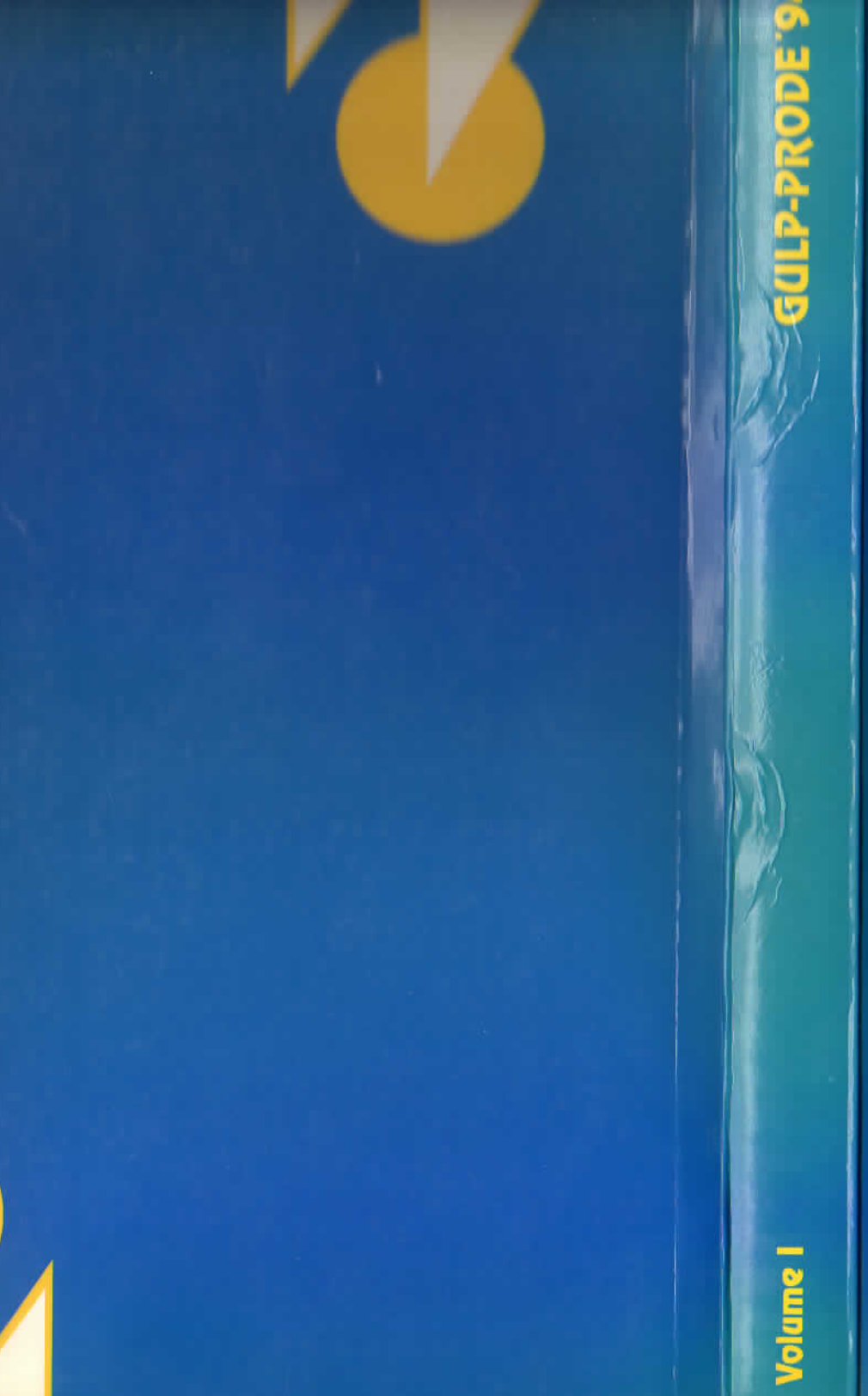

GULP-PRODE'94

Volume I

GULP-PRODE'94

1994 JOINT CONFERENCE ON DECLARATIVE PROGRAMMING

Editors:

M. Alpuente

R. Barbuti

I. Ramos

PEÑÍSCOLA, SPAIN
SEPTEMBER 19-22, 1994

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

María Alpuente Roberto Barbuti Isidro Ramos (Eds.)

Declarative Programming

1994 Joint Conference, GULP-PRODE'94
Peñíscola, Spain, September 19-22, 1994
Proceedings

Volume I

GULP-PRODE'94

UNIVERSIDAD POLITÉCNICA DE VALENCIA
46071 - Valencia, SPAIN

SPUPV-94.2046

Diseño Portada: SUSANA VIDAL

Edita: SERVICIO DE PUBLICACIONES
Camino de Vera, s/n
46071 VALENCIA
Tel. 96-387 70 12
Fax. 96-387 79 12

mprime: REPROVAL, S.L.
Tel. 96-369 22 72

)epósito Legal: V-2559-1994

Foreword

The Gruppo Ricercatori ed Utenti di Logic Programming (GULP, which stands for Logic Programming Researcher and User Group), is an affiliate of the Association of Logic Programming (ALP). The goals of the group are to make logic programming more popular and to create opportunities for the exchange of experiences and information between researchers and users working in the area for both public and private organizations.

To this purpose GULP promotes many different activities such as the exchange of information among its members and the organization of workshops, advanced schools and its annual Conference.

This time the Conference is going to be held in Spain together with the Spanish Conference PRODE on Declarative Programming. This represents a significative step towards the exchange of research experience among European latin countries. Such a goal had already been pursued by GULP with the enlargement of its conferences to Spanish and French researchers.

The main aims of this Conference are: 1) to serve as an occasion for those working in this area which are interested in meeting and exchanging experiences; 2) to illustrate the current state of the art in the area through invited talks given by well known researchers; 3) to enable students and researchers to learn more about logic and declarative programming by means of introductory tutorials.

The previous annual conferences were held in Genoa, Turin, Rome, Bologna, Padua, Pisa, Tremezzo, and Gizzeria Lido, and were attended each year by a growing number of participants.

The scientific program of this joint Conference GULP-PRODE includes papers from colleagues from many countries. The large international participation, considered together with the good technical quality of the papers accepted for presentation, is a further confirmation of the success of this joint event.

On behalf of GULP I would like to thank María Alpuente and all the other colleagues from the Technical University of Valencia for the organization of this year Conference.

July 1994
Genoa

Maurizio Martelli
President of GULP

Preface

This book contains the Proceedings of the 1994 Joint Conference on Declarative Programming GULP-PRODE'94 held in Peñíscola (Spain) September 19-22, 1994. GULP-PRODE'94 joins the Italian GULP *Conference on Logic Programming* and the Spanish PRODE *Congress on Declarative Programming*. This is the first time that GULP and PRODE join together. GULP-PRODE'94 was preceded by the eight previous GULP conferences in Genova (1986), Torino (1987), Roma (1988), Bologna (1989), Padova (1990), Pisa (1991), Tremezzo (1992) and Gizzeria (1993) and the three previous PRODE meetings in Torremolinos (1991), Madrid (1992) and Blanes (1993). GULP-PRODE'94 has been organized by the *Universidad Politécnica de Valencia*.

The technical program for the conference includes 60 refereed papers, 8 posters, 3 invited lectures by J.W. Lloyd (U. Bristol), J.-P. Jouannaud (U. Paris Sud) and P. Van Roy (DEC-PRL) and 2 tutorials by J.J. Moreno-Navarro (U.P. Madrid) and R. Nieuwenhuis and A. Rubio (U.P. Catalunya). We thank them for their willingness in accepting our invitation.

The papers in this book are printed in the order of presentation and thus grouped into sessions, all of which are thematic. All the papers were evaluated by at least two reviewers. In order for more work to be presented than would be possible from just the contributed papers, a poster session has also been incorporated into the program and the abstracts appear in this book.

We wish to express our gratitude to all the members of the program committee and all the referees for their care in reviewing and selecting papers. We gratefully acknowledge all the institutions and corporations who have supported this conference. We would like to extend our sincere thanks to all authors who submitted papers and all conference participants. We would also like to thank Riki Vandevoorde for her invaluable advice about the organization. Finally, we would like to especially highlight the contribution of the organizing committee whose work has made the conference possible.

July 1994
Valencia

María Alpuente
Roberto Barbuti
Isidro Ramos

Program Committee

María Alpuente	(U.P. Valencia)
Roberto Barbuti	(U. Pisa)
Luigia Carlucci Aiello	(U. Roma)
Paolo Ciancarini	(U. Bologna)
Nicoletta Cocco	(U. Venezia)
Philippe Codognet	(INRIA, France)
Carlos Delgado	(U.P. Madrid)
Moreno Falaschi	(U. Padova)
Ana García	(U.P. Madrid)
Pere García	(IIIA Blanes)
Laura Giordano	(U. Torino)
Manuel Hermenegildo	(U.P. Madrid)
María Teresa Hortalá	(U.C. Madrid)
Evelina Lamma	(U. Bologna)
Paolo Mancarella	(U. Pisa)
Juan José Moreno	(U.P. Madrid)
Marisa Navarro	(U. País Vasco)
Robert Nieuwenhuis	(U.P. Catalunya)
Mario Ornaghi	(U. Milano)
Catuscia Palamidessi	(U. Genova)
Inmaculada Pérez	(U. Málaga)
Maurizio Proietti	(IASI-CNR)
Isidro Ramos	(U.P. Valencia)
Mario Rodríguez	(U.C. Madrid)
Gianfranco Rossi	(U. Bologna)
José Jaime Ruz	(U.C. Madrid)
Domenico Saccà	(U. Calabria)
Maria Sessa	(U. Salerno)
Jaume Sistac	(U.P. Catalunya)
José María Troya	(U. Málaga)
Felisa Verdejo	(UNED)

Organizing Committee

Asunción Casanova	Salvador Lucas	María José Ramírez
Carlos García	Javier Oliver	María José Ramis
Vicente Gisbert	Javier Piris	Germán Vidal

List of Referees

J. Agustí	M. Celma	R. Giacobazzi	D. Nardi	G. Puebla
L. Araujo	P. Ciaccia	F. Giannotti	J. Oliver	J. Puyol
R. Bagnara	C. Codognet	A. Gil	N. Olivetti	C. Quer
E. Bertino	M. Díaz	S. Greco	A. Omicini	M.J. Ramírez
A. Bossi	A. Dovier	A. Herranz	F. Orejas	J.M. Rivero
P.T. Breuer	M. Fabris	J. Leach	L. Palopoli	H. Rodríguez
A. Brogi	M.M. Gallardo	J. Mariño	M.A. Pastor	A. Ruiz
F. Bueno	J. García	A. Martelli	D. Pedreschi	F. Sáenz
N. Busi	M.I. García	C. Martín	W. Penzo	L. Sánchez
D. Cabeza	M.J. García	F. Massacci	A. Pettorossi	M.L. Sapino
M. Carro	A. Gavilanes	P. Mello	E. Pimentel	F. Scarcello
J.C. Casamayor	G. Ghelli	L. Moreno	E. Plaza	E. Teniente

Table of Contents

Volume I

Table of Contents

Volume II

Automated Deduction with Constrained Clauses

Robert Nieuwenhuis and Albert Rubio

Universidad Politécnica de Cataluña
Dept. Lenguajes y Sistemas Informáticos
Pau Gargallo 5, E-08028 Barcelona, Spain
{roberto,rubio}@lsi.upc.es

Abstract

The aim of automated deduction (AD) is proving (semi-)automatically the validity of a formula, for instance by refutation. It is well-known that, among other applications, AD was the initial basis for logic programming. Like in Constraint Logic Programming, in refutational theorem provers the use of clauses with (symbolic) constraints has become a very useful approach, especially when working with built-in equality (i.e. with inference rules like paramodulation) or built-in equational theories. For instance, basic strategies (as in basic narrowing) can be represented by means of equality constrained clauses and proved refutationally complete. In this tutorial, deduction techniques for equality and ordering constrained first-order clauses are discussed, completeness results are given and the corresponding constraint solving problems are studied.

Integration of Functional and Logic Programming

Juan José Moreno-Navarro

LSIIS, Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo, Boadilla del Monte
28660 Madrid, Spain
jjmoreno@fi.upm.es

Abstract

The integration of declarative paradigms and, in particular, the integration of functional and logic programming, was an active area of research some years ago. New models and languages were proposed and it was shown that the new paradigm is feasible.

Recently, interest for this kind of languages has been spurred anew, especially for what concerns the improvement of the expressive power of the language as well as the efficient implementation.

The aim of this tutorial is to show the recent advances in the area especially in the following topics:

- Design principles of integrated programming languages accounting for a mature and up-to-date understanding of previously studied features meant to improve the expressive power of declarative languages: lazy evaluation, higher-order objects, types, constraints, negation, default rules, etc.
- Semantics models for integrated languages, in particular the role of the different *narrowing* strategies as operational semantics of the (different) features of the languages.
- Development of efficient implementations by combining existing techniques from logic programming (e.g., WAM, mode analysis, abstract interpretation) and functional programming (e.g., graph reduction, strictness analysis, optimization of deterministic computations) as well as new techniques (e.g., combination of normalization and narrowing, parallelization).

Practical Advantages of Declarative Programming

J.W. Lloyd

Department of Computer Science
University of Bristol
Bristol, BS8 1TR, U.K.

Abstract

In this paper, I discuss the practical advantages of declarative programming. I begin with a discussion of the possible meanings of the term "declarative" and then go on to present the practical advantages of declarative programming under five headings: teaching, semantics, programmer productivity, meta-programming, and parallelism. I then make some general remarks about the extent to which declarativeness is taken seriously even by researchers in declarative programming, the possibilities for combining functional and logic programming, and, finally, a significant limitation of the current declarative approach to programming. I conclude with a summary of the main points made in the paper.

1 Declarative Programming

Informally, declarative programming involves stating *what* is to be computed, but not necessarily *how* it is to be computed. Equivalently, in the terminology of Kowalski's equation *algorithm* = *logic* + *control*, it involves stating the *logic* of an algorithm, but not necessarily the *control*. This informal definition does indeed capture the intuitive idea of declarative programming, but a more detailed analysis is needed so that the practical advantages of the declarative approach to programming can be explained.

I begin this analysis with what I consider to be the key idea of declarative programming, which is that

- a program is a theory (in some suitable logic), and
- computation is deduction from the theory.

What logics are suitable? The main requirements are that the logic should have a model theory, a proof theory, a soundness theorem (that is, computed answers should be correct), and, preferably, a completeness theorem (that is, correct answers should be computed). Thus most of the better-known logics including first order logic and a variety of higher order logics qualify. For example, (unsorted) first order logic is

the logic of (pure!) Prolog, polymorphic many-sorted first order logic is the logic of Gödel [5], and λ-calculus is the logic for many functional languages, such as Haskell [1]. In the context of the present discussion, the most crucial of these requirements is that the logic should have a model theory because, as I shall explain below, this is the real source of the "declarativeness" in declarative programming.

This view of declarative programming shows that the concept is wide-ranging. It includes logic programming and functional programming, and intersects significantly with other research areas such as formal methods, program synthesis, theorem proving, and algebraic specification methods, all of which have a strong declarative component. This view also highlights the fact that the current divide between the fields of functional and logic programming is almost entirely historical and sociological rather than technical. Both fields are trying to solve the same problems with techniques that are really very similar. It's time the gap was bridged.

In fact, the close connection between functional and logic programming is emphasized by the terminology used by Alan Robinson. He calls logic programming, *relational* programming, and he calls the combination of functional and relational programming, *logic* programming. This terminology is very apt, since it emphasizes that the dominant symbols of functional programs are functions and the dominant symbols of relational programs are relations (or predicates). With this terminology, we could effectively identify declarative programming with logic programming. It is a pity that terminology such as relational (or, perhaps, predicative) programming wasn't used from beginning of what is now called logic programming because, with hindsight, it is obviously more appropriate. However, in the following, we conform to the standard terminology to avoid confusion.

I return now to the discussion of the general principles of declarative programming. The starting point for the programming process is the particular problem that the programmer is trying to solve. The problem is then formalized as an interpretation (called the *intended* interpretation) of a language in the logic at hand. The intended interpretation specifies the various domains and the meaning of the symbols of the language in these domains. In practice, the intended interpretation is rarely written down precisely, although in principle this should always be possible.

Now, of course, it is taken for granted here that it *is* possible to capture the intended application by an interpretation in a suitable logic. Not all applications can be (directly) modelled this way and for such applications other formalisms may have to be employed. However, a very large class of applications can be modelled naturally by means of an interpretation. In fact, this class is larger than is sometimes appreciated. For example, it might be thought that such an approach cannot (directly) model situations where a knowledge base is changing over time. Now it is true that the intended interpretation of the knowledge base is changing. However, the knowledge base should properly be regarded as data to various meta-programs, such as query processors or assimilators. Thus the knowledge base can be accessed and changed by these meta-programs. The meta-programs themselves have fixed intended interpretations which fits well with the setting for declarative programming given above. However, as I explain below, there are limits to the applicability of declarative programming, as defined above, and it is important that these limitations be recognized.

Now, based on the intended interpretation, the *logic component* of a program is then written. This logic component is a particular kind of theory which is usually suitably restricted so as to admit an efficient theorem proving procedure. Typically, in logic programming languages, the logic component of a program is a theory consisting of suitably restricted first order formulas (often completed definitions, in the sense of Keith Clark [6]) as axioms. In other approaches to declarative programming, different logics are used. For example, in functional programming, the logic component of a program can be understood to be a collection of expressions in the λ-calculus. It is crucial that the intended interpretation be a model for the logic component of the program, that is, each axiom in the theory be true in the intended interpretation. This is because an implementation must guarantee that computed answers be true in all models of the logic component of the program and hence be true in the intended interpretation. Ultimately, the programmer is interested in *computing truth in the intended interpretation*. It should be clear now why the model theory is so important for declarative programming. The model theory is needed to support the concept of an intended interpretation which in turn is needed to state precisely *what* it is that should be computed.

Perhaps it is worth remarking that the term "model theory" is not one which is commonly used when functional programmers describe the declarative semantics of functional programs. The traditional functional programming account starts by mapping the constructs of the functional language back into constructs of the λ-calculus. Having done this, the declarative semantics of a program is given by its denotational semantics, that is, a suitable domain is described and the meaning of expressions is given by assigning them meanings in this domain. (An account of this is given in [2], for example.) However, while this kind of model is technically rather different to, say, a model of a polymorphic many-sorted theory, the essential idea is the same. In fact, I have a few problems with the functional approach to the declarative semantics. First, the constructions of domain theory are rather technical and it's not clear to me to what extent ordinary programmers can be given a sufficiently simple picture of the declarative semantics of their programs using this approach. Second, this approach is usually presented as a semantics which is constructed *from* the program, whereas I would prefer the model (that is, intended interpretation) was given first (once the alphabet of the application was defined), as it is the intended interpretation which is the first formalization of the application. Third, I think the *untyped* λ-calculus is really the wrong place to start. I much prefer the view that programs should be written as *typed* theories from the beginning, so that a better starting point for the (declarative and procedural) semantics of functional programs is a typed logic, for example, typed λ-calculus. So, for example, if a functional program is understood as a theory in type theory (say, Church's simple theory of types), then we can use Henkin models to capture the declarative semantics. Henkin models have the nice property that they generalize first order models in a natural way. Thus I expect that they should be understandable by ordinary programmers, although I have no real evidence to support this! This approach also seems to provide a suitable foundation for languages which combine functional and logic programming, a point to which I shall return later.

The programmer, having written a correct logic component of a program, may now need to give the *control component* of the program, which is concerned with control of the proof procedure for the logic. Whether the control component is needed, and to what extent, depends on the particular situation. For example, querying a deductive database with first order logic, which is a simple form of declarative programming, normally wouldn't require the user to give control information since deductive database systems typically have sophisticated query processors which are able to do find efficient ways of answering queries without user intervention. On the other hand, the writing of a large-scale Prolog program will usually require substantial (implicit and explicit) control information to be specified by the programmer. Generally speaking, apart from a few simple tasks, limitations of current software systems require programmers to specify at least part of the control.

Now declarative programming can be understood in two main senses. In the weak sense, declarative programming means that programs are theories, but that a programmer may have to supply control information to produce an efficient program. Declarative programming, in the strong sense, means that programs are theories and all control information is supplied automatically by the system. In other words, for declarative programming in the strong sense, the programmer only has to provide a theory (or, perhaps, an intended interpretation from which the theory can be obtained by the system). This is to some extent an ideal which is probably not attainable (nor, in some cases, totally desirable), but it does at least provide a challenging and appropriate target. Typical modern functional languages, such as Haskell, are rather close to providing strong declarative programming, since programmers rarely have to be aware of control issues. On the other hand, typical modern logic programming languages, such as Gödel, provide declarative programming only in the weak sense. The difference here is almost entirely due to the complications caused by the (explicit) non-determinism provided by logic languages, but not by functional languages.

The issue of how much of the control component of a program a programmer needs to provide is, of course, crucially important in practice. However, systems which can be used both as weak and, on occasions, strong declarative programming systems are feasible now and, indeed, may very well survive long into the future even after the problems of providing strong declarative programming in general have been overcome. The point here is that, for many applications, it really doesn't matter if the program is not as efficient as it could be and so strong declarative programming, even in its currently very restricted form, is sufficient. However, for other applications, the programmer may supply control information, either because the system is not clever enough to work it out for itself or because the programmer has a particular algorithm in mind and wants to force the system to employ this algorithm. Generally speaking, even strong declarative programming systems should provide a way for programmers to ensure that desired upper bounds on the space and/or time complexity of programs are not exceeded. There is no real conflict here – the system can either be left to work out the control for itself or else control facilities (and, very likely, specific information about the particular implementation being used) should be provided so that a programmer can ensure the desired space and/or time requirements are met.

With these preliminaries out of the way, I now discuss the practical advantages

of declarative programming under five headings: (a) teaching, (b) semantics, (c) programmer productivity, (d) meta-programming, and (e) parallelism.

2 Teaching

Typical Computer Science programming courses involve teaching both imperative and declarative languages. On the imperative side, Modula-2 is a common, and excellent, choice. On the declarative side, Miranda or Haskell are typical functional languages used and Prolog is the usual choice of logic language. Miranda and Haskell are excellent for teaching as, to a very large extent, students only have to concern themselves with the logic component of a program.

In the last couple of years, there has been some discussion in the logic programming community about the use of Prolog as a teaching language and, in particular, as a first language. A remarkable characteristic of this debate is that, while most logic programmers argue that Prolog should be taught *somewhere* in the undergraduate curriculum, almost nobody is arguing that it should be the *first* teaching language! Usually, the reason given for avoiding Prolog as the first language is that its non-logical features make it too complicated for beginning programmers. I think this is true, but I would also add as equally serious flaws its lack of a type system and a module system. This situation is surely a serious indictment of the field of logic programming. In spite of its claims of declarativeness, until recently at least, there hasn't been a single logic programming language sufficiently declarative (and hence simple) to be used successfully as a first teaching language.

I think that the arrival of Gödel could change this situation. Gödel fits much better than Prolog into the undergraduate and graduate curricula since it has a type system and a module system similar to other commonly used teaching languages such as Miranda and Modula-2. Also most of the problematical non-logical predicates of Prolog simply aren't present in Gödel (they are replaced by declarative counterparts) and so the cause of much confusion and difficulty is avoided. This case has been argued in greater detail elsewhere [5] and I refer the reader to the discussion there. My own experience with teaching Gödel to masters students at Bristol has been very encouraging. For example, a common remark by students at the end of the course is that they find the Gödel type system so helpful that they really don't want to go back to using Prolog! In any case, I would strongly encourage teachers of programming courses to try for themselves the experiment of using Gödel instead of Prolog.

So let us make the assumption that we have a suitable declarative language available (Haskell or Gödel will do nicely) and let us see what advantages such a declarative approach brings. The key aspect of my approach is that there should be a natural progression in a programming course (or succession of such courses) from, at first, pure specification, through more detailed consideration of what the system does when running a program (that is, the control aspect), finally finishing up with a detailed study of space and time complexity of various algorithms and the relevant aspects of implementations. It is very advantageous to be able to consider just the logic component at the very beginning. This is because, for many applications, the pro-

grammer's only interest is in having the computer carry out some task and the fewer details the programmer has to put into a program, the quicker and more efficiently the program can be written. I am making the assumption here, as is appropriate in many circumstances, that squeezing the last ounce of efficiency out of a program is not necessary and that the most costly component of the computerization process is the programmer's time. Naturally, this approach of concentrating at first only on the logic component of a program requires a declarative language and clearly won't work at all for an imperative language such as Modula-2. This, then, is the major reason I advocate starting first with a declarative language: the *primary* task of *all* programming is the statement of the logic of the task to be solved by the computer and the use of a declarative language makes it possible for student programmers to concentrate at first solely on this task.

By way of illustration, I outline briefly an approach to teaching *beginning* programmers using a declarative language. First, as we are concerned with teaching programming to beginners, we can take advantage of the fact that the applications considered can be carefully controlled. For example, list processing and querying databases make suitable starting points. Then, as for any programming task, the application is modelled by an intended interpretation in the logic underlying the programming language. Beginning programmers should be encouraged to write down the intended interpretation, even if only informally. Given that only simple applications are being considered, this is a feasible, and very instructive, task for the student to carry out.

Next the logic component of the program is written. With carefully chosen applications and appropriate programming language support, programmers can largely ignore control issues in the beginning. As usual, a key requirement is that the intended interpretation should be a model for the program and this should be checked, informally at least, by students. Once again, for simple applications, this is certainly feasible.

Finally, the program is run on various goals to check that everything is working correctly. Typically, it won't be and some debugging will be required. At this point, we can again take advantage of the fact that the language is declarative by exploiting a debugging technique known as declarative debugging [6], which was introduced and called algorithmic debugging by Ehud Shapiro. There isn't space here to explain this approach to debugging in detail, but the main idea is very simple: to detect the cause of missing answers and wrong answers, it is sufficient for the programmer to know the intended interpretation of the program. Since I have taken this as a key assumption of this entire approach to declarative programming, this isn't an unreasonable requirement. Essentially, the programmer presents the debugger with a symptom of a bug and the debugger asks a series of questions about the intended interpretation of the program, finally displaying the bug in the program to the programmer. Note that this approach doesn't handle other kinds of bugs such as infinite loops, floundering, or deadlock, which are procedural in nature and hence must be handled by other methods. Fortunately, as I explain below, for beginning programmers this turns out not to be too much of a limitation.

What programming language support do we need to make all this work? First,

the language must be declarative in as strong a sense as possible. Second, types and modules must be supported. These are not declarative features but, as is argued in [5], no modern language can be considered credible without them. Third, the programming system must have considerable autonomous control of the search. This is crucial if the student programmers are to be shielded from having to be concerned about control in the beginning. In its most general form, providing autonomous control to avoid infinite loops and such like is very difficult (in fact, undecidable). However, in the context of teaching beginning programmers, much can be done. For example, breadth-first search of the search tree (in the case of logic programming) can ameliorate the difficulty. This may not be very efficient, but for small programs is unlikely to be prohibitively expensive. Furthermore, simple control preprocessors, such as provided by Lee Naish for the NU-Prolog system [8], can autonomously generate sufficient control to avoid many problems for a wide range of programs. Finally, as I explained above, declarative debugging must be supported.

I believe the declarative approach to teaching programming outlined above has considerable merit and illustrates an important practical advantage of declarative programming.

3 Semantics

One problem which has plagued Computer Science over the years, and looks unlikely to be solved in the near future, is the gap between theory and practice. Nowhere is this gap more evident than with programming languages. Typical widely-used programming languages, such as Fortran, C, and Ada, are a nightmare for theoreticians. The semantics of such languages is extremely messy which means that, in practice, it is very difficult to reason, informally or formally, about programs in such languages. Nor is the problem confined to imperative languages – practical Prolog programs are generally only marginally more analyzable than C programs! This gap between the undeniably elegant theory of logic programming and the practice of typical large-scale Prolog programming is just too great and too embarrassing to ignore any longer.

This issue of designing programming languages for which programs have simple semantics is, quite simply, crucial. Until practical programming languages with simple semantics get into widespread use, programming will inevitably be a time-consuming and error-prone task. Having a simple semantics is not simply something nice for theoreticians. It is the key to many techniques, such as program analysis, program optimization, program synthesis, verification, and systematic program construction, which will eventually revolutionize the programming process.

This is another area where declarative programming can make an important practical contribution. Modern functional and logic programming languages, such as Haskell and Gödel, really do have simple semantics compared with the majority of widely-used languages. Of the two, Haskell is probably the cleanest in this regard, but the semantic difficulties that Gödel has are related to its non-deterministic nature which provides facilities and expressive power not possessed by Haskell. In any case, whatever the semantic difficulties of, say, Haskell or Gödel, there are nothing

compared to the semantic problems of Fortran, C, or even Prolog.

The complicated semantics of most widely-used programming languages partly manifests itself in current programming practice, which is very far from being ideal. Typically, programmers program at a low level (I regard C and Prolog, say, as low-level languages), they start from scratch for each application, and they have no tools to analyze, transform, or reason about their programs. The inevitable consequence of this is that programming takes much longer than it should and hence is unnecessarily expensive. We must move the programming process to a higher plane where programmers typically can employ substantial pieces of already written code, where concern about low-level implementation issues can be largely avoided, where there are tools available to effectively analyze and optimize programs, and where tools can allow a programmer to effectively reason about the correctness or otherwise of their programs.

All of these facilities are highly desirable and I see no way at all of achieving them with any of the current widely-used languages. The only class of languages which I see having any chance of providing these facilities is the class of declarative languages. If a program really is a theory in some logic, then there is much more chance of being able to analyze the program, transform the program, and so on. There is good evidence to support this claim from the logic programming community. For example, there is a considerable body of work on analysis and transformation of *pure* Prolog which really does work in practice and most of this work is directly applicable to the *entire* Gödel language. I don't mean to imply by this that carrying out such tasks is easy for a declarative language, only that for a declarative language many irrelevant difficulties (for example, assignment in C or assert/retract in Prolog[1]) are swept away and the real difficulties, which genuinely require attention, are exposed. In other words, with a declarative language, the problems are still difficult, but at least time isn't wasted trying to solve problems which shouldn't be there in the first place!

4 Programmer Productivity

Many computer scientists spend much of their time trying to discover efficient algorithms. Clearly, this effort is important as the difference for an application of, say, an $O(n \log n)$ versus an $O(n^2)$ algorithm may mean the difference between success and failure. This kind of argument has often been used to justify the low-level programming which typically takes place to implement efficient algorithms. However, there is another cost of programming which can easily be ignored by computer scientists and that is the cost of programmer time and the cost of maintaining and upgrading existing code. Often having the most efficient algorithm isn't so important; often a programmer would be very happy with a programming system which only required the problem be specified in some way and the system itself find a reasonably efficient algorithm. Nor should we ignore the fact that we want to make programming accessible to ordinary people, not just those with a computer science degree. Ordinary people generally aren't interested (and rightly so) in low-level programming details – they just want to express the problem in some reasonably congenial way and let the system get on with solving the problem.

[1] Assignment may very well be present at a low level in an implementation, but it has no place as a language construct because its semantics is too complicated; assert/retract was simply a mistake from the beginning.

I believe declarative programming has a big contribution to make in improving programmer productivity and making programming available to a wider range of people. Certainly, as long as we continue to program with imperative languages, we will make little progress in this regard. Since, in an imperative language, the logic and control are mixed up together, programmers have no choice but to be concerned about a lot of low-level detail. This adversely affects programmer productivity and precludes most ordinary people from becoming programmers. But once logic and control are split, as they are with declarative languages, the opportunity arises of taking the responsibility for producing the control component away from the programmer. Current declarative languages generally are still only weakly declarative and so programmers still have to specify at least part of the control component. But this situation is improving rapidly are we can look forward to practical languages which are reasonably close to being strongly declarative in the near future.

Having to deal only (or mostly) with the logic component simplifies many things for the programmer. First, (the logic component of) a declarative program is generally easier to write and to understand than a corresponding imperative program. Second, a declarative program is also easier to reason about and to transform, as much current research in functional and logic programming shows. Finally, it sets us on the right road to the ultimate goal of synthesizing efficient programs from specifications.

5 Meta-Programming

The essential characteristic of meta-programming is that a meta-program is a program which uses another program (the object program) as data. Meta-programming techniques underlie many of the applications of logic programming. For example, knowledge base systems consist of a number of knowledge bases (the object programs), which are manipulated by interpreters and assimilators (the meta-programs). Other important kinds of software, such as debuggers, compilers, and program transformers, are meta-programs. Thus meta-programming includes a large and important class of programming tasks.

It is rather surprising then that programming languages have traditionally provided very little support for meta-programming. Lisp made much of the fact that data and programs were the same, but it turned out that this didn't really provide much assistance for large-scale meta-programming. Modern functional languages appear to me to effectively ignore this whole class of applications since they provide little direct support. Only logic programming languages have taken meta-programming seriously, but unfortunately virtually all follow the Prolog approach to meta-programming which is seriously flawed. (The argument in support of this claim is given in [5]).

In fact, as the Gödel language shows, providing full-scale meta-programming facilities is a substantial task, both in design and implementation. The key idea is to

introduce an abstract data type for the type of a term representing an object program, and then to provide a large number of useful operations on this type. The relevant Gödel system modules, **Syntax** and **Programs**, provide such facilities and contain over 150 predicates. The term representing an object program is obtained from a rather complex ground representation [5], the complexity of which is almost completely hidden from the programmer by the abstractness of the data type. When one sees how much Gödel provides in this regard, it is clear how weak are the meta-programming facilities of functional and imperative languages, all of which should be following the same basic idea as Gödel.

One key property of the Gödel approach to meta-programming is that it is declarative, in contrast to the approach of Prolog. This declarativeness can be exploited to provide significant practical advantages. For example, the declarative approach to meta-programming makes possible advanced software engineering tools such as compiler-generators. To obtain a compiler-generator, one must have an effective self-applicable partial evaluator. The partial evaluator is then partially evaluated with respect to itself as input to produce a compiler-generator. (This exploits the third Futamura projection.) The key to self-applicability is declarativeness of the partial evaluator. If the partial evaluator is sufficiently declarative, then effective self-application is possible, as the SAGE partial evaluator [3] shows. If the partial evaluator is not sufficiently declarative, then effective self-application is impossible, as much experience with Prolog has shown.

Now why should a programmer be interested in having at hand a partial evaluator such as SAGE and the compiler-generator it provides? First, the partial evaluator is a tool that is essential in carrying forward the move towards higher-level programming. Writing declarative programs inevitably introduces inefficiencies which have to be transformed or compiled away. Partial evaluation is a technique which has proved to be successful in this regard. By way of illustration of this, the use of abstract data types (which is not purely a declarative concept, but nonetheless an important way raising the level of programming) in a logic programming language restricts the possibilities for clause indexing. However, partial evaluation can easily push structures back into the heads of clauses so that the crucial ability to index is regained. In general terms, partial evaluation is a key tool which allows programmers to program declaratively and at a high level, and yet still retain much of the efficiency of low-level imperative programming.

But what about a compiler-generator? Strictly speaking, a compiler-generator is redundant since everything it can achieve can also be achieved with the partial evaluator from which it was derived. However, there is a significant practical advantage of having a compiler-generator which is that it can greatly reduce program development time. For example, suppose a programmer wants to write an interpreter for some specialized proof procedure for a logic programming language. The interpreter is, of course, a meta-program and it can be given as input to a compiler-generator. The result is a specialized version of the partial evaluator which is, in effect, a compiler corresponding to this interpreter. When an object program comes along, it can be given as input to this compiler and the result is a specialized version of the original interpreter, where the specialization is with respect to the object program. Now the final result can be achieved by the partial evaluator alone – simply partially evaluate the interpreter with respect to the object program. But this requires partially evaluating the interpreter over and over again for each object program. By using the compiler-generator, we specialize what we can of the interpreter just once and then complete in an incremental way the specialization by giving the resulting compiler the object program as input. For an interpreter which is going to be run on many object programs, the approach using the compiler-generator can save a considerable amount of time compared with the approach using the partial evaluator directly. Now recall, what was the key to all of this being possible? It was that the partial evaluator be (sufficiently) declarative!

I believe we have hardly begun to scratch the surface of what will be possible through declarative meta-programming. I hope researchers producing analysis and transformation tools will take the ideas of declarative meta-programming more seriously, and that designers of declarative languages will make sure that new languages have adequate facilities for the important class of meta-programming applications.

6 Parallelism

Parallelism is currently an area of intense research activity in Computer Science. This is as it should be – many practical problems require huge amounts of computing for their solution and parallelism provides an obvious way of harnessing the computing power required. While many difficult problems associated with parallel computer architectures remain to be solved, I concentrate here on the software problems associated with programming parallel computers and the possible contribution that declarative programming might make towards solving these problems.

It is well known that parallel computers are hard to program. Widely-available programming languages which are used on parallel machines are usually ill-suited to the task. For example, older languages such as Fortran have required retro-fitting of parallel constructs to enable them to be used effectively on parallel computers. Furthermore, on top of all the other difficulties of programming, on a parallel computer a programmer is also likely to have to cope with deadlock, load balancing, and so on. One, more modern, class of programming languages that can be used on parallel computers are the concurrent languages – those which have explicit facilities in the languages to express concurrency, communication, and synchronization. These clearly have much potential, but I will concentrate instead of the class of declarative languages for which the exploitation of parallelism is implicit. In other words, whether a declarative program is being run on a sequential or parallel computer is transparent to the programmer. The only discernible difference should be that programs run much faster on a parallel machine!

Declarative languages are well-suited for programming parallel computers. This is partly because there is generally much implicit parallelism in declarative programs. For example, a functional programming can be run in parallel by applying several reductions at the same time and a logic program can be run in parallel by exploiting (implicit) And-parallelism, Or-parallelism, or even both together. The declarative-

ness is crucial here as the more declarative the programming language, the more implicit parallelism there is likely to be. This is illustrated by a considerable amount of research on parallelizing Prolog in the logic programming community, which has discovered that the non-logical features of Prolog, such as `var`, `nonvar`, `assert`, and `retract`, greatly inhibit the possible exploitation of parallelism. The lesson here is clear: the more declarative we can make a programming language, the greater will be the amount of implicit parallelism that can be exploited. There is another aspect to the problems caused by the non-logical features of Prolog – not only do they restrict the amount of parallelism that can be exploited, but their parallel implementation takes an inordinate amount of effort (especially if one wants to preserve Prolog's sequential semantics). Once again the lesson is clear: the more declarative we can make a programming language, the easier will be its parallel implementation.

Implicit exploitation of parallelism in a declarative program is very convenient for the programmer who, consequently, has no further difficulties beyond those which are present in programming a sequential computer. But the system itself must now be clever enough to find a way of running the program in the most efficient manner. This is a very hard task in general and the subject of much current research, but the results so far are rather encouraging and suggest that it will indeed be possible in the near future for programmers to routinely run declarative programs on a parallel computer in a more or less transparent way.

I conclude this section with a discussion of the problems of debugging programs which are to run on a parallel computer. Clearly, if transparency is to be maintained, it must be possible for programmers to be able to debug their programs without any particular knowledge of the way the system ran them. For the large class of bugs consisting of wrong or missing answers, declarative debugging comes to the rescue. The reason for this is as follows. A declarative debugger typically is either pulling apart the statements in a program to try to find the bug or else is running goals with respect to the program and checking the results of running these goals with the programmer to see if they are correct or not. Now whether these goals are run sequentially or in parallel makes no difference at all. The upshot of this is that declarative programming works just as well on a parallel computer as on a sequential one. In particular, all the programmer needs to know in either case is the intended interpretation of the program.

7 Discussion

Having discussed various specific aspects of the practical advantages of declarative programming, I now turn to some general remarks.

The first is that it is hard to escape the conclusion that many researchers in logic programming, while claiming to be using declarative programming, do not actually take declarativeness all that seriously. There is good evidence for this claim. For example, in spite of the fact that it has been clear for many years that Prolog is not really credible as a declarative language, it is still by far the dominant logic programming language. The *core* of Prolog (so-called "pure" Prolog) is declarative,

but large-scale practical Prolog programs typically use many non-logical facilities and these features destroy the declarativeness of such programs in a rather dramatic way. Furthermore, while many extensions and variations of Prolog have been introduced and studied by the logic programming community over the last 15 years, including concurrent languages, constraint languages, and higher order languages, these languages have been essentially built on top of Prolog and therefore have inherit many of Prolog's semantic problems. For example, most of these languages use Prolog's approach to meta-programming. It seems to me that if logic programmers generally had taken declarativeness seriously, this situation would have been rectified many years ago because as the Gödel language shows the argument that the non-logical features are needed to make logic programming languages practical is simply false. What distinguishes logic programming most from many other approaches to computing and what promises to make it succeed where many other approaches have failed is its declarativeness. Logic programming has so far not succeeded in making the sort of impact in the world of computing that many people, including myself, expected. There are various reasons for this, but I believe very high on the list is the field's failure to capitalize on its most important asset – its declarative nature.

I mentioned briefly earlier that the fields of functional and logic programming should be combined. If one forgets for a moment the history of how these fields came into existence, it seems very curious that they are so separated. Apart from periodic bursts of interest in producing a combined functional and logic programming language (mainly, it seems, by logic programmers), the two fields and the researchers in the fields rarely interact with one another. And yet both fields are trying to solve the same problems, both have the same declarative approach, and the differences between recent languages in each of the fields is not great. For example, a comparison between Haskell and Goedel reveals that, while there are many small differences, there are also many similarities in philosophy and facilities. The more important differences are that the logic of Haskell is λ-calculus, while the logic of Gödel is many-sorted polymorphic first order logic; Haskell handles higher order functions, but Gödel does not; Gödel provides considerable meta-programming facilities, but Haskell does not; and Gödel provides (explicit) non-determinism, but Haskell does not. I think it is time that the attempt to produce languages which are elegant combinations of functional and logic programming ideas is revived (and, in fact, there is evidence that this is happening at the moment [4]). The higher-order logic language λ-Prolog [7] is sure to provide a number of important ideas for this synthesis. An elegant functional/logic programming language looks to me to be entirely feasible and, when achieved, would provide us with a language which embodied the best ideas of both fields, including types, modules, higher order facilities, meta-programming facilities, and non-determinism. Ten years ago, David Turner wrote [9]:

> It would be very desirable if we could find some more general system of assertional programming, of which both functional and logic programming in their present forms could be exhibited as special cases.

Over the last decade, there have been several important attempts carry out (at least part of) this programme, but, whatever the reasons, today there is still no widely-

used language which could be regarded as an elegant synthesis of the best ideas of functional and logic programming. It's time we remedied this deficiency.

Perhaps, amidst all the euphoria of the advantages of declarative programming, it would be wise to conclude with some remarks on the limitations of this approach. The forms of declarative programming based on standard logics, such as first order logic or higher order logic, do not cope at all well with the temporal aspects of many applications. The reason is that the model theory of such logics is essentially "static". For applications in which there is much interaction with users or processes of various kinds (for example, industrial processes), such declarative languages do not cope so well and often programmers have to resort to rather *ad hoc* techniques to achieve the desired result. For example, in Prolog, a programmer can take advantage of the fact that subgoals are solved in a left to right order. (Actually, most imperative approaches have exactly the same kinds of difficulties, but I don't think this absolves the declarative programming community from finding good solutions.) We need new "declarative" formalisms which can cope better with the temporal aspects of applications. Obviously, the currently existing temporal logics are a good starting point, but it seems we are still far from having a formalism which generalizes currently existing declarative ones by including temporal facilities in an elegant way. Much more attention needs to be paid to this important aspect of the applications of declarative programming.

8 Conclusions

By way of conclusion, I summarize now the main points made in this paper.

- The key idea of declarative programming is that a program is a theory (in some suitable logic) and that computation is deduction from the theory.
- Declarative programming includes logic programming and functional programming, and intersects significantly with other research areas.
- The primary task of all programming is the statement of the logic of the task to be solved by the computer.
- Ultimately, programmers are interested in computing truth in the intended interpretation.
- Declarative debugging is a key technique needed for declarative programming.
- Having a simple semantics is the key to the application of many techniques, such as program analysis, program optimization, program synthesis, verification, and systematic construction.
- Declarative programming has a big contribution to make in improving programmer productivity and making programming available to a wider range of people.
- The declarative approach to meta-programming makes possible advanced software engineering tools such as compiler-generators.

- The more declarative we can make a programming language, the greater will be the amount of implicit parallelism that can be exploited.
- The more declarative we can make a programming language, the easier will be its parallel implementation.
- The fields of functional and logic programming should be combined.
- We need new "declarative" formalisms which can cope better with the temporal aspects of applications.

References

[1] J.H. Fasel *et al.* Special issue on the functional programming language Haskell. *ACM SIGPLAN Notices*, 27(5), 1992.

[2] A.J. Field and P.G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[3] C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[4] M. Hanus. The integration of functions into logic programming: A survey. Technical Report MPI-I-94-201, Max-Planck-Institut Für Informatik, 1994. To appear in The Journal of Logic Programming.

[5] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994. Logic Programming Series.

[6] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[7] G. Nadathur and D. Miller. An overview of λ-Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, pages 810–827. MIT Press, 1988.

[8] J. A. Thom and J. Zobel. Nu-prolog reference manual, version 1.3. Technical report, Machine Intelligence Project, Department of Computer Science, University of Melbourne, 1988.

[9] D.A. Turner. Programs as executable specifications. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice-Hall, 1985.

Analysis and Refinement of Constraint Answer Sets in a Planning System

Michael Nitsche
GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany
E-mail: micha@first.gmd.de

Abstract

The background of this paper is an algorithm to solve the can-build problem — a subtask of Materials Requirements Planning — based on Constraint Logic Programming. The most general solution returned by the algorithm is expressed in terms of constraints and cannot directly be used in a planning tool. This paper investigates how the user can be supported to understand, analyse and refine a constraint answer set maintaining his domain and technical terms. An interactive decision support mechanism has been provided that allows to add management information and intuitive knowledge by specifying constraints implicitly via a command interpreter.

1 Introduction

Production planning strategies are extremely difficult and complex, since at any one time, it cannot be told exactly how the demands of the market are going to develop in the future. *Materials Requirements Planning (MRP)* is the process by which a company decides how to organise its purchases from other firms and the manufacturing of final products, their components and subcomponents in order to meet a given production plan.

The *can-build problem* is a subtask of MRP. Solving the can-build problem means establishing an executable production plan when restrictions on the availability of resources occur. The reasons for such restrictions may be quite varied — parts are not delivered in the required time or quality, production capacity is limited, the manufacture of subcomponents cannot be completed on schedule for various reasons etc.

Owing to the large amount of data that has to be considered in successfully scheduling any kind of production and in view of the dynamic nature of the planning process, powerful tool support is needed. During the development of a commercial MRP system, we have investigated the feasibility of solving can-build problems using Constraint Logic Programming [5]. This means applying an advanced logic programming technique to a real-world problem and — to our knowledge — has not been done before. Using CLP($\mathcal{R}$) [1] as implementation language, we have designed an algorithm that presents the solution to the particular can-build problem considered in most general terms.

MRP systems are used by managers and planners, who cannot be expected to have detailed knowledge of CLP. This paper sets out to investigate how the user can be supported to analyse and refine a solution that was returned by the algorithm, i.e. how his requirements can be expressed in terms he is familiar with and still be translated into constraints understandable by the system.

We have provided an interactive decision support mechanism, whose higher-level character is maintained by a command interpreter. It allows the user to view the constraint answer set under a certain projection. By adding additional constraints (specified by parameterised commands) according to his preferences, he is able to refine the most general solution by further instantiation. In this way, management information and intuitive knowledge can be added after the system has determined what is basically feasible.

In our paper, we assume familiarity with Constraint Logic Programming. No previous knowledge of MRP and other engineering details is required, since all the background needed is covered in the next section, which also explains the basic idea behind the can-build algorithm. In Section 3, we explain how the user is supported in analysing and refining the constraint set returned by the algorithm. The command language is illustrated at some examples. Section 4 contains a summary. In Appendix A we give a detailed outline of the can-build algorithm, in Appendix B we show two example products.

2 Technical Background

Various MRP systems are used in different branches of industry. There are hardly any publications available about particular MRP strategies, which are normally adapted to the type of products and the technology used by a company. We have therefore confined our investigations to one particular system — Hewlett-Packard's I-MRP tool (for Interactive Materials Requirements Planning), which was developed at Hewlett-Packard Laboratories, Bristol, as a decision support system for use in the planning process [3, 4]. All technical details explained in Section 2.1 are defined by this tool.

2.1 Planning Data

In order to be able to supply the user with the information needed, the I-MRP system has access to various basic data, which is stored in external databases. Basically, there are two types of data — product-related data and planning-related data.

Product-related data is the data providing information on how products are composed of their subcomponents. This database is called *bill of materials (BOM)*. The BOM for each product can be represented as a directed, acyclic graph, with the nodes representing subcomponents and parts contained in the product, and the arcs representing the parent-child relation between a part and its subparts. The arcs are labeled with the number of parts contained in the parent; this figure is called *quantity-per*. Nodes that have no arcs originating from them represent *purchased parts*, which are the smallest units of a product. All other parts in the BOM are called *fabricated parts*.

If the construction of some part (i.e., the BOM) is modified for technological reasons, the parent-child relation changes over time. Such updates of the BOM are called *engineering changes*. Engineering changes are important if we are to represent the manufacturing process properly. The smallest planning unit of the I-MRP system is the working day. Hence, all time-varying quantities (such as quantity-pers) are represented as intervals, which is the basic data structure used by the can-build algorithm. An *interval* is a list of pairs d(<day>,<number>) where <day> is an integer and <number> a real number. Day numbers are always instantiated, being considered relative to the current working day, which is day 1 of the planning period. For the manipulation of intervals we have implemented a small interval algebra. The particular meaning of an interval depends on the context in which it is used.

Other information stored in the BOM is for each fabricated part a *cycle time* (the number of days needed to manufacture the part from its children), *yield* and *shrinkage* — the latter numbers represent percentages of parts that can actually be used (in view of quality requirements etc.) or that is expected to get lost after manufacturing, respectively.

While product-related data is merely engineering information and not dependent on what is actually produced, planning-related data basically represents production and demand figures. This data comprises the *build plan*, which defines how many units of each product have to be manufactured at what time, the *purchased part supply plan* specifying when and how many units of those parts have to be purchased which the company does not produce itself, and *on-hand balance* and *intermediate stock*, which are the numbers of purchased and fabricated parts respectively that are currently kept in stock. Again, it is essential for the model that the demand figures in the build plan and the purchased part supply plan are not just numbers, but functions over time represented as intervals.

From the basic data explicitly stored in the various databases, the system is able to derive other data which might be of interest to the user. Our algorithm uses an *availability profile* for each part to indicate how many items of the part become available each day. For purchased parts, this is the sum of the on-hand balance (available on day 1) and the purchased part supply plan; for fabricated parts, it is merely the intermediate stock (again, available on day 1).

2.2 The Can-Build Problem and Algorithm

Can-build questions are not normally asked in the regular planning cycle. However, they become very important as soon as we encounter restrictions on the availability of resources (e.g. materials, subcomponents, capacity) making it impossible for us to entirely fulfil all our manufacturing plans. In some sense, this means reversing the usual MRP process and asking: "What can be built with the things we have got?" instead of "What things are needed to build what we want?"

The can-build problem is nontrivial for various reasons. Many parts and subcomponents can be used in several products (and in more than one place in a single product). They are called *multiple-usage parts*. This introduces nondeterminism and prevents the use of a "pure" bottom-up strategy. Engineering changes introduce a

time-dependent dimension, since the output from the same amount of resources varies in such cases from day to day. With intermediate stock, the demand for a part is not propagated proportionally to its children.

In order to handle the complications mentioned above, the can-build problem was broken down into smaller subproblems. Taking the working day as the smallest planning unit, we have designed an incremental algorithm which consecutively takes into account all constraints that are valid on a single day.

For each day, we run a modified bottom-up algorithm. The number of items that can be built on that day is calculated in a bottom-up manner as the sum of the number of items in stock and the number of items that can be manufactured from the children of the part under consideration. Obviously, this requires solving the can-build problem for all the children first. Whenever nondeterminism occurs, because there are several uses for the same part, constrained variables are introduced describing how many items are used immediately, and how many are left for later use. Thus, the decision on how to split the availability of a part is deferred until the end of the algorithm. An informal outline of our can-build algorithm is given in Appendix A.

The solution for the current can-build problem is represented by a *can-build profile*. The number of parts that can be built on a particular day is called the *can-build number*. For each product or part considered, the user is given a usually non-ground interval with constraints imposed on the variables to indicate how many items of this product can be built at what time. The can-build profile is always the most general solution to the particular can-build problem, i.e. every ground instance of the can-build profile is a solution to the current can-build problem, and vice versa. Obviously, this has to include the possibility that nothing is built at all. It also includes that everything that can be built on a certain day may be left for later use, i.e. the output of the production can be accumulated.[1]

A typical can-build profile for some product generated by the algorithm presented in Appendix A might look like

[d(1,N1),d(2,N2),d(3,N3),d(4,N4)] with the constraints
$0 \leq N1$, $0 \leq N2$, $0 \leq N3$, $0 \leq N4$,
$N1 \leq 5$, $N1 + N2 \leq 15$,
$2*N1 + 2*N2 + N3 \leq 40$, $2*N1 + 2*N2 + N3 + N4 \leq 60$

Each of the following intervals is an instance of this can-build profile

[d(1,0),d(2,0),d(3,0),d(4,0)]
[d(1,5),d(2,10),d(3,10),d(4,20)]
[d(1,0),d(2,0),d(3,0),d(4,60)]
[d(1,5),d(2,5),d(3,5),d(4,N4)] with $0 \leq N4$, $N4 \leq 35$

[1] The conditions that a particular demand has to be satisfied or that a certain minimum of a product has to be manufactured, are additional constraints imposed by the user. They are not represented in the BOM or the availability profiles and thus not considered at this stage.

3 Constraint Analysis

3.1 General Aspects

As one can see from the example in the previous section, the most general solution produced by our can-build algorithm is normally not of much interest to the user since it does not tell much about what is really possible. The user needs a ground can-build profile for each part which can actually be executed as a production plan and which meets all his additional requirements.

We therefore have to find a way to assign a value to each solution instance in order to be able to compare their feasibility. For most practical cases, it is not possible to provide in advance all the information that the algorithm might need to uniquely identify one ground answer as the best one. This is because we cannot *tell in* advance, from the restrictions on the availability of purchased parts, what kind of constraints will be imposed on the top-level products. Another reason is that decisions on 'compromises' between several products sharing common subcomponents are taken by human beings (the manufacturing managers and planners) and cannot easily be expressed in clear figures. However, the most general solution provides the starting-point for a further analysis of the constraints imposed on it.

In our implementation of the can-build problem, analysis and refinement of constraints are realised by an interactive command interpreter (written in CLP($\mathcal{R}$), as well). This allows the user to extract whatever information he needs from the general solution and introduce new constraints to specify the particular solution he is aiming for. But first we want to go into some detail about redundancies contained in the original solution and how they can be removed.

3.2 Removing Redundant Constraints

Normally thousands of constraints are accumulated by the CLP($\mathcal{R}$) system when executing the can-build algorithm. These constraints involve variables representing the can-build numbers for certain products on certain days, but they also contain temporary variables which are introduced by the simplex algorithm during program execution. Such temporary variables cannot be related explicitly to any variable used in the program; they are of no meaning to the user.

The task of eliminating temporary variables and redundant constraints is left to the output module of CLP($\mathcal{R}$). Ideally, the output constraints will only involve target variables (i.e. those variables contained in the query) and be free of redundancy, but this is not always possible. In general, eliminating redundancies from a set of linear arithmetic constraints is a nontrivial problem [6].

Since certain properties hold for the constraint set returned by the can-build algorithm, we can remove most redundancies using a quite simple algorithm. All can-build numbers for each product must have values greater than or equal to 0; all other constraints are of the format $\sum a_i N_i \geq 0$ or $\sum a_j N_j \leq c$ with N_i and N_j being can-build numbers and a_i, a_j and c being numeric constants greater than 0. Thus, two types of redundant constraints can easily be removed.

- Constraints of the format $\sum a_i N_i \geq 0$ can be removed directly because they are implied by the ≥ 0 constraints.
- A constraint $\sum a_i N_i \leq c$ is implied by a similar one $\sum b_i N_i \leq d$ if both sums involve the same can-build numbers N_i and $da_i \leq cb_i$ for all i.
  In other words, $\sum cb_i N_i \leq cd$ implies $\sum da_i N_i \leq cd$ if $da_i N_i \leq cb_i N_i$ for all i.[2]

The algorithm checks all constraints (except the ≥ 0 constraints) pairwise if one of them implies the other and removes the latter one where appropriate. This results in a significantly smaller constraint set. The ≥ 0 constraints need not be displayed at all. But even a constraint set of this sort is of little meaning to the user. We therefore have to find a way of comparing different ground can-build profiles that are contained in the most general solution.

3.3 Comparing Ground Solutions

The most general solution produced by our can-build algorithm does in fact describe a set of ground can-build profiles. Each of these ground solutions satisfies all the restrictions imposed on the availability of parts and the dependencies between them. The user of the program now wants to find the ground instance that best meets his requirements. His decision might be based on additional constraints that he wants to be satisfied, e.g. one or more of the following

- On a particular day (or for a time period) the manufactured number of items of a product shall be fixed.
- A minimum production is to be set for some parts or products.
- All products are to be built according to their demand. If this is not possible, a priority is given to each product indicating its importance.
- All available resources shall go into the production of one particular product, i.e. this product is to be maximised first.
- Each product has a certain value (or yields a certain profit), and the total manufactured value shall be maximised.
- Of a certain product, as many as possible are needed as early as possible (even if a later output would be higher because of engineering changes).

The main reason for implementing the command interpreter was to maintain a higher-level character of the constraint analysis. The user cannot be bothered to express his requirements in terms of CLP($\mathcal{R}$). Considering the examples above, there are certain types of constraints the user might want to be satisfied. Such additional constraints can be classified in two categories.

[2] To enable us to check two constraints of this type for redundancy, we might allow some of the a_i or b_i to be 0.

Demand constraints are constraints that can be written explicitly in terms of one or several can-build numbers (for one or several products). Like the first three examples above, they normally specify that a certain demand is to be satisfied, either exactly or as a lower bound. Demand constraints might be inconsistent with the constraints that are already imposed on the current can-build profiles for some or all of the products under consideration.

Optimisation constraints allow the user to specify an optimisation criterion that is to be satisfied. Like the last three above, they cannot be written explicitly in terms of can-build numbers, but rather define an implicit target function in terms of can-build numbers whose value is to be maximised. Introducing an optimisation constraint means defining an additional constraint that forces the value of the target function to be equal to its maximum.

In order to motivate the commands for specifying optimisation criteria, we define a partial ordering on the set of all ground can-build solutions to a particular problem by the following two heuristic constraints:

- Consider two ground can-build profiles P1 and P2 for the same product. Let the can-build number of P1 always be greater than or equal to the can-build number of P2 for the same day. Then P1 is at least as good as P2.[3]
- Consider two other ground can-build profiles P3 and P4 for the same product. Let the accumulated can-build total of P3 for each day (i.e. the sum of all can-build numbers from the beginning of the interval to that day) be greater than or equal to the accumulated total of P4 for the same day. Then P3 is at least as good as P4.[4]

Two ground can-build profiles are not comparable and have no common upper bound in the ordering defined above if (and only if) engineering changes allow a better output of the final product from the same resources later in the planning cycle. The decision as to whether 500 items of a product today are of greater value to the company than 1200 items of the same product next week (using the same resources) is nontrivial and has to be taken by the user of the program. What we have done is to provide commands for introducing either type of optimisation constraint — maximise the total number of products built, or build as many products as possible as early as possible — on the current solution. The user might then try them in a "what if" manner to decide what best satisfies his preferences.

Unlike demand constraints, optimisation constraints are always consistent with the current constraint set. If the can-build profiles under consideration are "sufficiently ground", the range of the target function might be one single value only. No further constraints have to be introduced in this case.

[3] This statement is not valid if we wish to avoid overproduction.

[4] This does not take into account the possibility that we might not be able to sell the product immediately and would like to minimise the cost of storing manufactured products.

3.4 The Command Interpreter

We explain now the command interpreter prototype that allows a constraint analysis and refinement expressed in technical terms. In an application program, this has to be realised by a system-dependent interface. For this reason, no particular efforts have been made yet to design a very user-friendly syntax. However, the commands are in terms of production planning rather than Constraint Logic Programming.

The command interpreter is entered once the most general solution to the current can-build problem has been returned by the system, allowing the user to implicitly specify any of the additional constraints mentioned in the previous section. After each command, the specified constraints (if any) are added to the constraint set. The modified constraint set is displayed after removal of redundancies, and the user is prompted to give the next command.

There is one multipurpose command for introducing new constraints and for viewing existing ones under a certain projection.[5] The general format of this command is

`<part specification>:<day specification>:<command>.`

The first part `<part specification>` of the command defines the subset of parts to which the command is applied. This may be the keyword `all`, the name of a part, or a list of part names. The order of the parts defines their decreasing priority, which is important if demand constraints cannot be satisfied for all the parts at the same time.

The second part `<day specification>` specifies the time period for which the command is to be executed. It may be a number for a single day, a time period (two numbers separated by a colon), or the whole planning interval (specified by the keyword `total`.

The third part `<command>` describes the command itself. Each command is always applied to the products specified previously for the defined time period. Demand constraints are specified by `equal(<number>)`, `min(<number>)` and `demand`, which fixes the number of items produced, sets a minimum number of items to be produced, or constrains production to satisfy the demand, i.e. build plan (as stored in the database).

Optimisation constraints are specified by `max` and `early`, which maximise the total manufactured number of items, or incrementally maximise the number of items produced on each day ("as many as possible as early as possible"), respectively. A `<list of values>` can be passed as an argument to both commands, in which case each part is assigned a value according to the list, this value being maximised rather than just the number of items.

If `show` is given as a command, no new constraints are introduced, but a projection of the constraints on the parts and the time period specified is produced and displayed.

Other commands allow the user to store the current constraint set under a name, restore a previously stored or the original constraint set, undo an arbitrary number of commands, or call any CLP($\mathcal{R}$) goal. These facilities enable the user to analyse

[5] See the next section for examples.

the most general solution returned by the can-build algorithm in a "what if" manner, until a ground solution is found that best satisfies his requirements.

He might first introduce some constraints that are to be satisfied (such as minimum production figures). This modified constraint set — representing the 'basic' requirements — can be stored and retrieved at any time. Afterwards the user may "play around" specifying further constraints, removing them or returning to his basic constraint set.

3.5 Examples

In this section we illustrate the use of the command interpreter at a few examples. We also give the additional constraints introduced by the system, which are not visible explicitly for the user. If the can-build algorithm is called for the products P1 and P2 shown in Appendix B, the following most general solution is returned:

P1: [d(1,N1),d(2,N2),d(3,N3)]
P2: [d(1,N4),d(2,N5)] with the constraints[6]
$0 \leq N1, 0 \leq N2, 0 \leq N3, 0 \leq N4, 0 \leq N5$,[7]
$N1 + N2 + N3 \leq 20$, $N4 + N5 \leq 15$,
$2*N1 + N4 \leq 10$,
$2*N1 + 2*N2 + N3 + N4 + N5 \leq 25$

The whole output of P1 and P2 is maximised using the command

`all:total:max.`

which introduces the constraint

$N1 + N2 + N3 + N4 + N5 = \max(N1 + N2 + N3 + N4 + N5)$.

The result is

P1: [d(1,0),d(2,0),d(3,N3)]
P2: [d(1,N4),d(2,25–N3–N4)] with the constraints
$10 \leq N3 \leq 20$, $N4 \leq 10$, $N3 + N4 \leq 25$

Any ground intervals for P1 and P2 satisfying these constraints yield a maximum total number of items produced. To find out how much of P2 can be produced, the following command is used

`p2:total:show.`

This produces a projection of the constraints on the can-build profile for P2:

P2: [d(1,N4),d(2,N5)] with
$5 \leq N4 + N5 \leq 15$, $N4 \leq 10$

[6] The algorithm generates 12 constraints (not considering the ≥ 0 constraints), four of which are non-redundant.

[7] The ≥ 0 constraints for all non-ground can-build numbers are part of any solution. We do not mention them explicitly in the following examples.

The user might now want to define a total production for P1 first using

`p1:total:equal(12).`

which introduces N3 = 12 and gives

P1: [d(1,0),d(2,0),d(3,12)]
P2: [d(1,N4),d(2,13–N4)] with the constraint
$N4 \leq 10$

Producing everything of P2 as early as possible is specified by

`p2:total:early.`

This introduces N4 = max(N4) and results in the ground can-build profiles

P1: [d(1,0),d(2,0),d(3,12)]
P2: [d(1,10),d(2,3)]

If instead of `p1:total:equal(12)` above we had written `p1:total:max` and then `p2:total:early`, the result would have been

P1: [d(1,0),d(2,0),d(3,20)]
P2: [d(1,5),d(2,0)]

If the command

`all:total:early.`

is given at the top level, the following constraints are introduced subsequentially on the most general solution

$N1 + N4 = \max(N1 + N4)$
$N2 + N5 = \max(N2 + N5)$
$N3 = \max(N3)$

This gives immediately the ground solution

P1: [d(1,0),d(2,5),d(3,0)]
P2: [d(1,10),d(2,5)]

Note that in the latter case we get only a total output of 20 items instead of 25 above.

4 Summary

We have presented a tool that enables users of a planning system to analyse and refine constraint answer sets (generated by a CLP language) maintaining their own technical terms. An interactive decision support mechanism has been provided that allows to add management information and intuitive knowledge. The necessary constraints are specified implicitly via a command interpreter. The analysis is based on an algorithm to solve the can-build problem using CLP($\mathcal{R}$). It is capable of handling some advanced features of Materials Requirements Planning, such as multiple products and engineering changes.

Appendix

A Can-Build Algorithm

Here we present an outline of our can-build algorithm that considers a list of products to be manufactured. The recursive step that finds what is possible according to the children of a part is step (3b). The merge procedure mentioned there works incrementally over the working days of the planning period in order to represent engineering changes (i.e. different quantity-pers for the same child on different days) properly. The nondeterministic splitting of resources is done in (4b). More details about the algorithm are given in [7], some of the difficulties inherent in several of the steps are explained in [8].

Algorithm 1: Can-Build Algorithm for a List of Top-Level Products

(1) Generate the list of multiple-usage parts and use it throughout the algorithm. Initialise the list of current availability profiles with the empty list.

(2) Take now subsequently every product in the list of top-level products as the current part C and continue with (3) passing the same list of current availability profiles around (modifying it according to the algorithm).

(3) If C is not a multiple-usage part, do

  (a) Get the availability profile for C from the database and generate a can-build profile from it.

  (b) Solve the can-build problem for all the direct children of C and merge the can-build profiles obtained into a single one for C. Adjust the result according to cycle time, yield and shrinkage for C.

  (c) Add the results of (3a) and (3b) to give the total can-build profile for C.

(4) If C is a multiple-usage part, do

  (a) If C has not been considered before, calculate its total can-build profile according to (3a)–(3c) above and take this as its current availability profile.

  (b) Split the current availability profile for C into a can-build profile (indicating how many items of the part are used at this point) and a new availability profile (indicating what is left for later use).

B Example Products

In this section, we present two simple products P1 and P2 sharing a common subcomponent P3. The parent-child relation together with the quantity-pers are shown in Figure 1. Note that there is an engineering change on day 3. All other quantity-pers remain unchanged throughout the planning period. For reasons of simplicity, we have

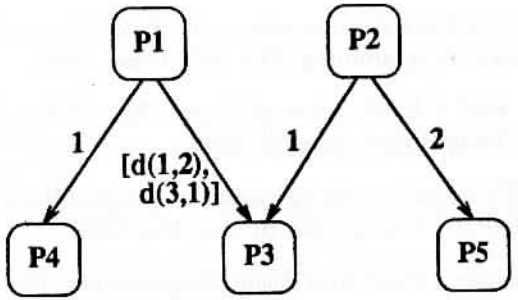

Figure 1: BOM for products P1 and P2

just given the respective numbers instead of writing [d(1,<number>)]. Complicating factors, such as cycle time, yield and shrinkage, are not involved.

The following table shows the availability profiles for all fabricated parts.

Part	Availability profile
P3	[d(1,10),d(2,15)]
P4	[d(1,20)]
P5	[d(1,30)]

The most general solution returned by the can-build algorithm for these products is shown in Section 3.5.

Acknowledgements

The research presented in this paper was conducted in the form of an individual project during postgraduate studies at Imperial College, London. I would like to thank my supervisors, Frank McCabe from the Department of Computing at Imperial College London and David Chan and Steve Owen from Hewlett-Packard Labs in Bristol for many fruitful discussions.

My studies were funded mainly by the Cusanuswerk — a foundation providing student grants run by the Roman Catholic Church in Germany — and partly by the British Chamber of Commerce in Germany. Further work is performed as part of the WISPRO project supported by the German Ministry of Research and Technology (BMFT).

We also wish to thank anonymous referees for their useful comments.

References

[1] N. Heintze et al.: *The* CLP($\mathcal{R}$) *Programmer's Manual, Version 1.2.* IBM Technical Report, T. J. Watson Research Center, Yorktown Heights, USA, September 1992.

[2] N. Heintze et al.: *On Meta-Programming in* CLP($\mathcal{R}$). Proc. of the North American Conference on Logic Programming, The MIT Press, 1989.

[3] *Getting Started with I-MRP.* Internal Paper, Knowledge Based Progr. Dept., Hewlett-Packard Labs Bristol, January 1992.

[4] *BTS Models / BTS Requirements Specification.* Internal Papers, Knowledge Based Progr. Dept., Hewlett-Packard Labs Bristol, May 1992.

[5] J. Jaffar, J-L. Lassez: *Constraint Logic Programming.* Proc. 14th ACM POPL Conference, pp. 111–119, Munich, January 1987.

[6] J-L. Lassez, T. Huynh, K. McAloon: *Simplification and Elimination of Redundant Linear Arithmetic Constraints.* Proc. of the North American Conference on Logic Programming, The MIT Press, 1989.

[7] M. Nitsche: *Solving the Can-Build Problem Using Constraint Logic Programming.* MSc Thesis, Imperial College London, 1992.

[8] M. Nitsche: *Solving the Can-Build Problem Using* CLP($\mathcal{R}$) — *A Case Study.* Submitted to PLILP '94.

Types as Constraints in Logic Programming and Type Constraint Processing

Hans-Joachim Goltz
National Research Centre for Informatics
and Information Technology, GMD-FIRST Berlin
Rudower Chaussee 5, D-12489 Berlin, Germany
email: goltz@first.gmd.de

Abstract

A concept of types and type sorts for logic programming is introduced. Instead of defining "well-sorted" terms and substitutions, type constraints and type sort constraints are defined, where the approach is based on term models. A type is interpreted as a set of object terms (data terms) and a type sort is interpreted as a set of types. Parameters of types can be object terms and type variables. Types are defined by means of type rules, where type rules can be regarded as rules for the generation of the elements belonging to the corresponding type. The definitions of types and type sorts can be transformed into definite clauses and the least Herbrand model of these definite clauses can be used for the semantics. Computing with type constraints and type sort constraints is discussed and a strategy for type constraint processing is proposed. Type constraint processing is based on SLD-resolution with a specific computation rule.

1 Introduction

The paradigm of constraint solving plays a more and more important role in knowledge based systems and problem solving since it combines great expressive power with declarative programming. Constraint logic programming (CLP) is an elegant generalization of logic programming where the basic operation of logic programming, the unification, is replaced by the more general concept of constraint solving over a computation domain.

Structuring the universe of discourse is a successful method in the field of knowledge representation and automated reasoning. The introduction of types and sorts provides an appropiate means for such a structuring of the universe. Much research has been devoted to this task, and several proposals for various type systems have been made.