Design and Implementation of Practical Constraint Logic Programming Systems

Spiro Michaylov

August 24, 1992
CMU-CS-92-168

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored partly by IBM through a graduate fellowship and a joint study agreement, and partly by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: programming languages, declarative programming, logic programming, constraints, programming methodology, efficient implementation

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Design and Implementation of
Practical Constraint Logic Programming Systems

SPIRO MICHAYLOV

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Frank Pfenning — MAJOR PROFESSOR — August 24, 1992 — DATE

Peter Lee — MAJOR PROFESSOR — Aug 24, 1992 — DATE

R. F. Rashid — DEAN — 8/24/92 — DATE

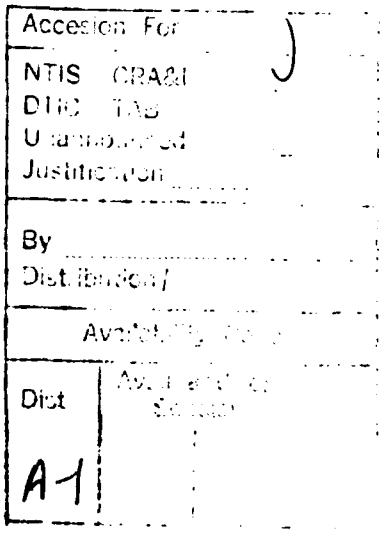
APPROVED:

Paul Christiano — PROVOST — 25 August 1992 — DATE

Abstract

The Constraint Logic Programming (CLP) scheme, developed by Jaffar and Lassez, defines a class of rule–based constraint programming languages. These generalize traditional logic programming languages (like Prolog) by replacing the basic operational step, unification, with constraint solving. While CLP languages have a tremendous advantage in terms of expressive power, they must be shown to be amenable to practical implementations. This thesis describes a systematic approach to the design and implementation of practical CLP systems. The approach is evaluated with respect to two major objectives. First, the Prolog subset of the languages must be executed with essentially the efficiency of an equivalent Prolog system. Second, the cost of constraint solving must be commensurate with the complexity of the constraints arising.

The language CLP($\mathcal{R}$), whose domain is uninterpreted functors over real numbers, is the central case study. First, the design of CLP($\mathcal{R}$) is discussed in relation to programming methodology. The discussion of implementation begins with an interpreter that achieves the efficiency of equivalent Prolog interpreters, and meets many of the basic efficiency requirements for constraint solving. Many of the principles applied in the interpreter are then used to develop an abstract machine for CLP($\mathcal{R}$), leading to a compiler-based system achieving performance comparable to modern Prolog compilers. Furthermore, it is shown how this technology can be extended so that the efficiency of CLP($\mathcal{R}$) approaches that of imperative programming languages. Finally, to demonstrate the wider applicability of the techniques developed, it is shown how they can be used to design and implement a version of Elf, a language with equality constraints over λ-expressions with dependent types.



Contents

List of Figures

Acknowledgements

The task of remembering *all* the people who helped me to get to the point of finishing this thesis, without missing anybody, is almost as daunting as writing the thesis itself.

I'm grateful to Gordon Preston for convincing me that I should learn some mathematical logic, thus bumping me onto the right path. John Crossley actually got me interested in it, and since then has provided valuable encouragement and support. Joxan Jaffar got me interested in logic programming, somehow convinced me that I really wanted a Ph.D. no matter how much I had thought otherwise, and that I was prepared to move to the other side of the world for five years to get it. Since then Joxan has provided guidance, encouragement and support beyond measure, in addition to fostering a thrilling research environment that transcended minor details like geographical location. During the years since it has been my pleasure and privilege to also work closely with Nevin Heintze, Peter Stuckey, Roland Yap and Chut Ngeow Yee, who all demonstrated through the fellowship of the "bird cage" that research could also be fun. I am also grateful to Jean-Louis Lassez for his thoughtful guidance and staunch support.

I have been most fortunate since coming to CMU in having a dynamic, friendly and supportive environment in which to work. I am particularly grateful to Nico Habermann for his insistence that I should be free to pursue *my* interests as much as possible, to Ed Clarke for his early support and guidance, Peter Lee for enthusiastic support as my co-advisor, and to Dana Scott for his insightful comments and guidance. Frank Pfenning's support, encouragement and guidance has been invaluable for the last few years. He has taught me a great deal about the λ-calculus and type theory, has never been phased by the sheer denseness of my questions, and has helped to make even writing a thesis quite exciting. The excellent administrative and support staff in the School of Computer Science have made my life and work much easier than they might have been.

At various times I have also learned a great deal by working with Scott Dietzen, Mike Epstein, Ed Freeman, Niels Jørgensen, Pierre Lim, David Long, Kim Marriott, Benjamin Pierce and Vijay Saraswat. The other faculty and students of the Ergo project and its successors have also provided many valuable learning opportunities.

I would like to thank my office-mates Nevin Heintze and Dean Pomerleau for putting up with me through good times and not-so-good, and finally all of my friends, including many of the above, who have helped make life so interesting, and have in recent times helped to keep me relatively sane.

For the past four years I have been supported financially by IBM through a graduate fellowship and other programs. These have contributed significantly to the progress of the work described in this thesis, and have helped to ensure my academic freedom.

It's just as well we don't have to thank our parents for all the wonderful things they do. In addition to the usual amazing efforts, my parents made great sacrifices to make sure I didn't have to participate in eastern Europe's half-century of wasted potential. For all these things, and their support as I have done what *I* wanted to do, my gratitude is boundless.

Pittsburgh
August 24, 1992

THESIS COMMITTEE

Peter Lee *(co-chair)*
Frank Pfenning *(co-chair)*
Dana Scott

Joxan Jaffar *(IBM)*

Part I

Background

Chapter 1

Introduction

Declarative programming is based on the idea that, for software engineering reasons, programs should be as close as possible to specifications of the problem and problem domain. While declarative languages have traditionally had a disadvantage with respect to imperative programming languages in terms of efficiency, it has been argued that the software engineering advantage made up for the lack of efficiency. While this argument is to some extent valid, the efficiency disadvantage has nevertheless hampered the acceptance of such languages for serious applications programming. The high-level goal of this thesis is to demonstrate, through the use of two specific languages, how this disadvantage can be alleviated for the constraint logic programming paradigm, which consists of a class of rule-based constraint programming languages.

The essential idea of logic programming is that a program should be a set of axioms in some logic and a computation should be a search for a constructive proof of a goal statement from the program. In the late 60s and early 70s a number of researchers developed systems that put these ideas into practice in the form of the programming languages Absys and Prolog. Over the next decade a great deal of work was done on semantics, programming methodology and implementation technology, to the point where now a substantial number of commercial Prolog systems exist, and Prolog has begun to be taken seriously as a programming language. Recently, considerable progress has been made towards the goal of developing Prolog compilers that achieve performance comparable to conventional imperative languages.

It has been widely observed that the expressive power of logic programming languages like Prolog is limited when reasoning in certain basic domains, such as arithmetic. The logic programming framework in its pure form requires the programmer to write code describing these domains in excruciating detail, and hence Prolog systems have tended to include *ad hoc* facilities for such reasoning. For this reason, Jaffar and Lassez defined the Constraint Logic Programming (CLP) scheme [83], which defines a class of rule–based constraint languages. These differ from traditional logic programming languages in that the basic operational step is constraint satisfaction rather than unification. A CLP language over a particular domain of computation can provide tremendous advantages in terms of expressive power over a logic programming language like Prolog, if the domain is appropriate to the applications being tackled. As

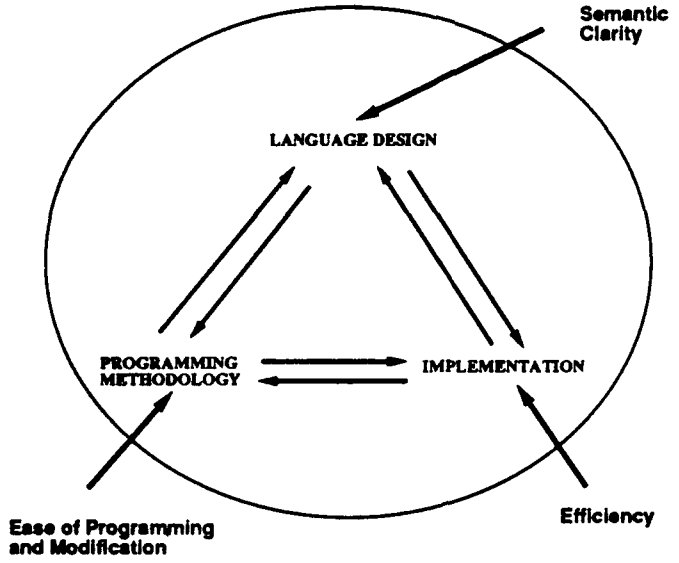


Figure 1.1: Requirements of a Programming Language

a result, it is easier to realize the high-level software-engineering objectives of logic programming with CLP languages.

Because the CLP scheme is so general, using it to design a practical CLP language requires more than choosing a domain that seems useful: efficiency must be taken into consideration. Indeed, it is easy to design a CLP language that is potentially useful, in terms of expressive power, yet obviously unimplementable. This may be the case if the language requires constraint solving in a domain that is undecidable, if no efficient decision algorithm exists, or no such algorithm is adaptable to the CLP operational model. Even the more realistic languages pose difficult implementation problems, as complicated decision algorithms can have too much overhead for the kinds of problems repeatedly arising in the execution of a program. The question is whether it is possible to define a range of CLP languages powerful enough to provide real benefit over traditional logic programming and yet can be efficiently executed.

The specific objective of this thesis is to develop and demonstrate language design and implementation techniques for making a class of CLP languages practical. It describes a systematic approach to the design and implementation of practical CLP systems. The work begins with the understanding that designing a practical CLP language depends on an understanding of not only programming methodology, but also implementation efficiency issues. Furthermore, to implement a CLP language efficiently, it is again necessary to consider programming methodology. To clarify what is unusual about this approach to language design and implementation, let us consider the diagram in Figure 1.1. It expresses the essential requirements of a programming language and its implementations. The parameters that need to be balanced are language design, programming methodology and implementation strategy. The requirements can be broadly divided into those that are external (semantic clarity of the language, ease

of developing and modifying code, efficient execution of programs) and those that are internal (the causal relationships *between* the parameters). Some of the internal relationships between the parameters are natural or even trivial, and others are forced on us by pragmatics. The natural ones are:

- *design* $\longrightarrow$ *implementation*
  Obviously, the language design defines the implementation problem.

- *design* $\longrightarrow$ *methodology*
  Again, the programmer can only use what is provided by the language.

- *methodology* $\longrightarrow$ *design*
  It should come as no surprise that the language designer is typically influenced heavily by an intended style of programming and class of applications.

The following influences are less than desirable, but in the case of CLP, must be observed.

- *methodology* $\longrightarrow$ *implementation*
  Ideally, it should be possible for the implementor to work in a vacuum: simply implement all features of a language as efficiently as possible. Because of the power of CLP languages, this is often impossible. In particular, implementing some aspects of constraint solving efficiently will hinder the efficiency of another aspect. For this reason, it is important to know which features are most important to the programmer.

- *implementation* $\longrightarrow$ *design*
  Again, because of implementation difficulties resulting from an excessively general domain, it is necessary to consider the problems of solving various kinds of constraints when designing the language.

- *implementation* $\longrightarrow$ *methodology*
  This is what we hope to avoid: ideally, the programmer should not have to understand how the system is implemented.

As a result of exploring these relationships for two case studies and considering other domains, this thesis relies heavily on the observations that:

- *The simple cases of constraint solving are usually the most frequent.*
  A programming language based on rules and constraints is useful largely because programs can be more than just static descriptions of constraint satisfaction problems. During the execution of a program, systems of constraints are gradually constructed, composed and specialized. Such computation results in large numbers of simple constraints on the way to building complex constraint networks.

- *For a large range of domains and applications, handling the simple cases of constraint solving well is necessary for an efficient implementation overall.*

> These cases are frequent enough to form a bottleneck if the overheads of dealing with them are too high. It is still important to deal with the general case efficiently, but it is not sufficient.

The language CLP($\mathcal{R}$), whose domain is uninterpreted functors over real numbers, provides the central case study. First, the design of CLP($\mathcal{R}$) is discussed in relation to programming methodology. The discussion of implementation begins with an interpreter that achieves the efficiency of equivalent Prolog interpreters, and meets many of the basic efficiency requirements for constraint solving. Many of the principles applied in the interpreter are then used to develop an abstract machine and compilation model for CLP($\mathcal{R}$). The IBM compiler-based CLP($\mathcal{R}$) system has been heavily influenced by the ideas in this thesis, and in turn has provided a testbed for them. It achieves performance comparable to modern Prolog compilers for Prolog programs, and improves significantly on arithmetic constraint solving performance of the earlier interpreter. This system has been widely distributed, and has been used by various programmers for developing non-trivial applications. It is also shown how this technology can be extended so that the efficiency of CLP($\mathcal{R}$) approaches that of imperative programming languages.

Finally, to demonstrate the wider applicability of the techniques developed, it is shown how they can be used in the design and implementation of Elf, a language with equality constraints over λ-expressions with dependent types. Superficially, Elf is a very different language to CLP($\mathcal{R}$). However, the parallels with CLP($\mathcal{R}$) in terms of programming methodology and implementation tradeoffs are quite striking.

Some comments need to be made about the context of the work on CLP($\mathcal{R}$) described in this thesis.

CLP($\mathcal{R}$) was first described in 1986 [70], and its original implementation was described in [85] and distributed openly in 1987. Hence, CLP($\mathcal{R}$) was the first arithmetic-based CLP system available. A number of CLP languages related to CLP($\mathcal{R}$) eventually emerged. The first of these, Prolog III, was described in [30], formally specified in [31] and distributed (commercially) around 1990. While the significant differences between CLP($\mathcal{R}$) and Prolog III are described in subsequent chapters, the important point here is that the work done in this thesis is completely independent from that on Prolog III.

The author has participated in the ongoing CLP($\mathcal{R}$) implementation effort from its beginnings, thus putting into practice many aspects of this thesis. However, there are aspects of CLP($\mathcal{R}$), and of CLP in general, that are not addressed in this thesis. For example, solving linear inequalities and projecting constraints for output are not addressed in detail.

The remainder of the thesis is organized as follows. Chapter 2 concludes this part of the thesis with a survey of the areas of logic programming, constraint programming and constraint logic programming, concentrating on language design, programming methodology and system implementation issues. Part II deals with language design and programming methodology. It begins with Chapter 3, introducing CLP language design issues, as affected by programming methodology and implementation issues. Then Chapter 4 discusses the design of CLP($\mathcal{R}$), explores one area of application in de-

tail, and ends with an empirical study of CLP($\mathcal{R}$) programs. Similarly, Elf is explored in Chapter 5. Part III deals with implementation issues and techniques. Chapter 6 deals with implementation issues in greater detail, and outlines a strategy for efficient implementation of CLP systems. The following four chapters explore four aspects of this strategy in detail, concentrating on two implementations of CLP($\mathcal{R}$). Chapter 7 discusses the organization of multiple constraint solvers, and Chapter 8 discusses managing the delaying of hard constraints. Chapter 9 deals with the problem of tailoring constraint solvers to support the CLP operational model. Then Chapter 10 addresses the issue of compiling CLP languages, again concentrating on CLP($\mathcal{R}$). Finally, Chapter 11 discusses a plan for implementing Elf from the viewpoint of the same basic techniques.

Chapter 2

Constraints and Logic Programming

The concept of Constraint Logic Programming is a natural fusion of the concepts of logic programming and constraint programming. Therefore, we begin by considering the basic concepts and history of the two separate areas. Then we introduce CLP, review a number of CLP languages and systems, and discuss applications.

2.1 Logic Programming and Prolog

The original motivation for logic programming was the belief that, for software engineering reasons, it is desirable for programs and specifications to be identical. That is, programming should be done in an executable specification language. Furthermore, it was believed that first–order logic was an appropriate specification language. It then remained to determine how some subset of first–order logic could be made executable, given the understanding that general theorem proving in first–order logic was infeasible as a basis for a programming language.

For various perspectives on the history of logic programming, see [25, 48, 98]. In this brief review, we will ignore the development of logic programming semantics, for which the interested reader should see the book by Lloyd [110]. Instead, we concentrate on the development of the operational model (and hence design of the language), the development of efficient implementations, and the understanding of programming methodology.

2.1.1 The Prolog Operational Model

Much of the early work on logic programming was the result of the confluence of two different research endeavors: automated theorem proving and natural language processing. This work was made possible by Robinson and others [146] through the discovery of resolution and the re-discovery of unification, earlier described by Herbrand in 1930 [71]. Thereafter, a number of systems were developed that essentially incorporated a

subset of resolution and could loosely be called logic programming languages. The first one seems to have been Absys 1, developed in 1967 by Elcock and others [48, 49, 54]. PLANNER, which was notable for its extreme inefficiency, was developed by Hewitt [72] in 1969. Colmerauer and others [26, 27] implemented Prolog (*Programmation et Logique*) in 1973 and Kowalski [97, 100] systematically formulated the procedural interpretation of a Horn clause set in 1974. It was the work of the groups headed by Colmerauer and Kowalski that was the precursor of much of modern logic programming, and Prolog is still clearly the dominant logic programming language. Because of the importance of logic programming and Prolog to the subject matter of this thesis, an informal introduction is given here.

The essential idea of Prolog, as Kowalski showed, is that an axiom

$$A \text{ if } B_1 \text{ and } B_2 \text{ and } \ldots \text{ and } B_n$$

can be considered as a procedure definition in a recursive programming language, where A defines the name and arguments of the procedure and the B_is are the body. Also, it says that to solve A, it is sufficient to solve all of the B_is — that is, to execute them. Now, a program is a set of such procedure definitions, and to run a program we give a goal, which is just a list of procedure calls. That is,

$$B_1 \text{ and } B_2 \text{ and } \ldots \text{ and } B_n$$

which simply says that the B_is need to be solved. The axioms correspond to definite Horn clauses in first–order logic, with function symbols used to create data structures. Goals correspond to the negation of a conjunction of atoms in first-order logic. Hence executing a program corresponds to combining the negation of a statement with a set of Horn clauses, and deriving a contradiction by resolution, hence proving the original statement by *reductio ad absurdam*.

Prolog is an implementation of this idea with a different syntax and a specific rule describing how the search aspect of resolution is to be carried out. Variables are strings starting with an upper case character, and predicate and function symbols are strings starting with lower case characters. Commas represent conjunction and `:-` represents "if". Goals begin with `?-`, and both rules and goals end with a period. Operationally, a list of goals is solved from left to right, and the search for appropriate rules is depth–first. The notion of unification is altered somewhat, as we will describe later. To give a simple example, the incomplete program from [172] in Figure 2.1 represents a simple ancestor relationship, where `father(X,Y)` says that `X` is the father of `Y`, and `son(X,Y)` says that `X` is the son of `Y`. An appropriate goal might be `?- grandfather(terach, X)`. with the expected answer being `X = isaac`.

It is important to consider why Prolog was designed the way it was. The choice of language (definite clauses) and program invocation (a conjunction of atoms) interacts with the choice of operational model (linear resolution, depth-first search) in a significant way. At each operational step, a derived goal is resolved with an input clause. Thus each resolution step can be seen as analogous to a procedure call in a conventional programming language. Furthermore, this enables the notion of "activation record"

```
father(terach, abraham).
father(abraham, isaac).

male(terach).
male(abraham).
male(isaac).

son(X, Y) :-
        father(Y, X),
        male(X).

grandfather(X, Z) :-
        father(X, Y),
        father(Y, Z).
```

Figure 2.1: Simple Prolog example

from conventional programming languages to be used to represent variable bindings in Prolog — an important step in making a Prolog implementation efficient, by avoiding the copying of clauses. Because depth-first search is used, the proof tree that needs to be represented at any point in time is a list, so the space used is merely linear in the depth into the tree. Backward chaining search (starting from the goal and searching back to the axioms) also helps make programming correspond more intuitively to algorithmic notions. The left-right atom selection rule enforces this correspondence, as the programmer knows what order "statements" will be executed in. Additionally, this avoids the cost of keeping track of which atoms have yet to be reduced, as a simple continuation can be used. Similarly, the depth-first search rule allows the use of a single failure continuation. Even at this level, however, the operational model involves a compromise. While corresponding well to algorithmic notions of programming, and being amenable to efficient execution, completeness has been lost. A program may logically imply some solution, but the execution will result in an infinite loop. This is the first of many examples we will discuss of language design being influenced by observations about implementation and programming methodology. It was judged that the problems introduced by this operational model would not interfere with programming significantly, and that the efficiency benefits were sufficient to justify the decisions.

A further compromise in the design of Prolog was the omission of the occurs check. This is a case of the usual Robinson unification algorithm that prevents a variable from being unified with a term that properly contains it, since no finite term can satisfy the corresponding equation. Clearly the decision to omit the occurs check makes sense from an efficiency viewpoint, since it is expensive, in general requiring a full traversal of a (possibly large) term. At the time it was believed that the cost could not be avoided in

any other way. This damages soundness and completeness, causing most real systems to crash under certain circumstances – the consequences were described in some detail by Plaisted [144]. It was originally believed that only contrived programs would cause a problem. However, since then it has been shown that

- It can be determined dynamically and efficiently that the occurs check can safely be eliminated in many cases (see for example [11]).
- Global analysis techniques can conservatively inform a compiler of many more instances where the occurs check can be eliminated.
- Certain important programming techniques (*e.g.*: difference lists) result in programs that run incorrectly as a result of leaving the occurs check out.
- As will be discussed in more detail later, an alternate approach to solving this problem is to change the semantics of Prolog so that the problem becomes a feature. This was the approach taken by Colmerauer in Prolog II [28].

In contrast with the largely successful compromises described above, the occurs check provides an example about the pitfalls of making design decisions on the basis of the kinds of programs "nobody writes", since they are really observations about what "nobody has written yet". Later we will have cause to remember this lesson.

Even the choice of basing Prolog on Horn clauses was a major compromise in its design. One unfortunate aspect of programming with Horn clauses is that it is not possible to directly represent negative information. That is, we cannot say that a certain property does not hold or that a certain property holds if some other does not. Such negation would require a more general form of resolution, which would introduce inefficiency and does not correspond readily to algorithmic notions. A closely related problem is that of disjunction – we cannot say that one of a number of facts holds, without knowing which one.

The history of "solutions" to the negation problem again contains some interesting lessons. The original compromise solution, dating all the way back to Absys, was to add the (semantically) incorrect inference rule called negation as failure. Again it was argued, somewhat convincingly, that it was what programmers really needed. Again, examples were found where this was false. More recently, there have been many attempts to formulate a notion of negation that are feasible, semantically well-behaved, and actually useful to programmers. An approach that fits in well with the ideology of this thesis is that of nH-Prolog [111], which implements classical negation, but such that pure Prolog programs can run efficiently, and efficiency degrades depending on how much negation is used. The name "nH-Prolog" comes from the idea that the language is best suited to programs that are "near Horn". For a general overview of the issue of negation in logic programming, the reader should consult the two survey papers by Shepherdson [157, 158].

2.1.2 Implementation Techniques

As soon as Prolog was developed, a large amount of activity was devoted to striving for more efficient implementations. The invention of structure sharing by Boyer and Moore [18, 122] was important to making the early implementations at all practical, as large numbers of derived goals could be represented. In the mid to late 1970s, D. H. D. Warren and others at the University of Edinburgh developed the DEC–10 Prolog compiler [188, 190]. This was a significant development, as it demonstrated that Prolog could be executed just as efficiently as LISP. At that time, it had been considered that Prolog's efficiency disadvantage with respect to LISP was a barrier to its broader acceptance.

In 1982, both Bruynooghe [20] and Mellish [115] reported that in some cases structure copying was appropriate for the representation of complex terms, laying part of the foundation for the Prolog compilers of today. In particular, these studies showed that while neither strategy had a very clear advantage in terms of space utilization copying made accessing complex terms faster, while making their creation slower. The issue was settled by the observation that a structure, once created, might be accessed many times.

In 1983 D.H.D. Warren [189] described an abstract instruction set intended as a target language for compiling Prolog, as well as an associated architecture. This became known as Warren's abstract machine (WAM), and has provided the basis for most modern Prolog compilers. The WAM is particularly suitable for software emulation of compiled code. The essential idea behind it is that the instructions can be used to represent variants of the unification operation – specialized by partially evaluating unification with respect to the terms in the program. Since the WAM was described, a number of less major developments have led to a number of efficient compilers being made available commercially. These developments include clever strategies for register allocation, clause indexing (whose effect is heavily dependent on programming style), management of dynamic code (that is, involving assert/retract), and so on. Many variants of the WAM exist for making various operations more efficient, and many systems use a vastly increased number of specialized instructions to improve overall performance. Native code compilers are now also widely available.

Most recently, two Prolog compilers developed by Van Roy [147] and Taylor [180] have demonstrated that even the efficiency of imperative languages may eventually be attainable. Both systems are based on global analysis of programs using the technique of abstract interpretation [40]. This technique obtains large amounts of information about the types and instantiation of variables in clauses, allowing particularly efficient code to be generated. Once again the efficiency is obtained by making some assumptions and claims about programming methodology. The programmer is required to give information about which predicates will be used in queries and how — and in the case of Taylor's system, is limited to using only one predicate symbol at the top level. The analysis and hence compilation works best if this set of "allowed queries" is quite limited. Furthermore, the analysis only works well if the use of dynamic code and meta-level programming is reasonably localized — preferably limited to a few modules.

2.1.3 Programming Methodology and Practical Experience

Like the areas of logic programming semantics and Prolog implementation, the area of Prolog programming methodology has come a long way. While Clocksin and Mellish [24] gave the first detailed account of how Prolog could be used for serious programming, the text by Sterling and Shapiro [172] takes a more modern and conceptually sounder approach. Most recently, O'Keefe [132] described how to write Prolog programs that are both well-structured and efficient, and Sterling [171] collected a number of articles describing the use of Prolog in practical settings. In summary, Prolog programming methodology has been widely discussed and is quite well understood.

The accumulated wisdom falls into the following broad categories:

- Effectively expressing the program.
  This begins with rule-based programming. The use of recursion, logical variables, and partially instantiated data structures is also important. At a more advanced level, we have the use of difference lists, and "all solutions" predicates.

- Making programs efficient.
  This involves understanding aspects of the implementation, and using them to the programmer's advantage. For example, many Prolog compilers index clauses using only one argument position in the head. Thus it can be useful for the programmer to know which argument position this is. Furthermore, many compilers include special optimizations for shallow[1] backtracking and tail recursion, and taking advantage of these can be very useful. Sometimes programs can be written to make it easier for the compiler to determine that a predicate is deterministic, so that choice point records are not created. (Choice point records are a runtime structure used to keep track of which alternate clauses might be used to solve a subgoal.) Judicious use of certain control predicates, such as the cut predicate, can help the compiler find opportunities for removing choice point records that have already been created.

- Sidestepping the deficiencies of Prolog.
  This includes the effective (and correct) use of negation as failure. The cut and `once` predicates, and `if then else` are important for limiting the number of solutions to subgoals.

2.2 Constraints and Constraint Programming

Constraint programming has its roots in artificial intelligence. Traditionally, constraint satisfaction problems (CSPs) consist of a graph $\langle \mathcal{N}, \mathcal{A} \rangle$ where nodes $\mathcal{N} = \{n_1, \ldots, n_k\}$ and $\mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$, a set of values $\mathcal{V} = \{v_1, \ldots, v_m\}$ and for each arc $a \in \mathcal{A}$ a set $\mathcal{V}_a \subseteq \mathcal{V} \times \mathcal{V}$. The problem is to find a mapping ϕ of values to nodes such that for each $(n_i, n_j) \in \mathcal{A}$, we have $(\phi(n_i), \phi(n_j)) \in \mathcal{V}_{(n_i,n_j)}$. This and its many small variations are often called the labeling problem. The idea is that the nodes correspond to variables

[1] Shallow backtracking is searching for an appropriate clause head.

that need to be assigned values from the domain $\mathcal{V}$ such that the constraints represented by the $\mathcal{V}_a$ are satisfied. (Clearly, a relation can trivially be regarded as a constraint.)

The essential idea of constraint programming has been that a language should include a number of primitive predicates and operators over some domains (such as equality and inequality predicates, and addition and subtraction operators, on integers) and that computation should proceed by applying built-in constraint satisfaction algorithms for these domains. The hope is that this makes computation non-procedural (ie: declarative) since the order of constraints is unimportant, and thus a program is a problem specification rather than a procedure for solving the problem. The analogy with CSP is weak since the arc relations are replaced by the definitions of primitive constraints, and constraint satisfaction is no longer necessarily just a search problem.

As an example, let us consider the problem of modeling a simple electrical circuit with a voltage source V and two parallel resistors R_1 and R_2. The currents through the resistors are I_1 and I_2 respectively, and that through the voltage source is I_T. Then the system is described by the constraints

$$\begin{aligned} I_T &= I_1 + I_2 \\ V &= I_1 R_1 \\ V &= I_2 R_2. \end{aligned}$$

We will return to this in the following discussion. We begin by discussing constraint satisfaction techniques in isolation, then discuss how constraints can be used as the basis for a programming language, and finally survey some of the major systems based on these ideas.

2.2.1 Constraint Solving

A large part of the power of constraint languages comes from the power of the underlying constraint solving algorithms, which are briefly surveyed here.

Classical Constraint Satisfaction

Classical constraint satisfaction problems are in general NP-complete, and are typically solved by some form of backtracking search through the possible assignments to respective variables. This search can be made more efficient by incorporating some form of intelligent backtracking [42], heuristics to help choose which variables to instantiate first, and with which values, to maximize pruning (variable and value ordering – [64]), and consistency labeling techniques. These are a class of pruning techniques that make use of local inconsistences [64]. They include the well known techniques of forward checking, partial lookahead and full lookahead. The papers by Davis [37] and Dechter & Pearl [41] provide a comprehensive overview of this area.

Algorithms for Specific Domains

Of course a number of domains are so important that constraint solving in them has generated a large amount of specialized investigation. They need to be considered from

a number of viewpoints, including theoretical complexity and performance in practice. Furthermore, as we shall see later, it is sometimes important that they be flexible. In particular, algorithms that can quickly re-solve an augmented set of constraints are of great value. Here we give an overview of some of the algorithms of greatest potential interest for CLP systems.

Real arithmetic has generated the most activity because it is so fundamental and so important in practice. Tarski [179] showed that the theory of real closed fields was decidable, and gave a decision algorithm. Not only is the worst-case complexity of this algorithm doubly exponential, but it is totally infeasible in practice. Some work has been done on more efficient variants of this algorithm (by *e.g.* Arnon [8]), but this seems to have had little impact on applications. The restricted case of linear equations and inequalities has been dealt with rather successfully. For linear equations, we have Gaussian elimination, and a host of iterative algorithms with various advantages, mostly in terms of numerical stability if floating point arithmetic is to be used. Linear inequalities can be solved using linear programming techniques, such as the Simplex algorithm [35], which is exponential in the worst case, performs well in practice, and is in a formal sense polynomial on average [15, 153]. Khachian [94] gave a linear programming algorithm that is polynomial in the worst case, but is disappointing in practice. More recently, another polynomial-time algorithm by Karmarkar [93] has been getting quite impressive results. In fact, sets of linear inequalities restricted to no more than two variables per constraint can be solved very efficiently, as was shown by Shostak [161]. The algorithm was extended to larger numbers of variables, and seemed to perform well, as long as the number of variables was not too high. Nonlinear real constraints are more problematic. Techniques used include interval arithmetic, iterative methods, and symbolic algebra based on Gröbner bases [149]. It should be noted that the typical implementations of the algorithms mentioned use floating point arithmetic, which is unsound. This is an instance of compromise between efficiency and semantics.

Because of the unsoundness of floating point arithmetic, some work has been done on (infinite precision) rational arithmetic. Rational arithmetic for polynomials with equations and inequalities is also decidable. Additionally, work has been done on sound implementations of floating point arithmetic [133, 105].

Boolean constraint solving, essentially propositional theorem proving, has also generated a substantial amount of activity. Of course it is well known that the problem is NP-complete, so the work has concentrated on algorithms that tend to work relatively well in practice, if only for certain kinds of problems. The techniques include the method of truth tables, semantic unification [21], SL-resolution [100], and Gröbner bases [151]. The Gröbner bases technique has the advantage of producing a canonical form (whose significance will be discussed in Part III), not introducing spurious variables, and being somewhat incremental (another issue to be discussed in Part III). SL-resolution is also incremental, but requires formulas to be translated to clausal form. The Davis-Putnam algorithm is also being considered, and appears to be able to deal with constraints on much larger numbers of variables. However, the representation it uses is not well suited to the problem of producing a suitable projection of the constraints for the purpose of output.

Word equations, or equivalently equations between string expressions where concatenation is the only operation, are of considerable theoretical and practical interest. In particular, they are applicable to natural language processing. The decidability problem was solved by Makanin [113], although the decision procedure given there did not actually produce a set of solutions to an equation. Jaffar [81] gave a decision procedure that also generates a (possibly infinite) minimal and complete set of solutions. This algorithm, however, is impractical, and this problem has not been solved in the unrestricted case. Furthermore, it is not clear that a restriction can be found that makes the problem tractable but useful. An attempt at this will be discussed later in this chapter.

2.2.2 Classification of Constraint Languages

It is rather difficult to review the development of constraint programming historically because the work has often failed to build upon or even acknowledge previous work. However, constraint programming languages tend to be characterized according to three key parameters:

- How constraints in a program are interpreted. The issue is whether they represent constraints as such or are templates for constructing constraints at run time. The former are called *static* constraints, and the latter *dynamic* constraints, and the run-time copies of these are *constraint instances.*

- Whether constraints affect the control of a program. That is, what effect, if any, the result of solving some constraints can have on the future execution of the program.

- How and to what extent constraints are solved. A system of constraints is solved by local propagation if all the variables in the system become determined after a finite number of local propagation steps. A *local propagation step* occurs when a constraint has a sufficient number of determined variables for some of its other variables to be determined. These newly determined variables may then precipitate further local propagation steps in other constraints. Instead of local propagation, a more involved algorithm, such as Gaussian elimination, may be used. Also, it is possible to specify that only constraints in certain syntactic classes are solved. For example, constraints on real numbers might be restricted to being linear.

Returning to the electrical circuit example above, the 2nd and 3rd equations can be seen as instances of the (dynamic) constraint $V = IR$. Now, if we are given values for V, R_1 and I_T, then all unknowns can be solved using local propagation as follows:

$$I_1 \leftarrow V/R_1$$
$$I_2 \leftarrow I_T - I_1$$
$$R_2 \leftarrow V/I_2$$

where $\leftarrow$ denotes assignment. On the other hand, if we are given R_1, R_2 and I_T, it is not possible to order the equations so that they can be solved by this simple evaluation and assignment method, since there are cyclic interdependencies between the variables. That is, the system cannot be solved by local propagation, and a simultaneous equation solver is needed.

2.2.3 Review of Systems

Now, rather than attempting a historical development, we will classify some of the most important constraint languages according to the above three parameters, and their intended application areas, where appropriate.

A Problem Solving Language: REF-ARF

In 1970, Fikes [53] described a system called REF-ARF. The REF component was an essentially procedural programming language, with `if-then` and `condition` constructs where boolean conditions were either integer or boolean constraints. Programs were executed by the system called ARF, which employed heuristic-controlled backtracking, since the statements involving constraints potentially resulted in nondeterministic choicepoints. The integer constraints only allowed addition and subtraction, so were relatively easy to solve. The test and generate paradigm was also employed. This was facilitated by integer variables always being bounded above and below.

The MIT Approach

The MIT AI Lab studied constraints in the 70s and early 80s from the viewpoint of general problem solving and especially applications to circuit analysis and synthesis. The CONSTRAINTS system, described by Steele and Sussman [169, 170, 176] used local propagation to solve constraints. The system of constraints was built up by connecting instances of macro-definitions, but was essentially static. To avoid the problem of solving simultaneous equations, special-purpose heuristics, such as the voltage divider law, were built into the circuit analysis programs. Some of the other systems, such as EL/ARS [175, 168] and SYN [39] used MACSYMA as their basic constraint solver, to avoid the restrictions of local propagation. While Steele [169] noted the conceptual correspondence between logic programming and the constraint paradigm, he did not take any concrete steps to exploit this correspondence.

Interactive Graphics: SKETCHPAD, THINGLAB, MAGRITE and JUNO

Sketchpad [177] was the first interactive drawing program. It allowed the user to build up geometric objects from primitives and geometric constraints. The constraints were solved using local propagation where possible, and relaxation.

ThingLab [16, 17] was an object oriented language incorporating many of the ideas of Sketchpad. It also used local propagation and relaxation, but separated constraint solving into planning and execution phases. When a user started to manipulate part of

an object, a plan was generated for quickly re-solving the appropriate constraints for that part of the object. This plan was then repeatedly executed while the manipulation continued.

While some of the process was later automated, in the original version of ThingLab the user had to specify methods for propagating values for each constraint predicate. The user was also responsible for ensuring that these methods actually satisfied the predicate. Instances of constraint templates could be obtained by using the object-oriented facilities of the language, but were essentially static during execution.

Magritte [58] was an editor for simple line drawings. Local propagation was used to solve constraints, but the constraint network was searched in a breadth-first manner to find the shortest path to a solution.

Juno [129] was a system for geometrical layout. It was unusual in not using local-propagation at all – everything was solved using a Newton-Raphson solver. The approach to defining constraints was novel, as either textual or graphical description could be modified by the user, changes immediately being reflected in the other form.

Typesetting: METAFONT and IDEAL

Ideal [184] is a picture description language for typesetting. It uses complex numbers to represent positions, and solves linear equations only. One complex equation really corresponds to two real equations. The language has no general purpose control structures. Constraints are built up hierarchically using boxes. METAFONT [95], a font definition package, uses a solver similar to that of Ideal, but somewhat weaker. Li [108] describes a text layout system that also solves linear systems.

Amalgamation with Other Paradigms

Freeman-Benson [56] describes and motivates incorporating constraints into imperative programming. The paradigm is called Constraint Imperative Programming (CIP). In the sample language, Kaleidoscope '90, there are two kinds of constraint expressions. These are explicit constraints and Smalltalk-80 assignments. Variables are time-stamped, so an assignment of the form `x <- x + 1` establishes the constraint $x_{i+1} = x_i + 1$. Constraints are also hierarchical, some being required, and others just being preferred, at various levels of priority.

Hill [73] developed a language called MEL, which adds constraints to LI^{C}P. Darlington and Guo [36] proposed a framework for Constraint Functional Programming.

Spreadsheets and Modeling Tools

Spreadsheets such as Visicalc^{TM} are essentially constraint systems relying on local propagation. The constraints are generally static, but they do incorporate (rather *ad hoc*) control constructs. Their special feature is that the user's view is a matrix of data cells that are related by constraints. Changing the value of a cell instantly results in changes to the values of the cells that depend on it. However, in most systems the propagation tends to be "forward propagation" in the sense that some cells are

designated data cells, whose values are provided by the user, and others are "derived" calls, whose values are to be computed based on the values of other cells.

TK!Solver [96] is a spreadsheet-based system that incorporates undirectional constraints, solved by full local propagation and, where necessary, relaxation. If relaxation is to be used, the user is required to provide an initial guess for the values of the appropriate variables.

HEQS [43] is a financial modeling system incorporating an extended version of the Ideal constraint solver.

A Meta-Language for Constraint Languages: BERTRAND

Leler [106] proposed the language Bertrand essentially as a meta-language for implementing constraint solvers. It is a general constraint programming language, based on term rewriting, with dynamic constraints and general control mechanisms. All constructs of the language are based on augmented rewrite rules, and the programmer must simply add rules for the constraint solving algorithm. A solver for linear equations, using Gaussian elimination, is one of those proposed. Constraint solving in Bertrand can control the execution of programs, through a conditional mechanism called "higher order constraints" – constraints on other constraints. Some interesting ideas on compiling Bertrand were also proposed.

2.3 Constraint Logic Programming

A number of researchers recognized that replacing the syntactic unification of Prolog with some other form of unification would increase its expressive power. A lot of the early work dealt with unification with respect to some equality theory that was part of the program, using an operational model that relied on generalized unification [150]. A general framework for such languages was given by Jaffar, Lassez and Maher [80]. The major problem with these languages is that unification with respect to an equality theory has never been shown to be efficient in practice.

Apparently the first deliberate attempt to replace unification uniformly by constraint solving in some other domain was Colmerauer's Prolog II [28], which is based on the observation that unification without the occurs check corresponds more closely to solving equations over rational trees, as described below.

The Constraint Logic Programming scheme was originally described by Jaffar and Lassez in [82, 83]. It defines a class of rule–based constraint languages, each of which is specified by giving a structure of computation. That is, it gives a formal basis for logic programming languages where unification of finite trees is replaced by constraint solving in some domain as the fundamental operational step. Traditional logic programming can be seen as an instance of this scheme.

2.3.1 Examples

Let us consider a number of structures and simple example programs for CLP over those structures. First, consider a subset of boolean algebra with the underlying set $\{0,1\}$, the interpreted constants 0 and 1, interpreted function symbols $\vee, \oplus, \cdot, \neg$ and the only interpreted predicate (constraint) symbol $=$. Then a full adder with inputs `In1` and `In2`, carry input `CarryIn` and output `Out` and carry output `CarryOut` could be represented by the single clause

```
adder(In1, In2, CarryIn, Out, CarryOut) :-
      In1 ⊕ In2 = X1,
      In1 · In2 = A1,
      X1 ⊕ CarryIn = Out,
      X1 · CarryIn = A2,
      A1 ∨ A2 = CarryOut.
```

To run this on a typical example we could use the goal `?- adder(1, 1, 1, Out, CarryOut).` and obtain the answer constraint `Out = 1, CarryOut = 1`, or, alternately, we can use a goal like `?- adder(1, 1, CarryIn, Out, 1).` to obtain the answer constraint `Out = CarryIn`.

As another example consider the structure with the finite (possibly empty) strings over the alphabet $\{a,b\}$ with the interpreted constants ϵ, a, b, interpreted function symbol $\oplus$ for concatenation, and equality. Then a program for recognizing and generating palindromes might be written as:

```
unit(a).
unit(b).

palindrome(ε).
palindrome(X) :-
      unit(X).
palindrome(X ⊕ Y ⊕ X) :-
      unit(X),
      palindrome(Y).
```

Then the goal `?- palindrome(a ⊕ b ⊕ a).` can be expected to succeed with an empty answer constraint, while the goal `?- palindrome(X ⊕ b ⊕ a).` has an infinite number of potential answer constraints including `X = a`, and `X = a ⊕ b`, *etc.*

2.3.2 Review of Languages and Systems

Since the CLP scheme was defined, a number of older programming languages have been described in terms of it, and many new languages have been designed with it in mind. Prolog II can be seen as an instance of the scheme. This subsection is a review of the major proposed CLP languages and/or implemented systems.

ABSYS

As already mentioned, the early versions of Absys were very similar to Prolog. However, in a late version [49] some simple arithmetic constraints were introduced. These were solved by local propagation.

Prolog II

Described in [28, 29], the intent of Prolog II was to design a version of Prolog that had a defensible, yet efficient, unification algorithm. It was observed that the kinds of loops that the occurs check sought to avoid occur in rational trees. Rational trees can be defined as either

- the possibly infinite trees with a finite number of subtrees, or
- those that can be represented as a finite set of term equations of the form

$$\begin{aligned} x_1 &= \Psi_1(\tilde{x}, \tilde{t}) \\ &\vdots \\ x_n &= \Psi_n(\tilde{x}, \tilde{t}) \end{aligned}$$

  where the t_i are parameters, x_1 to x_n are all distinct variables, and the right hand sides are all finite terms.

Unification of rational trees was quite efficient, and would solve one of the major semantic problems of Prolog. Since it was observed that equations were being solved in the domain of rational trees, and disequations could be solved as well, explicit disequations were added to the language. An interesting aspect of the language is that delay is used to deal with disequality constraints. In particular, the `dif/2` predicate is really a disequality constraint that suspends whenever a variable could be bound to make the two terms equal. That is, for example `?- dif(f(X),g(X))` immediately succeeds, `?- dif(f(c),f(c))` immediately fails, and `?- dif(f(X),f(c))` suspends until `X` is instantiated (bound to something other than a variable). In that case, it is awakened and fails if X is now `c`, and succeeds otherwise. The utility of general delay was also recognized, leading to the implementation of the `freeze/2` predicate, which delays the second argument, a goal, until the first has been instantiated.

Unfortunately, while it is possible to give small programs that simulate certain automata, Prolog II has never found a major application area of its own. It can reasonably be used as an alternative to Prolog, but there are certain applications of Prolog where finite trees *are* the appropriate domain. For example, in the implementation of queues in terms of difference lists as described in [172], the clause `empty(X\X).` describes an empty queue, but without the occurs check a call to this clause may succeed erroneously with a non-empty queue, through the creation of a cyclic binding.

CLP($\mathcal{R}$)

CLP($\mathcal{R}$) [70, 85] incorporates uninterpreted functors over real numbers as the domain, allowing as explicit constraints equalities between terms and equalities and inequalities between arithmetic terms. Constraints that are deemed to be too difficult for the constraint solver are delayed in the hope that they will become simpler. The implementations use floating point arithmetic to approximate real arithmetic. Throughout this thesis, the design, programming methodology and implementation of CLP($\mathcal{R}$) will be discussed at length.

Prolog III

Prolog III [30, 31, 32], the successor to Prolog II, was designed as a general fusion between constraint programming and logic programming. The domain consists of rational trees, rational numbers, lists and booleans. Equality and disequality constraints are available on all terms. Furthermore, inequalities are available for rationals. The usual four arithmetic operations are available for rationals, concatenation for tuples, and the usual boolean connectives. In what was called a "Final Specification" for Prolog III [31], the difficulty of dealing with nonlinear arithmetic, and arbitrary list concatenations (essentially word unification) was acknowledged. The problem was dealt with through syntactic restrictions. That is, a product of two variables was syntactically disallowed, as was list concatenation where the first list was a variable and its length was not explicitly given. This had the advantage of leaving the operational semantics very clean, but restricting the way programs could be written. This methodological issue will be discussed at considerable length later. However, Version 1.1 of Prolog III deals with both situations by delaying the constraints until sufficient information is available. In the case of list concatenation, this complicates matters somewhat. Since the length of a list variable may be specified by annotation with a numeric variable, list constraints and arithmetic constraints can affect each other. This use of delay mechanisms was pioneered in the work on CLP($\mathcal{R}$) described in this thesis.

Given the restriction on list concatenation, its implementation in Prolog III is straightforward. Arithmetic is handled using a simplex algorithm on an infinite precision representation of rational numbers — *i.e.*: pairs of unbounded integers. Version 1.1 also implements floating point arithmetic, with some guarantees of precision, but the algorithm has not been published. In many respects the constraint solving and output algorithms differ radically from those in CLP($\mathcal{R}$). Booleans were originally solved using SL-resolution, but a Davis-Putnam solver has recently been experimented with. Prolog III is available as a commercial system.

CHIP

CHIP (Constraint Handling in Prolog) [46, 183] was designed to tackle constrained search problems, which are typically NP-complete. Its approach is strongly influenced by the operations research and classical constraint satisfaction viewpoints. CHIP operates on somewhat disjoint domains: finite trees, natural numbers, rational numbers,

booleans and finite sets of constants or natural numbers.

The novel aspect of the language is how it deals with the finite sets. That is, classical constraint satisfaction problems are encoded as sets of constraints over variables whose values are known to lie within these finite sets. Rather than using the temporal backtracking of Prolog, which leads to a generate-and-test search, the forward checking, lookahead and partial lookahead rules of classical CSP are used. These are known as consistency techniques [112]: *a priori* pruning, using constraints to reduce the search space before discovering a failure.

Variables that take their values from finite sets are known as "domain" variables — not to be confused with the generic use of the term "domain". Natural number terms consist of natural numbers, domain variables over natural numbers, and terms built up from the usual natural number operators. Consistency techniques are used to solve equations, inequalities and disequalities on natural number terms, symbolic constraints on domain variables, and user-defined constraints. The user declares certain predicates to be suitable as constraints for forward checking purposes if they satisfy a certain calling pattern. Then these are used in addition to primitive constraints for pruning the search space.

CHIP has been used for a large number of classical operations research problems with remarkable success. The implementation of a CHIP compiler was described by Aggoun and Beldiceanu in [1], and this provides a useful contrast with the design of the Constraint Logic Arithmetic Machine (CLAM) for CLP($\mathcal{R}$) compilation as described in this thesis. In particular, the CHIP compilation model does not allow for many low level optimizations of constraint solving to be expressed.

CAL

CAL [151] is actually a pair of languages using Gröbner bases techniques to solve constraints. Boolean CAL uses a special modification of the Buchberger algorithm for boolean constraints. In real CAL, polynomial equations are solved using the classical Buchberger algorithm for complex numbers. This is rather problematic, since unsatisfiability in complex numbers implies unsatisfiability in real numbers, but the converse does not hold. It is not satisfactory to simply say that it is a CLP language on complex numbers, however. In particular, the applications that the authors wish to address include robotics, where the existence of imaginary solutions is far from useful in many cases.

RISC-CLP(R)

Another system dealing with nonlinear arithmetic constraints, based on partial cylindrical algebraic decomposition and Gröbner bases is described by Hong [75]. The above concern about complex numbers is avoided by using the unsatisfiability in the complex domain only for pruning. This work is only preliminary, and the practicality of the approach has not yet been established.

BNR Prolog

BNR Prolog [133] was developed by Bell Northern Research in Canada. It provides arithmetic constraints on closed intervals of the real line. The motivation is to make real relational arithmetic sound, which it is usually not when floating point arithmetic is used. An unconstrained variable begins as the entire real line (or really bounded by the highest and lowest machine real) and constraints result in this interval being narrowed. It is possible that an interval will be narrowed to a point. To maintain soundness, intervals are always rounded "outwards" when an arithmetic operation is performed. Iterative solution techniques are used for nonlinear constraints.

Trilogy

Trilogy [103, 185, 186] has an operational model that is substantially divorced from logic programming. Prolog-like rules are combined with procedures of conventional programming paradigms. Linear equations and inequalities are solved in the domain of integers (Pressburger arithmetic). Mode declarations are used. It is available cheaply as a commercial product for personal computers.

λProlog and Elf

For our purposes, both λProlog [126] and Elf [139] can be seen as solving equations between terms of the λ-calculus with simple types and dependent types, respectively. The motivation for λProlog was meta-programming for both programming languages and logics. It is particularly suited to this application as variables in an object language can be represented using abstraction in the meta language. Elf does not have predicate variables of λProlog, but programs may reason about their partial correctness by manipulating proofs of sub-goals in a way that is guaranteed to be type correct.

CLP(Σ^*)

The domain of CLP(Σ^*) [187] is regular sets. The language is intended to provide a logic-based formalism for incorporating string handling and even file handling into logic programming. The author also points out that problems in computer security can be solved by reasoning with regular expressions. The atomic constraints of the language are of the form x in e where x is either a variable or a string, and e is a regular expression possibly containing variables. Where such a constraint can have multiple solutions, these are dealt with through backtracking. This brings up issues of scheduling, since constraints may have infinitely many solutions. The scheduling strategy allows only constraints capable of generating a finite set of solutions to be selected from a conjunction for evaluation. That is, others are essentially delayed. This ensures termination of unsatisfiable conjunctions.

CIL

CIL (Complex Indeterminate Language) [124, 125] is designed for natural language processing. It is a CLP language over partially tagged trees (PTTs). The terms are called Partially Specified Terms (PSTs), which have been described [3] as being exactly the same as feature matrices used in Unification Grammars [159]. CIL incorporates the `freeze/2` predicate of Prolog II.

LOGIN and LIFE

LIFE [3] is a language consisting of logic programming, functional programming and a facility for structured type inheritance — the ψ-calculus. The logic programming component together with the ψ-calculus is a CLP language over what are called ψ-terms, and the subtyping and type intersection operations of the ψ-calculus. This language is called LOGIN [4]. A ψ-term is essentially a data structure built out of constructors and access functions, and subject possibly to some equational constraints that reflect sharing of structure. In this way the calculus formalizes data structure inheritance.

CLPS

The CLP Shell, described by Lim and Stuckey [109] is an interesting and potentially valuable alternate approach to the systems discussed above. The objective of CLPS is to allow multiple constraint solvers to be incorporated into a CLP system with minimum effort. However, it remains to be shown that the overhead resulting from this loose coupling of constraint solvers is not excessive. The promise stems from the fact that the system may be useful for prototyping if the overhead is less than say an order of magnitude.

Concurrent Constraint Programming, and the Ask/Tell Framework

Concurrent Constraint (CC) programming [152] grew out of the work on committed choice logic programming languages [156], such as Concurrent Prolog [155]. These languages differ from normal logic programming in that they don't have the usual "don't know" nondeterminism. Instead, they use what is called "don't care" nondeterminism. That is, all clauses for a given predicate are tried concurrently, until one of them reaches a commit operation, at which time the other alternatives are discarded. Committed choice languages originally depended on complicated and cumbersome methods of synchronization. The idea of concurrent constraint programming was to deal with this problem by making synchronization dependent on whether some constraint was entailed by a global constraint store. Such constraints are "Ask" constraints. This global store is maintained by combining constraints asserted throughout the program – "Tell" constraints. The view of constraints in CC languages is rather different from that of this thesis, because entailment rather than (un)satisfiability is used to control execution. This view of constraints gives rise to an interesting and challenging set of

design and implementation problems quite different from the ones discussed in this thesis.

2.3.3 Applications

In recent years, considerable work has been done on applications of various CLP languages. One of the major application areas has been electrical circuit analysis, synthesis and diagnosis [59, 123, 163, 165, 164, 166, 162, 167]. Civil engineering [102] and mechanical engineering [173] have also attracted some attention. Engineering applications tend to combine hierarchical composition of complex systems, mathematical or boolean models, and — especially in the case of diagnosis and design — deep rule-based reasoning. Another major area has been that of options trading [74 78, 79, 104] and financial planning [12, 13, 14, 19]. These applications have tended to take the form of expert systems involving mathematical models. More traditional operations research problems have included cutting stock [44] and scheduling [45].

More exotic applications areas have included restriction site mapping in genetics [192]. Levitt surveyed the use of constraint languages in music [107], and Tobias implemented an expert system for tutoring about harmony in CLP($\mathcal{R}$) [182]. CLP($\mathcal{R}$) was also used for generating test data for communications protocols [57]. Prolog was developed by the Marseille group for natural language processing, so it should come as no surprise that this is turning out to be significant application area for LIFE [3], CIL [125], Prolog III [31] and λProlog [120, 34]. Both λProlog [62] and Elf [117] have been used for implementing natural semantics of programming languages. λProlcg has also been used for implementing natural deduction-style theorem provers [52]. Applications of CLP($\mathcal{R}$), λProlog and Elf will be discussed in more detail in Part II.

Part II

Languages and Programming

Chapter 3

Language Design

To design a CLP language, it is necessary to choose a domain of computation. Recall that choosing a domain involves choosing the underlying set of values with which we are computing, the expressions (operations) allowed on that set, and the constraints allowed on variables ranging over the set. As we shall see, the major theme of CLP language design is introducing suitable compromises, based on complications resulting from implementation issues. Furthermore, it is necessary for compromises to be introduced to the operational model as well.

It is also important to consider what minimum domain is necessary to provide enough utility to solve the required class of problems. In particular, most CLP languages should probably include uninterpreted functors, thus at least nominally making Prolog a subset. This makes a language more amenable to "general purpose" programming, providing at least basic data structuring mechanisms. As a medium term consideration, in order for CLP to be adopted into the applications community, it will be necessary to promote the languages as extensions of Prolog, which run Prolog programs essentially without modification, and with more or less the usual efficiency.

For the purposes of the following discussion, we define a CLP program as being a finite set of rules of the form

$$A :- \gamma_1, \cdots, \gamma_n, \quad n \geq 0$$

where A is an atom and each γ_i is either a constraint or an atom. An atom consists of a predicate symbol with the appropriate number of terms as arguments, and a constraint is a constraint symbol with the appropriate number of terms as arguments. A term is built up from variables and constants and function symbols from the domain.

3.1 CLP Operational Model

We begin by formally defining a CLP operational model, with enough generality to account for compromises motivated by implementation concerns. Inherent in the operational model of CLP is a *subgoal selection strategy*, which selects a delayed constraint or an atom from a given goal. Now let P denote a CLP program. Let G denote a

goal consisting of a sequence $\gamma_1, \cdots, \gamma_l$ and a function partitioning this sequence into multisets Λ of atoms, Σ of solved constraints and Δ of delayed constraints. In an initial goal, all the constraints are delayed (in Δ). We say that there is a *derivation step* from G to another goal G_2 if the subgoal selection strategy selects some γ_i for $1 \leq i \leq l$ and one of the following holds:

- $\gamma_i \in \Delta$ and $\Sigma \cup \{\gamma_i\}$ is satisfiable.
  Then G_2 is the sequence G with partition $\Lambda_2 = \Lambda$, $\Delta_2 = \Delta \setminus \{\gamma_i\}$ and $\Sigma_2 = \Sigma \cup \{\gamma_i\}$.

- $\gamma_i \in \Lambda$ and the program P contains a rule R that can be renamed so that it contains only new variables and takes the form: the head atom is H, the body is the sequence denoted by $B_1, B_2, \ldots, B_s$, $s \geq 0$, partitioned similarly into multisets Λ_1 and Δ_1.
  Then G_2 is the sequence $\gamma_1, \ldots, \gamma_{i-1}, \gamma_i = H^1, B_1, \ldots, B_s, \gamma_{i+1} \ldots, \gamma_n$, with the partition $\Sigma_2 = \Sigma$, $\Delta_2 = \Delta \cup \Delta_1 \cup \{\gamma_i = H\}$ and $\Lambda_2 = (\Lambda \setminus \{\gamma_i\}) \cup \Lambda_1$.

We say that γ_i in G above is the *selected subgoal.* Equivalently, γ_i is the subgoal of G chosen to be *reduced.*

A *derivation sequence* (or simply sequence) is a possibly infinite sequence of goals, starting with an initial goal, wherein there is a derivation step to each goal from the preceding goal. A sequence is *successful* if it is finite and its last goal contains only solved constraints. A sequence is *conditionally successful* if it is finite and its last goal contains only solved and delayed constraints. Finally, a *finitely failed* sequence is finite, neither successful nor conditionally successful, and such that no derivation step is possible from its last goal[2]. The constraints in the last goal of successful and conditionally successful sequences are called *answer constraints*, and they constitute the output of CLP programs.

Thus a program and goal are executed by a process of continuously reducing any remaining atoms or solving delayed constraints in the goal. There are two nondeterministic aspects in obtaining a derivation sequence from a given initial goal as we have defined above. One pertains to the subgoal selection strategy, The other pertains to the *search strategy*, which determines which rule is used next in the reduction of a selected atom in a subgoal.

As an example, consider the following program solving real arithmetic constraints.

```
ohm(V, I, R) :- V = I * R.
```

The following, in which we indicate solved constraints by underlining them, illustrates a successful derivation sequence:

[1] This abbreviates the collection of equations between corresponding arguments in γ_i and H.

[2] If the subgoal selection strategy is *fair* in the sense that every delayed constraint and atom in an infinite sequence is eventually selected, then one has the usual soundness and completeness results with respect to successful and finitely failed sequences [82].

```
?- ohm(V1, I, R1), ohm(V2, I, R2), V = V1 + V2,
   R1 = 15, R2 = 5.

?- V1 = V', I = I', R1 = R', V' = I' * R',
   ohm(V2, I, R2), V = V1 + V2, R1 = 15, R2 = 5.
   ⋮
?- V1 = V', I = I', R1 = R', V' = I' * R',
   ohm(V2, I, R2), V = V1 + V2, R1 = 15, R2 = 5.

?- V1 = V', I = I', R1 = R', V' = I' * R',
   V2 = V'', I = I'', R2 = R'', V'' = I'' * R'',
   V = V1 + V2, R1 = 15, R2 = 5.
   ⋮
?- V1 = V', I = I', R1 = R', V' = I' * R',
   V2 = V'', I = I'', R2 = R'', V'' = I'' * R'',
   V = V1 + V2, R1 = 15, R2 = 5.

?- V1 = V', I = I', R1 = R', V' = I' * R',
   V2 = V'', I = I'', R2 = R'', V'' = I'' * R'',
   V = V1 + V2, R1 = 15, R2 = 5.

?- V1 = V', I = I', R1 = R', V' = I' * R',
   V2 = V'', I = I'', R2 = R'', V'' = I'' * R'',
   V = V1 + V2, R1 = 15, R2 = 5.

?- V' = V1, V'' = V2, I = I' = I'', R' = R1, R'' = R2,
   V1 = 0.75*V, I = 0.05*V, R1 = 15, V2 = 0.25*V, R2 = 5.
```

Note that a "left-to-right" subgoal selection strategy was used above, except when the nonlinear delayed constraint V' = I' * R' was leftmost. The next leftmost and linear constraint was selected instead. In the fourth last goal, the constraint R1 = 15 became a solved constraint and so the delayed nonlinear constraint V' = I' * R' becomes linear. As will be discussed later in this chapter, this was no accident. It is then selected next. In the third last goal above, R2 = 5 was selected, thus allowing the other nonlinear constraint V'' = I'' * R'' to be selected next. Finally, note that we have written the last goal in a simplified form.

3.2 Complications Related to Deciding Constraints

In reality, a number of considerations complicate the language design process. Much of this thesis deals with these complications, and some of the techniques for dealing with them to produce "practical" programming languages and systems. For many seemingly desirable domains, the required decision algorithms simply do not exist. For others, the

decision problem may be open. It is then necessary to decide whether undecidability is expected to be a problem in practice. For many more domains, either the decision problem is known to be intractable, or the best known algorithms are impractical. Even when a reasonably (or very) efficient decision procedure exists, it may be incompatible with the CLP operational model. In particular, a CLP implementation requires incremental satisfiability testing to be efficient. That is, it must be possible to determine efficiently whether a satisfiable set of constraints, augmented with a new constraint, is still satisfiable. Efficiency here loosely means that the time should be proportional more to the size of the added constraint than that of the previous, satisfiable, set. Furthermore, because of backtracking, it must be possible to undo such augmentations of the constraint set efficiently.
To make the above more concrete, let us consider a number of examples.

- Natural numbers and integers are useful domains for many applications of an operations research flavor. Nonlinear polynomial equations over integers, known more commonly as diophantine equations, are undecidable. Linear equations and inequalities are decidable but expensive to decide.

- Equations between simply-typed λ-expressions are undecidable.

- Real polynomial equations and inequalities are decidable, but all known algorithms are extremely inefficient. Real linear equations and inequalities are efficiently decidable: polynomial algorithms are known, and some are even known to be efficient in practice.

- A linear algorithm for unification of finite trees is available: one was described by Paterson and Wegman in [136] However, no linear unification algorithm is known to perform well in practice. It is instructive to notice that the usual (Robinson) algorithm for unification of finite trees, as implemented in most Prolog systems, is quite incremental in practice. The cost of backtracking is linear in the number of variables instantiated since the last choicepoint, and unification cost is only somewhat dependent on the number of prior constraints (as instantiated variables are more costly to unify than free ones, since they have to be traversed). The Robinson algorithm is, depending on how terms and unifiers are represented, either exponential or polynomial in the worst case.

3.3 Effect on Language Design

Issues of incremental satisfiability are discussed further in Chapter 9. Where an efficient incremental solver is not readily available, the problems discussed above can often be dealt with by appropriate restriction and organization of constraint solvers. To apply these techniques, the design of a CLP language must begin with an intended application area (or areas) and programming style. If constraint solving is to be restricted, certain kinds of constraints must be identified as being too difficult to solve in the general case. In this thesis they will be called the *hard* constraints. We now consider various ways

to deal with hard constraints, given that the domain of computation is general enough that such constraints can be expressed in the language.

3.3.1 Syntactic Restrictions

One approach is to syntactically restrict the kinds of constraints on the given domain that can be expressed. Note that the domain itself specifies the operations and relations on the underlying set that are allowed. The syntactic restrictions here determine what expressions can be built up using the operators, and what expressions the various relation symbols can be applied to. Let us consider some examples.

- Arithmetic expressions can be restricted to being linear, by requiring that multiplication only occur between two constants or between a constant and a variable, but not between two variables.
- Simply-typed λ-expressions can be restricted essentially such that, for example, one existentially quantified variable cannot be applied to another. The detailed definition of the restricted language, and the significance of the restriction, will be discussed in Chapter 5.
- Word equations can be restricted to avoid the case where the length of an initial variable is unknown.
- Consider linear equations and inequalities on reals or rationals. Shostak [161] gave a decision algorithm based on computing loop residues that is particularly efficient when there are no more than two variables per constraint (note that three variables per constraint are sufficient to express any set of constraints).

The syntactic restrictions should be such that they still enable the programmer to express what is needed. As a result, not all problems of undecidability or inefficiency can be handled in this way. One reasonable alternative is to insist that the expressible constraints should at least be decidable. However, even this restriction is sometimes excessive, as is the case with λProlog and Elf, which will be discussed in Chapters 5 and 11. At the very least, the language should be restricted sufficiently as to make the *dynamic* restriction techniques discussed below applicable and useful. It turns out that syntactic restriction is usually an excessively brutal way to obtain an efficiently implementable programming language. This is because constraints that appear syntactically complex can often be simplified considerably by the time they are selected at runtime. These observations will be discussed in more detail in the next two chapters.

3.3.2 Dynamic Restrictions

In the philosophy of constraint programming it is important that the programmer should not have to be concerned with when information becomes available but just needs to provide *enough* constraints for it *eventually* to become available. A more effective way to cut the constraints down to size, then, is to restrict them dynamically.

That is, impose conditions on the form of a constraint when it is selected, rather than when it is parsed. By the time a constraint is selected, some of its variables may have been involved in a number of other constraints. Thus enough may be known about them that deciding the constraint has become much simpler than might be expected by examining it syntactically, in isolation. For example, a nonlinear arithmetic equation might be made linear by the grounding of one or more variables. Or an equation between two simply-typed λ expressions might similarly become deterministic. Of course, we should note that this discussion hinges on the language designer having a clear idea about the programming methodology that the language is intended to support. This issue will be discussed in some detail in the next two chapters.

There are two possible ways to deal with the situation where a constraint is selected and does not meet the appropriate syntactic requirements.

- *Runtime error*
  Halt execution of the program, and print an error message, hopefully giving information on how the restriction was not satisfied.

- *Delay*
  Assume, for the time being, that the constraint is consistent with the collected constraints, but do not solve it – just store it. Then if it later *does* satisfy the restriction, add it, and backtrack if it is inconsistent.

The first approach is the usual way to deal with system predicates in Prolog. For example, in most Prolog systems,

```
?- X = 1, Y = 3, Z is X + Y.
```

binds `Z` to `4`, while

```
?- X = 1, Z is X + Y, Y = 3.
```

will cause a runtime error when `Z is X + Y` is selected, as `Y` is still free. This is certainly an improvement on static restrictions. For example, in a CLP language dealing with reals, consider the following constraints with a left-right atom selection rule.

```
?- I1 + I2 = 10, I1 - I2 = 0, V = I2 * R + 2 * I2 * R.
```

By the time the third constraint is selected, `I1` and `I2` have values, so the equation is linear. However, such a restriction still places on the programmer the burden of worrying about *when* information becomes available, rather than just that of giving *enough* information.

Delay mechanisms have quite a long history in logic programming: a systematic study by Naish can be found in [128]. They have most commonly been used to delay the calling of certain subgoals, either by annotating the subgoal with a delay condition, or by annotating the rules defining a given predicate with the condition. Prolog II's **`freeze/2`** predicate is an example of the former, and the **`wait`** declarations of MU-Prolog and **`when`** declarations of Nu-Prolog are examples of the latter. MU-Prolog and

Nu-Prolog also use delay on certain system predicates, such as a special predicate for multiplication by local propagation. Carlsson [22] described how such general delay mechanisms could be implemented efficiently. To see the advantages of delay mechanisms for constraints, consider a permuted version of the example above, and assume that nonlinear equations and inequalities are delayed until they become linear.

```
?- V = I2 * R + 2 * I2 * R, I1 + I2 = 10, I1 - I2 = 0.
```

Here the first equation can simply be delayed when selected, and when the values of `I1` and `I2` are obtained from the subsequent constraints, it can be "awakened". Deeper examples of this phenomenon in real programs will be discussed in Chapter 4. As an example in a different domain, let us consider word equations. The goal

```
?- X.Y = abcde, Y = c.Z .
```

has the unique answer

```
X = ab, Y = cde, Z = de
```

But if constraints are delayed until the variable on the left of a concatenation is of known length, the constraints will not produce an answer. However, the goal

```
?- X.Y = abcde, Y = c.Z, |Z| = 2.
```

can obtain the answer using delay as follows.

- After the first constraint is delayed, additional constraints are established:

  ```
  |X| + |Y| = 5, |X| <= 5, |X| >= 0
  ```

- After the second constraint is encountered, the first cannot yet be awakened but additional (arithmetic) constraints are established:

  ```
  |X| + |Z| = 4, |X| <= 4, X.c.Z = abcde
  ```

- Finally, the addition of the third constraint makes it possible to infer the constraint

  ```
  |X| = 2
  ```

  so the first constraint can be awakened, binding `X` to `ab` and `Y` to `cde`.

The interested reader can find similar examples for CLP(Σ^*) in [187].

Delay gives the programmer considerable flexibility, but also additional responsibility. For a start it is now possible to obtain deadlock — a situation where only delayed constraints remain, but none meets the selection criteria. This might be considered an error condition, or the notion of termination might be extended to include the possibility of a deadlocked computation, where the answer to the query is **maybe**. The answer

constraint could then be printed, with its correctness being contingent on the (undetermined) satisfiability of the delayed constraints. Additional problems of programming methodology arise when this approach is chosen. In particular, the programmer is not *really* freed from the burden of understanding *when* information becomes available. Where constraint (un)satisfiability is used to control search, an inconsistency that is detected too late can lead to an infinite loop, both in cases when the query should have succeeded, and in cases where it should have failed finitely. Such a delay mechanism can thus lead to rather insidious bugs in a program. These problems are partially illustrated by the following definitions of the predicates `silly` and `dangerous`. We again assume that nonlinear equations and inequalities are delayed until enough information is available to make them linear.

```
silly(X) :- X*X < 10, X > 4, loopalways(X).

dangerous(X) :- X*X < -5, loopalways(X).
dangerous(3).
```

Any call to `silly` should simply fail, since the two constraints are inconsistent. However, if `X` is free in the call, the first constraint will be delayed, and the second does not produce enough information to awaken it. Hence the call to `loopalways` will happen, producing an infinite loop. A call to `dangerous` with a free value of `X` should succeed, binding `X` to 3, as the constraint is unsatisfiable. However, such a call will result in the constraint being delayed, thus ending up in an infinite loop. Of course, it should be pointed out that the problem we are describing is not new: the operational model of Prolog itself introduces very similar problems.

The issue of how to restrict a CLP language appropriately has often come up and been addressed in different ways in real systems. As will be discussed in some detail in Chapter 4, in CLP($\mathcal{R}$) the selection rule is modified to delay nonlinear constraints until they become linear. In the early documents on Prolog III on the other hand, including the so-called "Final Specification" [31], the syntactic approach to this same problem was strongly advocated, although the claims of this thesis about the methodological advantages of delay, had already been made. However, recent commercial versions of Prolog III employ delay for nonlinears also. Furthermore, the same approach is now used in Prolog III to deal with the intractability of word unification. A word equation is delayed until the length of any initial variables is known. (In fact, at the implementation level, it delays them until it is known whether the length of the leading variable is zero or non-zero [33]. As will be discussed in subsequent chapters, a similar argument has occurred over how the non-determinism of higher-order unification should be handled. Likewise, a delay approach could be applied to constraints on data aggregates, such as arrays. These examples will be discussed in more detail in Chapter 7.

It should be noted that even for applications that can run correctly with certain dynamic restrictions, these might have an adverse effect. For example, in Chapter 4 a crypto-arithmetic program is discussed that uses linear inequalities heavily. If only equations were decided, and inequalities were delayed until ground, this program would still run correctly. However, an important trade off exists in terms of efficiency: deciding

linear inequalities causes earlier pruning of the search space, but is considerably more costly per constraint. Thus which approach is faster actually depends on the number of constraints pruned versus the cost of solving an inequality.

Delay mechanisms, of course, bring their own implementation difficulties, although delaying constraints is not as problematic as delaying general calls, as described in [22]. For the operational model to be reasonably intuitive for the programmer, it is necessary to awaken delayed constraints as soon as possible after they become sufficiently simple. However, on backtracking, an awakened constraint must be resuspended. Naturally, backtracking complicates any indexing structures that are used for timely awakening. Of course there is an overhead associated with checking for awakened constraints whenever the constraint store is augmented, but this can usually be kept low. These issues are discussed in detail in Chapter 7.

3.4 Looking Ahead

The role of the next two chapters, about language design and programming methodology aspects of CLP($\mathcal{R}$), λProlog and Elf, make the assertions of this chapter at a more concrete level. All three languages are now well-enough understood and are used widely enough that some quite specific comments can be made about the interaction between language design and programming methodology. One of the claims of this thesis is that the pragmatic approach to designing CLP languages through restricting constraint solving by delay provides the right compromise between expressive power and efficiency.

Chapter 4

CLP($\mathcal{R}$)

The design and implementation of a CLP language for real arithmetic was essentially motivated by two observations:

- Real arithmetic is of fundamental importance in computer science, simply because of the demands of applications.
- Historically, many declarative programming languages have not dealt well with arithmetic.

In particular, real arithmetic was grafted onto most Prolog systems at an early stage, but in a form that ran counter to the philosophy of logic programming. This was usually in the form of the two-argument `is` predicate, which evaluated the arithmetic expression in its second argument, expecting all the variables in the expression to have already been bound to numbers, and assigning the resulting value to the free variable in the first argument. That is, arithmetic was essentially handled in the same way as in imperative languages. The following discussion introduces CLP($\mathcal{R}$), a systematic attempt to use the CLP framework to produce a language with real arithmetic that maintains the advantages of the logic programming framework. This is followed by an extensive discussion of programming methodology, together with empirical observations about the kinds of constraint solving that are required in typical programs. A discussion of some relatively large scale applications is included.

4.1 The Language and Operational Model

We now describe the syntax of CLP($\mathcal{R}$) programs and their intuitive semantics. Then we describe the operational model of the CLP($\mathcal{R}$) system, an approximation to the model prescribed by the CLP scheme as described in Chapter 3.

4.1.1 The Structure $\mathcal{R}$

Essentially, $\mathcal{R}$ is a two-sorted structure where one sort is the real numbers, and the other sort is the set of trees over uninterpreted functors and real numbers. The construction

of trees using uninterpreted functors is the only tree operation. The only real number operations are addition and multiplication. While equality is the only relation between trees, both equality and inequality are defined on numbers. The truth of these relations is defined in the obvious way: two trees are equal iff their root functors are the same and their corresponding subtrees, if any, are equal. The truth of real number relations is defined in the standard way.

We now proceed with the syntactic formulation of $\mathcal{R}$. In what follows, we shall assume tacitly, to avoid further formality, that the language CLP($\mathcal{R}$) is (statically) typed in the sense that the type (real number or functor) of each argument of each predicate symbol and functor, as well as the type of each variable, is pre-determined. In CLP($\mathcal{R}$) programs, therefore, each predicate symbol, functor and variable is used in a consistent way with respect to its type.

4.1.2 CLP($\mathcal{R}$) Constraints

Real constants and real variables are both *arithmetic terms*. If t_1, t_2 are arithmetic terms, then so are $(t_1 + t_2)$, $(t_1 - t_2)$ and $(t_1 * t_2)$. Uninterpreted constants and functors are like those in Prolog. Uninterpreted constants and arithmetic terms are *terms*, and so is any variable. The rest of the terms are defined inductively as follows: if f is an n-ary uninterpreted functor and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.

If t_1 and t_2 are arithmetic terms, then $t_1 = t_2$, $t_1 < t_2$ and $t_1 \leq t_2$ are all arithmetic *constraints*. If, however, not both the terms t_1 and t_2 are arithmetic terms, then only the expression $t_1 = t_2$ is a constraint. Both these kinds of constraints will be used in programs, and they form a subset of all the predicates that may appear in a program. Because these constraints have pre-defined meanings, we shall sometimes emphasize this by calling them *primitive constraints* (or simply constraints when confusion is unlikely). A primitive constraint is *solvable* iff there is an appropriate assignment of real numbers and ground terms to the variables therein such that the constraint evaluates to true.

In what follows, we adopt the (Prolog) convention of using strings beginning with capital letters to denote variables.

4.1.3 CLP($\mathcal{R}$) Programs

An *atom* is of the form $p(t_1, t_2, \ldots, t_n)$ where p is a predicate symbol distinct from $=$, $<$, and $\leq$, and $t_1, ..., t_n$ are terms. A *rule* is of the form

$$A_0 :- \alpha_1, \alpha_2, \ldots, \alpha_k.$$

where each α_i, $1 \leq i \leq k$, is either a primitive constraint or an atom. The atom A_0 is called the *head* of the rule while the remaining atoms and primitive constraints are known collectively as the *body* of the rule. In case there are no atoms in the body, we may call the rule a *fact* or a *unit rule*. In case there are no atoms and constraints in the body, we may abbreviate the rule to be simply of the form

$$A_0.$$

Finally, a CLP($\mathcal{R}$) program is defined to be a finite collection of rules.

Thus rules in CLP($\mathcal{R}$) have much the same format as those in Prolog except that primitive constraints may appear together with atoms in the body. The same applies to a CLP($\mathcal{R}$) *goal*; this is of the form

$$?- \alpha_1, \alpha_2, \ldots, \alpha_k.$$

where each $\alpha_i, 1 \leq i \leq k$, is either a primitive constraint or an atom. A sub-collection of the atoms and constraints in a goal is sometimes called a *subgoal* of the goal.

The following is a CLP($\mathcal{R}$) program for computing Fibonacci numbers:

```
fib(0, 1).
fib(1, 1).
fib(N, X1 + X2) :-
      N > 1,
      fib(N - 1, X1),
      fib(N - 2, X2).
```

Its declarative semantics is clear. A goal that asks for a number `A` such that `fib(A)` lies in between 80 and 90 is

```
?- 80 <= B, B <= 90, fib(A, B).
```

and, as will be shown next, the answer obtained when the program and goal are executed is `A` = 10, `B` = 89.

We conclude this subsection with another example program:

```
available_res(10).
available_res(14).
available_res(27).
available_res(60).
available_res(100).

available_cell(10).
available_cell(20).

ohm(V, I, R) :- V = I * R.

sum([], 0).
sum([H | T], N) :- N = H + M, sum(T, M).

kirchoff(L) :- sum(L, 0).
```

The program, together with the goal below, represents a simple preferred value problem in which we have an electrical circuit template and can choose from a finite set of

components to select those that make the circuit satisfy a certain property. In this example, the simple circuit has two resistors connected in series:

```
?- 14.5 < V2, V2 < 16.25,
   available_res(R1), available_res(R2),
   available_cell(V),
   ohm(V1, I1, R1), ohm(V2, I2, R2),
   kirchoff([I1, -I2]), kirchoff([-V, V1, V2]).
```

The three answer constraints for the program and goal are as follows.

```
14.5 < V2, V2 < 16.25,
V1 / 10 - V2 / 27 = 0,
V1 + V2 = 20

14.5 < V2, V2 < 16.25,
V1 / 14 - V2 / 60 = 0,
V1 + V2 = 20

14.5 < V2, V2 < 16.25,
V1 / 27 - V2 / 100 = 0,
V1 + V2 = 20
```

4.1.4 The Operational Model

The operational model presented in Chapter 3 is rather general: unnecessarily so for this discussion. In discussing CLP($\mathcal{R}$), we specialize the operational model by specifying a left-right subgoal selection strategy for atoms and delayed constraints. That is, considering atoms and delayed constraints as one sequence, these are solved one by one from left to right. Recall that in the general CLP operational model, all constraints are initially considered delayed. Selected constraints that are considered hard (in CLP($\mathcal{R}$), the nonlinear arithmetic constraints) remain delayed after selection, but are re-selected as soon as they cease to be hard, in preference to other delayed constraints. This specialized operational model suggests a rather different presentation, as given here. This presentation corresponds more readily to the programmer's mental view of the execution of programs.

Now let P denote a CLP($\mathcal{R}$) program. Let G denote a goal

$$\Sigma, \Delta \ ?- \quad A_1, A_2, \ldots, A_n, \quad n \geq 0.$$

where Σ is a set of solved constraints, Δ is a set of delayed constraints, the A_i are either atoms or constraints. We say that there is a *derivation step* from G to another goal G_2 if one of the following holds:

- Some $\delta \in \Delta$ is not hard with respect to Σ, and $\Sigma \cup \{\delta\}$ is solvable. Then G_2 is of the form

$$\Sigma \cup \{\delta\}, \Delta \setminus \{\delta\} \ ?- \quad A_1, A_2, \ldots, A_n.$$

- The first condition does not hold, A_1 is a constraint, and is hard with respect to Σ. Then G_2 is

$$\Sigma, \Delta \cup \{A_1\} \ ?- \quad A_2, \ldots, A_n.$$

  Of course, Σ remains solvable.

- The first condition does not hold, A_1 is a constraint, is not hard with respect to Σ, and $\Sigma \cup \{A_1\}$ is solvable. Then G_2 is

$$\Sigma \cup \{A_1\}, \Delta \ ?- \quad A_2, \ldots, A_n.$$

- The first condition does not hold, and A_1 is an atom. The program P contains a rule R that can be renamed so that it contains only new variables and takes the form

$$B \ :- \ B_1, B_2, \ldots, B_s, \quad s \geq 0.$$

  Then G_2 is

$$\Sigma, \Delta \ ?- \quad A = B, B_1, \ldots, B_s, A_2, \ldots, A_n$$

  where $A = B$ abbreviates the collection of equations between corresponding arguments in A and B. Of course, Σ remains solvable.

In the initial goal, $\Sigma = \Delta = \{\}$.

A *derivation sequence* (or simply sequence) is a possibly infinite sequence of goals, starting with an initial goal, wherein there is a derivation step to each goal from the preceding goal. A sequence is successful if it is finite and its last goal is $\Sigma, \{\}$?- . A sequence is conditionally successful if it is finite and its last goal is Σ, Δ ?- where Δ is nonempty. Finally, a finitely failed sequence is finite, neither successful nor conditionally successful, and such that no derivation step is possible from its last goal.

As an illustration, we return to the program and query used in Chapter 3, which happened to be in CLP($\mathcal{R}$).

```
ohm(V, I, R) :- V = I * R.
```

The successful derivation sequence is shown in Figure 4.1.

CLP($\mathcal{R}$) is an approximation to the operational model of the CLP scheme for four main reasons:

- To solve equations between non-arithmetic terms, an algorithm similar to a conventional unification algorithm is used. No "occurs check" is done, as this issue is largely orthogonal to those of interest.

- As has already been mentioned, a left-right subgoal selection strategy is used, and nonlinear arithmetic constraints are delayed until they become linear. CLP($\mathcal{R}$) employs a "top-to-bottom" depth-first search strategy in the generation of derivation sequences.

```
{}, {}
?- ohm(V1, I, R1), ohm(V2, I, R2), V = V1 + V2, R1 = 15, R2 = 5.
{}, {}
?- V1 = V', I = I', R1 = R', V' = I' * R',
ohm(V2, I, R2), V = V1 + V2, R1 = 15, R2 = 5.
      ⋮
{ V1 = V', I = I', R1 = R' }, { V' = I' * R' }
?- ohm(V2, I, R2), V = V1 + V2, R1 = 15, R2 = 5.
{ V1 = V', I = I', R1 = R' }, { V' = I' * R' }
?- V2 = V'', I = I'', R2 = R'', V'' = I'' * R'',
V = V1 + V2, R1 = 15, R2 = 5.
      ⋮
{ V1 = V', I = I', R1 = R', V2 = V'', I = I'', R2 = R'' },
{ V' = I' * R', V'' = I'' * R'' }
?- V = V1 + V2, R1 = 15, R2 = 5.
{ V1 = V', I = I', R1 = R', V2 = V'', I = I'', R2 = R'', V = V1 + V2 },
{ V' = I' * R', V'' = I'' * R'' }
?- R1 = 15, R2 = 5.
{ V1 = V', I = I', R1 = R', V2 = V'', I = I'', R2 = R'',
V = V1 + V2, R1 = 15 },
{ V' = I' * R', V'' = I'' * R'' }
?- R2 = 5.
{ V1 = V', I = I', R1 = R', V2 = V'', I = I'', R2 = R'',
V = V1 + V2, R1 = 15 V' = I' * R' },
{ V'' = I'' * R'' }
?- R2 = 5.
{ V1 = V', I = I', R1 = R', V2 = V'', I = I'', R2 = R'',
V = V1 + V2, R1 = 15 V' = I' * R', R2 = 5 },
{ V'' = I'' * R'' }
?-
{ V1 = V', I = I', R1 = R', V2 = V'', I = I'', R2 = R'',
V = V1 + V2, R1 = 15 V' = I' * R', R2 = 5, V'' = I'' * R'' }
{}
?-
{ V' = V1, V'' = V2, I = I' = I'', R' = R1, R'' = R2,
V1 = 0.75*V, I = 0.05*V, R1 = 15, V2 = 0.25*V, R2 = 5 },
{}
?-
```

Figure 4.1: Successful derivation sequence for ohm query

- A floating-point representation of real numbers is employed. A small and specifiable fixed tolerance is used in the comparisons of numbers.
- The constraint solver does not determine the solvability of all nonlinear constraints.

The first two points pertain to Prolog implementations as much as to CLP($\mathcal{R}$). The selection and search strategies are chosen simply for pragmatic reasons. In short, CLP($\mathcal{R}$) inherits the approaches to these two issues, as well as the occurs check, from usual Prolog implementation strategies. The third point, using floating-point numbers, clearly can give rise to unsoundness. However, our estimation is that alternative implementations (*e.g.* rational number or interval based methods) are unacceptably costly in a general-purpose system. Our experience over many application programs shows that the use of floating-point numbers has rarely been problematic, but there is certainly scope for a CLP system to be developed around some sound form of real arithmetic.

We now discuss the last point in the remainder of this section. As an example of delay in CLP($\mathcal{R}$), the following rule can be used for multiplying or dividing complex numbers:

```
c_mult(c(R1, I1), c(R2, I2), c(R3, I3)) :-
      R3 = R1 * R2 - I1 * I2,
      I3 = R1 * I2 + R2 * I1.
```

Any of the following goals will return a unique answer:

```
?- c_mult(c(1, 1), c(2, 2), Z).
?- c_mult(c(1, 1), Y, c(0, 4)).
?- c_mult(X, c(2, 2), c(0, 4)).
```

Execution of the first goal is performed by evaluation and assignment in the traditional way. The other two goals require the division of complex numbers, and the unique answers, one for each of the two cases, can only be obtained by the solution of (two) simultaneous linear equations. Thus none of the three goals above gives rise to any delayed constraints. The following goal, however, does:

```
?- c_mult(c(X, Y), c(X, Y), c(-3, 4)).
```

This is because the constraints obtained from expanding this goal are

```
X*X - Y*Y = -3,
2*X*Y = 4.
```

If, however, this goal also contained an atom or constraint whose expansion or solution grounds either X or Y, then a unique answer can be obtained. For example, with the rule

```
p(Y, Z) :- Y = 2*Z, Z = Y-1.
```

and goal

```
?- c_mult(c(X, Y), c(X, Y), c(-3, 4)), p(Y, Z).
```

the call to p gives a value to Y (grounds Y to 2), and so the goal can be executed by the CLP($\mathcal{R}$) system to return $X = 1, Y = 2, Z = 1$.

Clearly the delay/wakeup technique allows the easy introduction of many other (nonlinear) arithmetic functions. For example, it is a trivial matter to augment the CLP($\mathcal{R}$) system to include the functions *sin*, *cos*, *abs* and *pow*. We can define that the *abs* function is delayed until either the input can be evaluated as a ground value, or the vartiable equated to the output is ground and not positive. Similarly, the equation $X = pow(Y, Z)$, representing $X = Y^Z$, can be defined to awake upon the grounding of two of three variables X, Y, Z, or the grounding of Y to 0 or 1, or the grounding of Z to 0 or 1, etc.

4.2 Basic Programming Techniques

We begin this discussion with a number of quite simple programming examples. These will also serve to illustrate a range of CLP($\mathcal{R}$) programming techniques, and hence lay the foundations for a discussion of programming methodology and later implementation, as well as supporting earlier assertions about language design.

A primitive constraint in a rule typically represents a local property of the subproblem at hand. While these constraints represent the relationship between various parameters in a particular part of a problem, (for example, Ohm's law for a resistor in a circuit), we require *global constraints* to describe the way these parts interact (for example, the use of Kirchoff's current law in node analysis). In CLP($\mathcal{R}$), global properties of a problem are represented by means of rules, and the corresponding global constraints are those associated with the answer constraints of the program.

The most straightforward way to represent global properties in a program is by hierarchical composition. For example, a rule of the form

$$p(\ldots) \leftarrow \textit{constraints}\ ,\ p_1(\ldots),\ \ldots, p_n(\ldots).$$

can be used where $p_1, \ldots, p_n$ are definitions of largely independent parts of some system, and this rule combines these into a whole by sharing variables in the arguments and via the list of constraints. As a concrete example, consider the rule

```
resistor(V, I, R) :- V = I * R.
```

which describes the local property of a resistor's voltage/current relationship. Then the rules

```
parallel_circuit(V, I, R1, R2) :-
      I1 + I2 = I,
      resistor(V, I1, R1),
      resistor(V, I2, R2).
series_circuit(V, I, R1, R2) :-
      V1 + V2 = V,
      resistor(V1, I, R1),
      resistor(V2, I, R2).
```

combine two resistors to model their behavior in parallel and in series respectively. This is achieved in the first case by summing their currents and then using the common variable V to constrain the voltage drops across them to be equal. In the second rule, we have the dual situation — a shared variable for current and a constraint for voltage. One can build up more complicated systems by building a multi-level hierarchy of components. For example, to describe two parallel pairs of resistors in series we may write

```
parallel_series(V, I, R1, R2, R3, R4) :-
      V1 + V2 = V,
      parallel_circuit(V1, I, R1, R2),
      parallel_circuit(V2, I, R3, R4).
```

While the output of answer constraints is beyond the scope of this thesis, a lot of CLP($\mathcal{R}$) programming[1] is centered around producing such answers. Consider, for example, the following program, which relates the key parameters in a mortgage:

```
mortgage(P, Time, I, MP, B):-
      Time > 0,
      Time <= 1,
      Interest = Time * (P * I/1200),
      B = P + Interest - (Time * MP).
mortgage(P, Time, I, MP, B):-
      Time > 1,
      Interest = P * I/1200,
      mortgage(P + Interest - MP, Time - 1, I, MP, B).
```

The parameters above are principal (`P`), life of the mortgage in months (`Time`), annual interest rate (%) compounded monthly (`I`), the monthly payment (`MP`), and finally, the outstanding balance (`B`). The goal

```
?- mortgage(100000, 360, 12, MP, 0).
```

asks the straightforward query as to how much it would cost to finance a $100,000 mortgage at 12 percent for 30 years, and the answer obtained is `MP` = 1028.61. The

[1] See Section 4.4 for a discussion of experience with large amounts of CLP($\mathcal{R}$) code.

main point of this example, however, is that we can ask, not for the values of, but for the *relationship between* P, MP and B. For example,

```
?- mortgage(P, 120, 12, MP, B).
```

gives the answer

```
B = 0.302995*P - 69.700522*MP.
```

This particular example shows that answer constraints may be viewed as a partial evaluation of the program. In this case, the equation above is the result of partially evaluating the program with respect to Time = 120 and I = 12.

The "test-and-generate" methodology is typically applied to combinatorial and constraint satisfaction problems (CSPs) for which no better approach is known and so a search technique is used. (See [46], for example, for numerous references about successful practical applications.) This involves searching through the problem solution space and making use of the constraints to guide the search either by actively generating values or by pruning when the constraints become unsatisfiable.

In CLP($\mathcal{R}$), the general structure of a test-and-generate program is:

```
p( ... ) :-
      ... primitive constraints ... ,
      ... constraints expressed with predicates ... ,
      ... generators for the variables.
```

where generators are used to instantiate the variables that range over a particular domain. This is generally a better search strategy than "generate-and-test" since the constraints are tested before[2] the generators, thus avoiding generation whenever it is already clear that the constraints are inconsistent. In principle, with a perfect constraint solver for the underlying problem domain, the constraints alone are sufficient to define the answers without resorting to generators. No search would be required, except that which is done internally by the solver. Perfect constraint solvers over discrete domains, however, are typically impractical to use or even impossible to obtain.

Consider the following cryptarithmetic puzzle: find an assignment of the the digits, 0 to 9, to the variables S, E, N, D, M, O, R and Y such that no two variables are assigned the same digit, and the equation

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

can be evaluated in the obvious way. The CLP($\mathcal{R}$) program in Figure 4.2 is a direct representation of the problem. We note that even though the primitive constraints in the program are to be interpreted over the natural numbers, CLP($\mathcal{R}$) can still prune the search space. This is because unsolvability of those constraints in $\mathcal{R}$ implies their unsolvability in the natural numbers, and in this problem it is common for the constraints to be unsolvable in the reals.

[2] Recall that CLP($\mathcal{R}$) uses a left-to-right subgoal selection strategy.

```
solve([S, E, N, D, M, O, R, Y]) :-
      constraints([S, E, N, D, M, O, R, Y]),
      generate([D, R, O, E, N, M, Y, S]).

constraints([S, E, N, D, M, O, R, Y]) :-
      S >= 0, E >= 0, ...  , Y >= 0,
      S <= 9, E <= 9, ...  , Y <= 9,
      S >= 1, M >= 1,
      C1 >= 0, C2 >= 0, C3 >= 0, C4 >= 0,
      C1 <= 1, C2 <= 1, C3 <= 1, C4 <= 1,
      M = C1,
      C2 + S + M = O + C1 * 10,
      C3 + E + O = N + 10 * C2,
      C4 + N + R = E + 10 * C3,
      D + E = Y + 10*C4,
      bit(C1), bit(C2), bit(C3), bit(C4).

bit(0).
bit(1).

generate(L) :-
      gen_diff_digits(L, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).

gen_diff_digits([], _).
gen_diff_digits([H | T], L) :-
      select(H, L, L2),
      gen_diff_digits(T, L2),

select(H, [H | T], T).
select(H, [H2 | T], [H2 | T2]) :-
      select(H, T, T2).
```

Figure 4.2: SEND + MORE = MONEY

4.3 Major Examples: Electrical Engineering

Electrical engineering is an interesting application area for CLP($\mathcal{R}$), since both arithmetic computation (constraint-based) and knowledge representation (rule-based) are appropriate. It is not surprising that this area provided the driving examples for the pioneering work on constraint programming [39, 168, 175]. A preliminary discussion of the work in this section appears in [69].

4.3.1 Analysis and Synthesis of Analog Circuits

Before considering examples of the analysis and synthesis of analog circuits in CLP($\mathcal{R}$), we will briefly sketch a broad methodology for such programming. In general the constraints of such a system can be viewed in terms of a hierarchy. *Leaf constraints* describe the relationship between variables at a sub-system or "local" level, for example Ohm's law for a resistor in a circuit. *Parent constraints* describe the interaction between these sub-systems, for example the use of Kirchoff's current law in node analysis, which states that the sum of currents flowing into a node is zero. This distinction assists us in writing programs whose hierarchical structure reflects that of the problem to be solved. Such programs are easier to understand and reason about. In this programming methodology, leaf constraints are usually encapsulated within a single rule, while parent constraints are programmed as a single rule that combines a number of program modules. Readers needing an introduction to the basic material covered here might wish to consult [68] for circuit analysis and [154] for electronics.

Analysis of DC Circuits

Much of the material necessary for analysis of DC circuits has already been used for simple examples earlier in this chapter, although the approach here will be more systematic. The first component needed is a resistor, and recall that the rule for its behavior is just

```
resistor(V, I, R) :- V = I * R.
```

Some other components will need a piecewise linear model, which can be handled systematically using:

```
linpiece(X,[piece(C1,C2,F1,F2)|Ps]) :-
        C1 < X,
        X <= C2,
        F1 = F2.
linpiece(X,[P|Ps]) :-
        linpiece(X,Ps).
```

Intuitively the program checks whether the value `X` falls within the limits `C1` and `C2` and if so creates an equation `F1 = F2`.

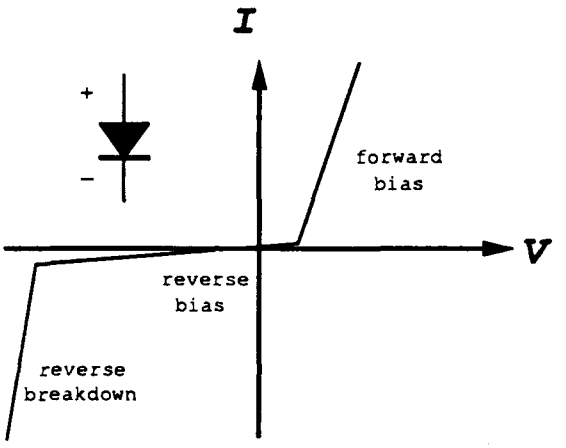


Figure 4.3: Piecewise linear diode model

- Now the resistor model can be re-expressed as:

```
resistor(V, I, R):-
    linpiece(V, [piece(_,_, V, I * R)]).
```

More usefully, diode behavior can be modeled using the piecewise linear approximation of Figure 4.3. The three linear pieces correspond to the three areas of operation of the diode — *reverse breakdown*, *reverse biased* and *forward biased*. Non-determinism and backtracking together take care of the question of which state the diode may be in for a particular circuit.

```
diode(V, I):-
    linpiece(V, [piece(   _, -100, I, (V+100)*10-0.1),
                 piece(-100,  0.6, I, 0.001*V),
                 piece( 0.6,    _, I, (V-0.6)*100+0.0006)]).
```

The following goal determines the behavior of the circuit in Figure 4.4 for circuit values $V = 5$ *volts*, $R_1 = 100\Omega, R_2 = 50\Omega, R_3 = 50\Omega, R_4 = 100\Omega$

```
?- V = 5, R1 = 100, R2 = 50, R3 = 50, R4 = 100,
   resistor(V-A, I1, R1),
   resistor(A, I2, R2),
   resistor(V-B, I3, R3),
   resistor(B, I4, R4),
   diode(B-A, I5),
   I1 + I5 = I2, I3 = I5 + I4.
```

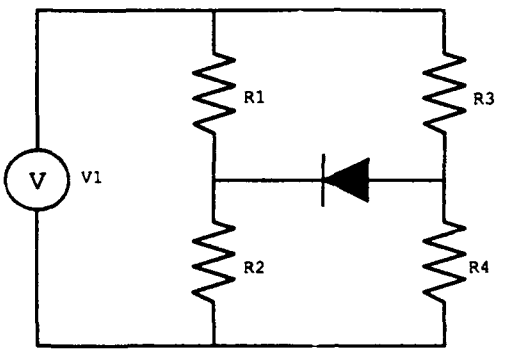


Figure 4.4: Resistive circuit with diode

Let us make some observations on what kinds of constraint solving are necessary to execute this program.

- Since the resistor values are provided at the beginning of the goal, the nonlinear equations established by repeated uses of `resistor` are immediately linear, so no delay is required.
- The diode model essentially offers a choice of three linear relationships. Usually, two of these will be inconsistent with the remaining component and input values, and hence the appropriate part of the diode's operation will be chosen by backtracking. For each possible branch of the model for each diode (in the example there is only one) simultaneous linear equations will have to be solved to determine consistency.

Analysis of Steady-State RLC Circuits

We first consider the application of this approach to the analysis of sinusoidal steady-state RLC (resistor, inductor, capacitor) circuits. We represent (sinusoidal) voltages and currents in the circuit as phasors using complex numbers, where $c(X, Y)$ is used to represent $X + iY$. For example the inductor and capacitor have voltage (V) – current (I) relationships $V = I(i\omega L)$ and $V = I/(i\omega C)$ respectively, where L is inductance, C is capacitance and ω is angular frequency. These devices may be modeled by the following rules:

```
inductor(V, I, L, Omega) :-
     c_mult(I, c(0, Omega*L), V).

capacitor(V, I, C, Omega) :-
     c_mult(V, c(0, Omega*C), I).
```

where `c_mult` is the complex multiplication predicate defined earlier in this chapter,

by:

```
c_mult(c(R1, I1), c(R2, I2), c(R3, I3)) :-
        R3 = R1 * R2 - I1 * I2,
        I3 = R1 * I2 + R2 * I1.
```

Let us consider now in more detail how the `c_mult` predicate might be used and what kinds of constraints it might establish. We use the italic notation for CLP($\mathcal{R}$) variables that have known values.

- In general, two nonlinear equations may have to be delayed until further information is available.
- When the two complex values to be multiplied are available in advance, the two constraints can be solved by local propagation, using simple arithmetic evaluation. That is, when two complex numbers are multiplied, `R1`, `I1`, `R2`, and `I2` are available, so the equations can be directed as the independent propagations:

$$R_3 \leftarrow R_1 R_2 - I_1 I_2$$
$$I_3 \leftarrow R_1 I_2 + R_2 I_1$$

- When two complex values to be divided are available in advance, two simultaneous linear equations must be solved locally. That is, for example, when `R1`, `I1`, `R3`, and `I3` are all available, it is necessary to solve

$$\begin{aligned} R_3 &= R_1 \mathtt{R2} - I_1 \mathtt{I2} \\ I_3 &= R_1 \mathtt{I2} + I_1 \mathtt{R2} \end{aligned}$$

  for `R2` and `I2`.

To connect networks of these components together, we have to satisfy the various conservation laws at the interface of the components. This can be done by ensuring that:

- all component terminals that are connected together are at the same voltage, and
- for each node, the sum of the currents flowing into the node is zero.

Appendix A.1 contains a general package for solving arbitrary sinusoidal steady-state RLC circuits that incorporates these ideas. The package is run using the predicate `circuit_solve` with four arguments. They are respectively: the source frequency, circuit representation, the nodes at ground potential and the nodes whose values are to be printed after the analysis. The package can also be used for D.C. analysis by setting the frequency and imaginary parts of quantities to zero. This is illustrated by the following goal, which analyzes the circuit of Figure 4.5.

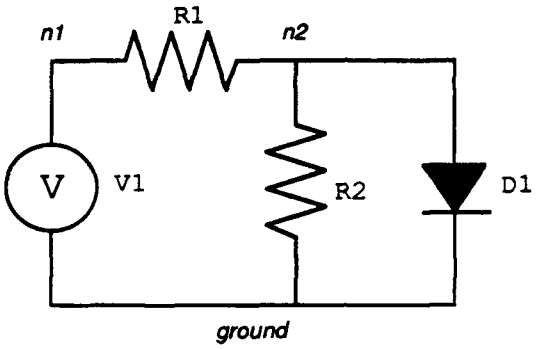


Figure 4.5: Use of the general package to solve a DC circuit

```
?- W = 0,
Vs = 10,
R1 = 100,
R2 = 50,
circuit_solve(W,
                [
                [voltage_source,v1,c(Vs,0),[n1,ground]],
                [resistor, r1, R1, [n1, n2]],
                [resistor, r2, R2, [n2, ground]],
                [diode, d1, in914, [n2, ground]]
                ],
                [ground],
                [n2]
                ).
```

Output is:

```
COMPONENT CONNECTIONS TO NODE n2
resistor r1: 100 Ohms
Node n2 Voltage c(0.60082, 0) Current c(-0.0939918, 0)
resistor r2: 50 Ohms
Node n2 Voltage c(0.60082, 0) Current c(0.0120164, 0)
diode d1: type in914
Node n2 Voltage c(0.60082, 0) Current c(0.0819754, 0)
```

The second example shows a larger RLC circuit (Figure 4.6).

```
?- W = 100, Vs = 10, Tr1 = 5, Tr2 = 0.2,
R1 = 200, R2 = 1000, R3 = 50, R4 = 30,
C1 = 0.05, L1 = 0.005,
circuit_solve(W,
        [
        [voltage_source, v1, c(Vs,0),[in, ground1]],
        [resistor, r1, R1, [in, n1]],
```

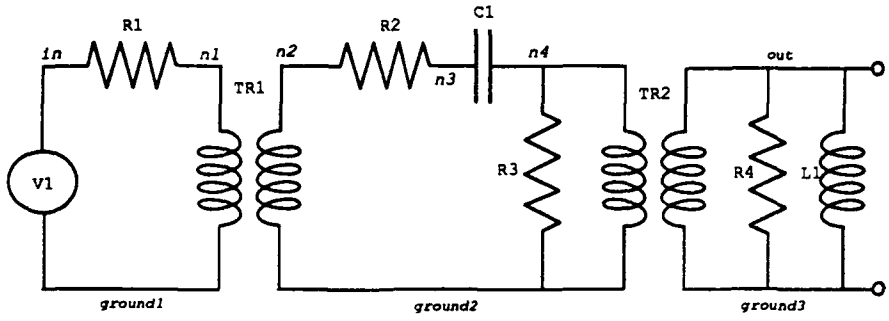


Figure 4.6: RLC Circuit

```
[transformer, t1, Tr1,[n1, ground1, n2, ground2]],
[resistor, r2, R2, [n2, n3]],
[capacitor, c1, C1, [n3, n4]],
[resistor, r3, R3, [n4, ground2]],
[transformer, t2, Tr2,[n4, ground2, out, ground3]],
[resistor, r4, R4, [out, ground3]]
[inductor, l1, L1, [out, ground3]]
],
[ground1, ground2, ground3],
[out]).
```

Output is:

```
COMPONENT CONNECTIONS TO NODE out
transformer t2: ratio of 0.2
Node out V c(3.34983e-06, 0.0001984) C c(-0.0003968 8.78204e-08)
resistor r4: 30 Ohms
Node out V c(3.34983e-06, 0.0001984) C c(1.11661e-07,6.61185e-06)
inductor l1: 0.005 Henry
Node out V c(3.34983e-06, 0.0001984) C c(0.0003967, -6.69967e-06)
```

Finally, let us again consider the critical operations that this program's execution depends on:

- traversal of the data structures representing the circuits, requiring essentially syntactic unification
- solving simultaneous linear equations representing constituent relationships and network relationships
- backtracking to find the right operating mode for components modeled by a piecewise linear relationship

Transistor Circuits: Analysis and Design

If more realistic applications are to be tackled, transistors must be addressed in some way. While the voltage-current behavior of the transistor is highly nonlinear, it can

be modeled for most practical purposes using piecewise linear techniques. We will illustrate this approach through the analysis and design of amplifier circuits using the bipolar junction transistor. This approach is equally applicable to other types of circuit and other types of transistor.

Typical engineering methods for the design of transistor circuits are iterative and guided by heuristics. This involves estimating key circuit parameters, computing the remaining unknown values, and then making corrections to the original estimates. The programs we present use a similar approach. However, we replace iteration by backtracking.

The design of transistor amplifiers falls into three stages.

1. The basic form of the amplifier is selected (for example: common-base, emitter-follower).

2. The biasing circuitry is designed to ensure that the transistor will operate in its active mode within the expected range of inputs to the circuit. Design requirements include insensitivity to transistor parameters such as common-emitter current gain, and temperature.

3. The requirements of the small signal operation of the amplifier are satisfied. These requirements may include high input resistance, low output resistance and predictable gain.

The package presented in Appendix A.2 provides both simple D.C. analysis for biasing and digital circuitry, and small signal analysis for transistor amplifiers. The transistor is modeled by three modes of operation: *active, saturated* and *cutoff*. For amplifier circuits we are primarily interested in the active mode of operation, while for digital circuits the saturated and cutoff modes become important. The rules in Figure 4.7 give the D.C. properties of an NPN transistor (Figure 4.8) in the three modes. The variables *Beta*, *Vbe* and *Vcesat* are device parameters; *Vx* and *Ix* are respectively voltages and currents, where x ranges over b (base), e (emitter), c (collector).

Again this will introduce an element of nondeterminism into the analysis, as the solution of simultaneous equations (and possibly inequalities) will cause backtracking until a consistent configuration is found.

```
transistor_dc(active, npn, Beta, Vbe, Vcesat,
              Vb, Vc, Ve, Ib, Ic, Ie) :-
        Vb = Ve + Vbe,
        Vc >= Vb, Ib >= 0,
        Ic = Beta*Ib,
        Ie = Ic + Ib.
transistor_dc(saturated, npn, Beta, Vbe, Vcesat,
              Vb, Vc, Ve, Ib, Ic, Ie) :-
        Vb = Ve + Vbe,
        Vc = Ve + Vcesat,
        Ib >= 0, Ic >= 0,
        Ie = Ic + Ib.
transistor_dc(cutoff, npn, Beta, Vbe, Vcesat,
              Vb, Vc, Ve, Ib, Ic, Ie) :-
        Vb < Ve + Vbe,
        Ib = 0,
        Ic = 0,
        Ie = 0.
```

Figure 4.7: DC model of NPN transistor

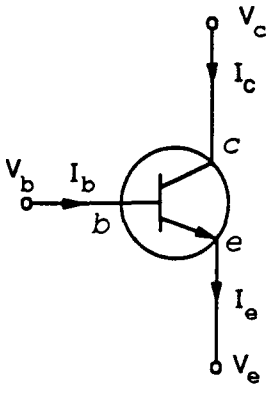


Figure 4.8: An NPN type transistor

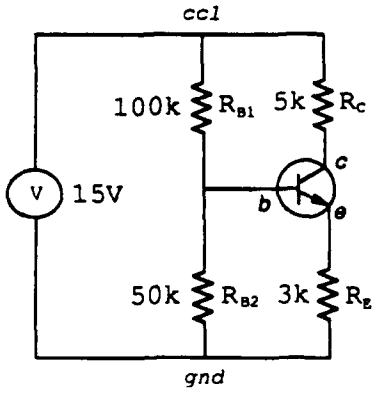


Figure 4.9: Biasing Circuit for NPN transistor

With the addition of the previously defined rule for modeling the (D.C.) properties of resistors, we are able to analyze some simple circuits. For example the simple biasing circuit shown in Figure 4.9 may be analyzed using the following goal:

```
?- resistor(15 - Vb, I1, 100),
   resistor(-Vb, I2, 50),
   I1 + I2 = Ib,
   transistor(State, npn, 100, 0.7, 0.3,
       Vb, Vc, Ve, Ib, Ic, Ie),
   resistor(Ve, Ie, 3),
   resistor(15 - Vc, Ic, 5).
```

Notice how the variable `State` is used here to determine which of the three modes the transistor must operate in.

The full program in Appendix A.2 is an extension of these rules to allow the D.C. analysis of more general transistor circuits, which may include capacitors and diodes. This program is written to reflect the structure of the problem by defining component voltage-current relationships at one level, and component connections, at a higher level, in terms of these voltage/current parameters. The circuit structure is defined by listing the components and their connections. For example, the following goal again analyzes the circuit of Figure 4.8.

```
?- Vcc1 = 15, dc_analysis(Vcc1, _, [
                [resistor, rb1, 100, [cc1, b]],
                [resistor, rb2, 50, [gnd, b]],
                [transistor, t1, [npn,tr1,State], [b,c,e]],
                [resistor, re, 3, [e,gnd]],
                [resistor, rc, 5, [c,cc1]]]).
```

Output is:

```
State = active
```

A number of different types of problems can be solved with this package. For example the following goal does not specify the component values, but instead constrains them to lie within certain ranges.

```
?- Vcc1 = 15, 27 <= R1, R1 <= 100, 5 <= R2, R2 <= 27,
         dc_analysis(Vcc1, _, [
                [resistor, r1, R1, [cc1,b]],
                [resistor, r2, R2, [gnd,b]],
                [transistor, t1, [npn,tr1,State], [b,c,e]],
                [resistor, r3, 3, [e,gnd]],
                [resistor, r4, 5, [c,cc1]]]).
```

Output is:

```
State = active
R1 = 50
R2 = 27
```

In this case, the resistor relationships only become linear when the solving of other equations results in the currents being determined. Then the resistances can be determined uniquely.

Finally we can re-employ the analysis program as a design program. This can be achieved by constraining the design parameters, choosing a circuit template, and co-routining the search for suitable components with a circuit analysis – the *test and generate* technique. After a circuit template has been chosen, the circuit analysis rules are used to set up the appropriate constraints, and then the search for components takes place. When values are chosen for components, an inappropriate choice will often cause the system of constraints to become unsatisfiable immediately, leading to backtracking (assuming that the determination of unsatisfiability does not rely on the consideration of nonlinear constraints). In this way we avoid an exhaustive search to find component values that satisfy the design constraints. A goal for designing a positive gain transistor amplifier is shown below.

```
?- Vcc = 15, Stability < 0.5, Gain > 0.5,
        Inresistance >= 25, Outresistance <= 0.5,
        full_analysis(Vcc, _, Circuit, _, _, Type, Stability,
                Gain, Inresistance, Outresistance).
```

Output is:

```
Circuit= [
          [capacitor, c1, C1, [p1,b]],
          [resistor, r1, 100, [b,cc1]],
          [resistor, r2, 100, [b,gnd]],
```

```
        [transistor, tr, [npn,tr0,active], [b,cc1,e]],
        [resistor, re, 5, [e,gnd]],
        [capacitor, c3, C3, [e,out]]]
Type = emitter_follower
Stability = 0.13
Gain = 0.99
Inresistance = 45.51
Outresistance = 0.47
```

The approach of co-routining constraint satisfaction with the search for components results in a search space pruning of two orders of magnitude and the total time taken is reduced by more than an order of magnitude over the exhaustive search approach. The execution of the analysis program when used in this way is completely different. The circuit template is chosen first, and then all of the various constraints are imposed without any actual component values having been established. This means that a large solved form of linear equations and inequalities, as well as delayed linear inequalities, will be established. Only when all this has been done will the component values be guessed at exhaustively, resulting in the partial simplification of solved constraints and the awakening of delayed constraints, which will frequently have to be undone when an incorrect guess is made. As this goal will certainly exercise our claims about the kinds of constraints arising in constraint solving, and the subsequent decisions about implementation, we will naturally return to it in the empirical study of CLP($\mathcal{R}$) programs.

4.3.2 Digital Signal Flow

We concentrate in this subsection on the simulation of linear shift-invariant digital systems with time as the independent variable [134]. We represent digital systems in terms of linear signal-flow graphs. A signal flow graph is a collection of nodes and directed branchers. Associated with each node is a variable, which represents the signal value at that node. Source nodes have their signal value determined at each stage by some input signal. Branches are either *multiplier* branches or *delay* branches. A multiplier branch multiplies the signal at its input end by its labeling coefficient to give its output; a delay branch has as output its input at the previous time step. The signals at each node in the signal flow graph are such that the signals in any outgoing branches are equal to the sum of the signals in the incoming branches (the *summing constraint*). As an example, consider the digital filter described in Figure 4.10.

We simulate the signal-flow graph by computing the signal values at the nodes at successive time steps, starting from some given initial values. This is achieved by collecting branch equations and summing constraints for each node. This, along with the input signal values to the source nodes, is sufficient to determine the signal values at each node for each time step.

The signal-flow graph simulator appearing in Appendix A.3 is called using the predicate `flow` with a description of the graph as the first argument and the name of the node where the value is to be printed as the second argument. The following goal describes the low pass filter of Figure 4.10.

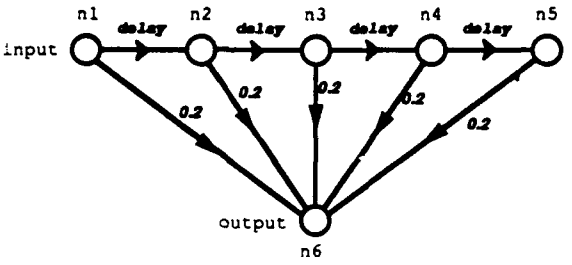


Figure 4.10: High frequency filter

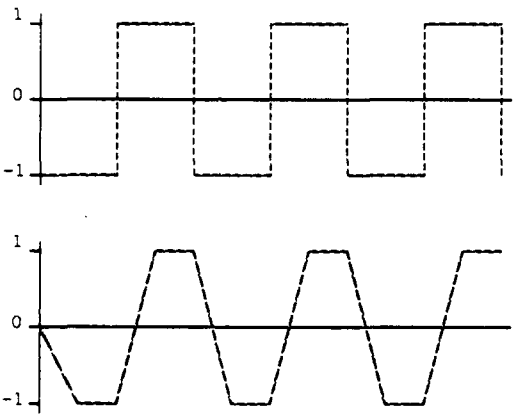
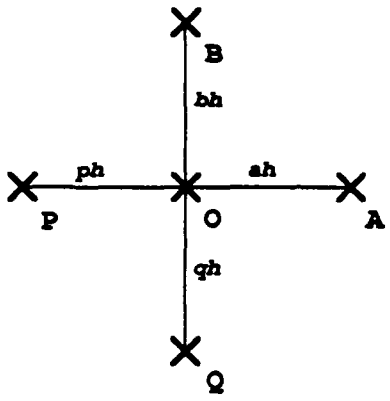


Figure 4.11: Input and output for filter

```
?- flow([
        [source, in, n1],
        [delay, n1, n2], [delay, n2, n3],
        [delay, n3, n4], [delay, n4, n5],
        [coeff(0.2), n1, n6], [coeff(0.2), n2, n6],
        [coeff(0.2), n3, n6], [coeff(0.2), n4, n6],
        [coeff(0.2), n5, n6]
        ],
        n6).
```

Figure 4.11 shows the output from the program in graphical form corresponding to the accompanying square wave input.

Again we should consider what kind of constraint solving is needed in this application. Much of it involves local propagation, as specific inputs are given, and just need to be propagated through the nodes of the signal flow graph. However, consider what happens when the graph involves a loop. We can reason about the way constraints are solved analogously to how the signal flows through the graph. So if the loop contains a delay branch, The node that obtains the feedback value depends only on earlier values,



$$\frac{u_O(ap+bq)}{apbq} = \frac{u_A}{a(a+p)} + \frac{u_B}{b(b+q)} + \frac{u_P}{p(p+a)} + \frac{u_Q}{q(q+b)}$$

Figure 4.12: Liebmann's 5-point approximation to Laplace's equation

so local propagation is still sufficient. However, if there is no delay element in the loop, that node's value depends partly on itself. Hence simultaneous equations must be solved.

4.3.3 Electro-Magnetic Field Analysis

Often it is undesirable or impractical to analyze electrical systems in terms of lumped circuit components. In many cases an analysis using electro-magnetic field theory is required, typically involving the solution of partial differential equations subject to some boundary or initial conditions. A simple way to solve these problems is to use a finite difference approximation. We will consider the five-point approximation to Laplace's equation known as Liebmann's method described in Figure 4.12, which can be used to solve Dirichlet, Neumann and mixed boundary value problems. The finite difference equations are the leaf constraints — they apply to a neighborhood of points. The complete finite difference problem is a parent constraint arising from the overlapping of these neighborhoods.

Figure 4.13 shows a simple program to solve the Dirichlet problem for Laplace's equation in two dimensions, with a sample query and output. The region of interest is represented as a matrix (list of lists) of anonymous variables, with constants at the edge specifying the boundary values. Constraints (finite difference equations) are collected at each point of the matrix by "sliding" a "window" of 9 points across the matrix and

Program:

```
laplace([_,_]).
laplace([H1,H2,H3|T]):-
        laplace_vec(H1,H2,H3),
        laplace([H2,H3|T]).

laplace_vec([_,_],[_,_],[_,_]).
laplace_vec([TL,T,TR|T1],[ML,M,MR|T2],[BL,B,BR|T3]):-
        B + T + ML + MR - 4*M = 0,
        laplace_vec([T,TR|T1],[M,MR|T2],[B,BR|T3]).
```

Sample goal:

```
?- laplace([[0,   0,   0,   0,   0,   0,   0],
            [100, _,   _,   _,   _,   _, 100],
            [100, _,   _,   _,   _,   _, 100],
            [100, _,   _,   _,   _,   _, 100],
            [100, _,   _,   _,   _,   _, 100],
            [100, _,   _,   _,   _,   _, 100],
            [100, 100, 100, 100, 100, 100, 100]]).
```

Output:

```
[[0,   0,    0,    0,    0,    0,    0],
[100, 53.1, 37.1, 33.1, 37.1, 53.1, 100],
[100, 75.4, 62.1, 58.1, 62.1, 75.4, 100],
[100, 86.5, 77.9, 75.0, 77.9, 86.5, 100],
[100, 92.8, 87.9, 86.2, 87.9, 92.8, 100],
[100, 96.9, 94.6, 93.9, 94.6, 96.9, 100],
[100, 100,  100,  100,  100,  100,  100]]
```

Figure 4.13: Solving Laplace's equation

referring only to the variables in that window. Simultaneous linear equations must be solved at run time.

4.4 Survey of Other Applications

Since the first prototype implementation of CLP($\mathcal{R}$) became generally available in 1987, a number of other applications have been explored in considerable detail by various researchers. In fact, some of these projects were inspired by early versions of the work described above. Some of the more significant projects are briefly outlined here. Most use versions of the system based on material in Part III of this thesis.

- *Genetic Restriction Site Mapping*
  The restriction site mapping problem is to reconstruct the placement of certain DNA subsequences on a molecule corresponding to a given enzyme given data on the digest fragments produced when the molecule is treated with that enzyme. In [192], Yap describes a methodology for solving this problem using CLP($\mathcal{R}$) programs. The problem is of tremendous practical importance, and is computationally intractable in general. The programs involve reasoning with linear equations and inequalities on real-valued variables.

- *Civil Engineering*
  In [102], Lakmazaheri and Rasdorf described the use of CLP($\mathcal{R}$) for the analysis and partial synthesis of truss structures. Their program consists of only sixteen rules, and deals with general two-dimensional truss structures with pin and roller supports, concentrated loads, and truss elements with uniform cross section. The program uses constraints to describe the physical behavior of structural components. Considerable use is made of symbolic answer constraints. The constraints tend to be linear and nonlinear equations. In addition to checking the validity of truss and support components, the program can be used to generate their spatial configurations.

- *Options Trading Analysis*
  OTAS [104, 78, 79] is a system for designing stock option investment portfolios. Linear equations and inequalities, as well as nonlinears were used to model expected payoffs. Unsatisfiable linear inequalities were also used. Additionally, considerable use was made of CLP($\mathcal{R}$)'s output constraints.

  In [74], Homiak described the use of CLP($\mathcal{R}$) for solving partial differential equations arising in the domain of options valuation. Problems are described using a symbolic input language from which CLP($\mathcal{R}$) constraints are generated. The examples in the document include cases where the grid has up to 6000 elements. The PDEs are solved using various finite difference techniques, and thus this (large) piece of software is a natural generalization of the tiny program shown above for solving Laplace's equation.

- *Configuration Design for Mechanical Systems*
  In [173], Sthanusubramonian discusses the use of CLP($\mathcal{R}$) to design gear boxes, given physical constraints on the size and layout of a gearbox, as well as functional constraints on the outputs. The arithmetic constraints are mostly nonlinear, and hence CLP($\mathcal{R}$) is mostly used to collect and maintain systems of nonlinear constraints.

- *Generation of Test Data for Protocol Verification*
  In Mockingbird [57], CLP($\mathcal{R}$) is used to implement grammars that model communications protocols. By running the programs "in reverse", these grammars are used to generate sample messages, and additional constraints are used to tune the type of messages produced. The constraints are typically linear equalities and inequalities, and the values considered are usually integers.

- *Temporal Reasoning*
  In [135], Ostroff describes the use of CLP($\mathcal{R}$) as a computational logic for modeling, simulating and verifying real-time discrete event processes. Constraint solving is used to avoid the combinatorial explosion that results from the state enumeration normally used for such verification. The constraints are mostly simple linear equations and inequalities, and the intended domain of the modeling is that of integers.

  In [5, 6], Amon and Borriello describe the use of CLP($\mathcal{R}$) for verifying that behavioral constraints on classes of asynchronous circuits are satisfied. The programs make use of linear equations and inequalities, and use answer constraints to relate timing delays, making tradeoffs apparent.

- *Interactive Tutoring for Music Theory*
  In [182], Tobias discusses the Harmony Intelligent Tutoring System, with particular emphasis on the Harmony Expert module, which is implemented in CLP($\mathcal{R}$). The Harmony Expert involves a complex set of inference rules incorporating linear equations and inequalities on variables ranging over a finite, convex set of integers. The methodology makes use of non-trivial output constraints as well.

- *Modeling Concurrent Execution of Programs*
  In [116], Mercouroff and Weitzman discuss the use of Prolog III and CLP($\mathcal{R}$) to model the execution times of a class of recursive parallel programs. The CLP programs involve linear equations and inequalities, and provide analytical solutions to the problem of determining the execution time of the corresponding parallel program. Rational numbers are the ideal domain.

4.5 Empirics

Now that we have studied programming methodology and typical programs from a static viewpoint, we need to understand how programs behave at run time. Thus we provide measurements for the runtime behavior of a number of CLP($\mathcal{R}$) programs.

Some of the results were foreshadowed by the study of Yap [191], using an early CLP($\mathcal{R}$) interpreter. The measurements were obtained by instrumenting Version 1.1 of the IBM CLP($\mathcal{R}$) compiler, many aspects of which are described in some detail later in this thesis. This analysis is somewhat circular, since the results are being used to justify decisions made about the language design and implementation, and are obtained from a system incorporating those decisions, using programs written by programmers often aware of those decisions, for that implementation. While this means that the figures should be taken with a grain of salt, they are still clearly of great value, since they demonstrate certain characteristics rather decisively.

The statistics are presented on a per-goal-per-program basis for a representative suite of programs. Some of the parameters will not make very much sense at this point in the exposition because they rest heavily on subsequent implementation chapters. Below we attempt to give some intuition as to what they correspond to in terms of the programs. They are in four broad categories:

- *Resolutions*
  This indicates the overall amount of computation and search.
  - *Call* The number of subgoals reduced. Shallow backtracking does not add to this total, but deep backtracking does.
  - *Fail* Number of failures of any kind, thus a measure of amount of backtracking.
- *Disagreement Pairs*
  These are essentially equations arising from attempting to resolve a subgoal.
  - *Top* Number of top level equations. Each attempt to reduce a subgoal with an n-ary predicate results in n such equations. Thus backtracking adds to this total.
  - *Rec* This is the total number of equations. One equation of the above kind can result in several equation, say when subterms of uninterpreted functors are equated. We call these equations disagreement pairs.
  - *%PP* The percentage of disagreement pairs that equate two solver variables.
  - *%VP* The percentage of disagreement pairs that equate a solver variable and a free variable.
  - *%VV* The percentage of disagreement pairs that equate two free variables.

- *Arithmetic Equations*
  Summarizes the amount of arithmetic dynamically occurring.
  - *EQ* Number of explicit linear arithmetic equations. This is loosely related to the number of such equations occurring in the program, since the compiler performs transformations on constraints.
  - *Num* Number of real (floating point) numbers explicitly assigned to solver variables.
  - *Calls* Number of equations actually passed to the linear equation solver.
  - *New* Number of the above equations that did not really require solving, since they included a new variable.
  - *Ground* Number of the above that immediately give a ground value to the new variable.
  - *Par* Number of equations passed to the solver that resulted in a simplifying substitution. Essentially these are the rest of the equations passed to the solver.

- *Inequalities*
  This is to capture the number of explicit linear inequalities arising.
  - *GT* Number of strict linear inequalities encountered.
  - *GE* Number of non-strict linear inequalities encountered.
  - *ICalls* Number of inequalities and equations passed to the inequality solver. There is no simple relationship between this parameter and the others above. The inequalities passed result either from the *GT* or *GE* parameters, but the equalities can result either from *EQ* instances of the *Par* kind, or from *Num* instances or *PP* type disagreement pairs. This will be explained in more detail when discussing the implementation of CLP($\mathcal{R}$).

The results are summarized in four tables. The first two, in Figures 4.14 and 4.15, are for a set of simple example programs. Some of these, like Mortgage or SMM actually perform a lot of computation, but they are typically small programs written to illustrate various aspects of the use or performance of CLP($\mathcal{R}$). The third and fourth tables, in Figures 4.16 and 4.17, contain results from more substantial programs. Most of these were also written to illustrate the use of CLP($\mathcal{R}$), but are often quite large and complicated, are more concerned with the actual applications, and in some cases were written by programmers who were not involved with any implementation of CLP($\mathcal{R}$). Many of the programs have already been discussed. We give some relevant details about the programs and queries.

- *Fib*
  The Fibonacci program for early in this chapter. The first goal uses it to compute F_{14}, and the second goal computes the inverse function.

- *Mortgage*
  This is a form of the mortgage program (similar to that shown in this chapter)

given in Chapter 10. The four queries given there are used, as they represent four different uses of the program.

- *Option*
  This is a program for reasoning about algebraic combinations of stock option transactions, described in [104]. The two goals obtain all solutions to queries about combinations of put and call options.

- *Ladder*
  A simple program for reasoning about resistive circuits – specifically various voltage dividers, ladder networks and bridges. Used for teaching CLP($\mathcal{R}$) programming at the University of Washington, and written by Alan Borning. The various goals analyze various circuits of this kind.

- *SMM*
  This is the SEND+MORE=MONEY puzzle of Figure 4.2.

- *Laplace*
  This is the program shown in this chapter for solving Laplace's equation, with a larger goal grid.

- *Critical*
  This is a program written by Roland Yap for critical path analysis. Constraints are used to avoid the multiple passes through the graph. The goals analyze various sizes of graphs.

- *Discrete*
  This program, written by Jonathan Ostroff and Jeff Klein, is described in [135]. It reasons about discrete event processes, and the query used to obtain these results is the `filterpath(trainGate)` query described in the paper.

- *Verify*
  This program described above, was written by Amon and Borriello [5, 6] for verifying asynchronous circuits. Some of the goals run for a very long time. These are examples from their various papers.

- *DNA*
  This is the program for genetic restriction site mapping written by Yap [192], using various examples similar to those in that paper.

- *EE*
  This is a program similar to that described earlier in this chapter and in [69] for analysis and synthesis of electrical circuits with diodes and transistors. The first goal determines the activation state of a transistor in a circuit, the second analyzes a NAND gate, the third designs an amplifier using the test-and-generate methodology, and the fourth obtains the same design with much more work, using exhaustive search.

- *Gears*
  This is a gear-train design program, written by Devika Subramanian and Cheuk-San Wang at Cornell University. The query is the biggest synthesis query made available, searching only for the first solution.

While we will draw heavily on these results in subsequent chapters, it is appropriate to make a few general observations at this stage.

1. The large numbers of VV and VP cases in unification in many programs show the importance of the extended unification algorithm in Chapter 7.

2. The "New" category of arithmetic equations shows that many linear equalities encountered are *a priori* clearly consistent with the set of collected constraints.

3. The "Ground" category of arithmetic equations demonstrates that a substantial fraction represent forward propagation steps.

4. The analysis of inequalities shows that in many programs a substantial fraction of inequalities are simple tests. The ratio of the number of arithmetic constraints to the number of disagreement pairs varies substantially between programs, but generally is very small.

5. All of the programs make use of non-trivial arithmetic constraint solving, but most of them do so sparingly.

Program	Query	Resolutions		Disagreement Pairs				
		Call	Fail	Top	Rec	%PP	%VP	%VV
Fib	Fw	611	0	2438	2438	75	25	0
	Bw	1505	999	8155	8155	70	30	0
Mortgage	1	361	1	1805	1805	0	80	20
	2	361	1	1805	1805	0	100	0
	3	361	1	1805	1805	0	100	0
	4	358	3	1800	1800	0	100	0
Option	1	33	15	206	206	0	63	27
	2	105	49	690	690	0	62	30
Ladder	1	15	4	47	117	11	26	3
	2	14	3	42	99	11	25	3
	3	20	7	73	188	11	26	3
	4	19	6	68	170	11	26	3
	5	21	9	86	227	11	27	3
	6	20	8	81	209	11	27	3
	7	213	101	930	2254	12	27	4
SMM		1262	1247	6580	11202	10	33	0
Laplace		113	90	514	2582	0	29	6
Critical	1	124	80	595	1433	0	15	16
	2	511	446	2882	7219	0	13	16
	3	382	339	2139	5310	0	14	15
Discrete	1	1079	547	1891	3439	0	28	0

Figure 4.14: Small Programs: Resolution and Unification

Program	Query	Arithmetic Equations						Inequalities		
		EQ	Num	Calls	New	Ground	Par	GT	GE	ICalls
Fib	Fw	1827	1221	4627	3049	1218	1218	609	0	0
	Bw	5559	3206	12848	8766	3071	4028	2475	0	39
Mortgage	1	719	5	1083	1083	719	0	360	359	0
	2	719	5	1083	1082	359	1	360	359	1
	3	719	3	1081	1080	359	1	361	360	362
	4	716	6	1081	1075	356	6	357	359	365
Option	1	15	28	66	62	6	4	14	32	37
	2	50	105	235	235	25	0	48	108	126
Ladder	1	12	10	35	23	0	12	0	0	0
	2	11	8	30	20	0	10	0	0	0
	3	19	15	56	36	0	20	0	0	0
	4	18	13	51	33	0	18	0	0	0
	5	24	14	65	41	0	24	0	0	0
	6	23	12	60	38	0	22	0	0	0
	7	267	111	686	438	19	248	19	0	0
SMM		4	63	276	64	0	212	0	26	236
Laplace		81	40	122	113	0	9	0	0	0
Critical	1	28	7	40	37	24	3	2	0	0
	2	105	21	151	133	81	18	10	0	0
	3	77	16	110	101	66	9	7	0	0
Discrete	1	22	362	194	171	22	23	0	30	31

Figure 4.15: Small Programs: Arithmetic

Program	Query	Resolutions		Disagreement Pairs				
		Call	Fail	Top	Rec	%PP	%VP	%VV
Verify	1	26941	26404	108466	226915	0	5	1
	2	132189	134280	543085	1078951	0	2	0
	3	6003326	6116029	24589775	49470767	0	1	0
	4	5036771	5148358	20702284	41478978	0	1	0
	5	6003267	6116029	24589775	49470767	0	1	0
DNA	1	99	68	549	1181	5	37	4
	2	10445	8939	58565	136715	6	41	2
EE	1	109	106	990	1733	1	8	23
	2	137	131	1248	2255	0	10	21
	3	1623	1978	17941	27605	1	9	23
	4	101583	126118	1062863	1745129	0	7	25
Gears		39161	26261	157762	659201	6	25	6

Figure 4.16: Large Programs: Resolution and Unification

Program	Query	Arithmetic Equations						Inequalities		
		EQ	Num	Calls	New	Ground	Par	GT	GE	ICalls
Verify	1	51	523	586	574	38	12	714	86	48
	2	175	1030	1217	1205	157	12	1417	482	49
	3	1218	8292	9511	9484	1102	27	9986	2276	279
	4	1254	6929	8184	8167	1168	17	8812	2874	257
	5	1218	8290	9509	9482	1102	27	9986	2276	281
DNA	1	64	27	147	105	43	42	35	26	87
	2	6969	1030	11350	8029	5311	3321	5383	1035	8896
EE	1	24	50	83	76	3	7	0	2	4
	2	37	59	103	93	7	10	2	4	7
	3	389	974	2586	1400	42	1186	2	20	364
	4	24215	18971	53085	45353	2436	7732	2	1472	3617
Gears		88650	53558	242001	230115	77805	11846	3419	0	0

Figure 4.17: Large Programs: Arithmetic

Chapter 5

λProlog and Elf

The idea of a language based on solving equations between simply-typed λ-expressions emerged from the desire to use λ-abstraction as a means of representing data rather than defining functions. The first language based on the idea was λProlog [121]. In this chapter we will discuss the closely related language Elf [139], although much of the material is equally applicable to λProlog. The chapter will begin with an introduction to the languages, and will continue with a discussion of programming methodology and empirical observations.

5.1 The Domain

The core domain of Elf is the set of $\beta\eta\alpha$-equivalence classes of λ-expressions with dependent types, where application is the only operation and equality is the only relation. The terms are thus typed lambda expressions with constructors (functors and constants) and existentially quantified variables. A term free of existential variables can be considered ground. As we shall see later, full Elf actually involves constraints on expressions from the λ-calculus with dependent types. For an introduction to the simply-typed λ-calculus, the reader should consult [143]. Here we will give only a brief overview by way of example. The equalities

$$\lambda x : \tau.\ x = \lambda y : \tau.\ y$$

$$\lambda x : \tau.\ fa = \lambda x : \tau.\ f((\lambda y : \tau.\ y)a)$$

hold, while

$$\lambda x : \tau.\ \lambda y : \tau.\ x = \lambda y : \tau.\ \lambda x : \tau.\ x$$

does not. Now consider equalities involving terms with existentially quantified variables.

$$\exists M : \tau.\ \lambda x : \tau.\ M\ x = \lambda x : \tau.\ x$$

is satisfiable, with $M = \lambda x : \tau.\ x$ being one solution. This is maximally general, and also unique, modulo $\beta\eta\alpha$-equivalence. On the other hand, some constraints can have

more than one maximally general solution. For example,

$$\exists M : \tau \to \tau.\ faa = Ma$$

has the solutions

$$\begin{aligned} M \;=\; & \lambda x : \tau.\ fxx \\ & \lambda x : \tau.\ fxa \\ & \lambda x : \tau.\ fax \\ & \lambda x : \tau.\ faa \end{aligned}$$

of which no two are $\beta\eta\alpha$-convertible. In fact, sometimes there are infinitely many maximally general solutions. Consider

$$\exists M : \tau \to \tau.\ M(fa) = f(Ma),$$

which has the solutions

$$M = \lambda x : \tau.\ f^n x,\quad n \in \{0, 1, 2, \cdots\}.$$

One major design issue related to these languages comes from the fact that constraint satisfiability is undecidable. It is semi-decidable, but the pre-unification algorithm of Huet [76], and Elliot's extension to dependent types [50], as demonstrated above, involve backtracking through solutions to individual equations. Details of these issues will be discussed in Chapter 11.

There are three prevailing approaches to dealing with this problem:

- *Tolerate the inefficiency*
  It may be that some kinds of programs actually can benefit from the full generality, although this has not been argued convincingly.

- *Restrict the language to a fragment with decidable constraints*
  In [119], Miller presents a language L_λ, described below, which is syntactically restricted such that the hard cases of unification do not occur. It is claimed that this subset is satisfactory for most examples. However, programming becomes more difficult for certain examples, as idioms that can be expressed directly in the full language here have to be coded up explicitly. This can also result in a loss of efficiency, as will be demonstrated in Section 5.6.

- *Introduce delay for the constraints that cause inefficiency*
  Many Elf programs actually conform to the restrictions of L_λ, and it turns out that many programs that do not conform to the syntax of L_λ still do not involve the difficult case of unification *dynamically*. Furthermore, it has been argued by Pfenning [139] that many of the remaining programs could avoid the difficult case if certain constraints were deferred until they were sufficiently instantiated to fall into an easier case. This is directly analogous to the case for delay of nonlinears in CLP($\mathcal{R}$), and will be examined more closely here and also in Chapter 11.

It should be noted that, while Elf is certainly a CLP language in essence, it departs from the pure CLP scheme in a number of unusual ways.

- it is strongly typed;
- embedded implication, first described by Miller in [118], is used to modify the program with temporary assumptions, as a central programming technique;
- universal quantification is provided, and is central to programming technique, as it essentially provides a new constant different from all other constants occurring so far;
- programs can manipulate proof objects, in a way that is kept under control by the type system.

The language λProlog is closely related to Elf, many of the same design, programming and implementation observations apply, and it is more widely used than Elf. Hence, it is useful to consider how the two languages differ. λProlog has no ability to reason about proofs of goals. However, it allows polymorphic constructors and predicates to be defined, and it also allows predicate variables. That is, predicate symbols are in many respects treated as constructors.

Finally, L_λ [119] is a syntactically restricted version of L_λ, with a corresponding restriction of LF [140] for Elf, where variables that are universally quantified at the top level of a clause can only be applied to a set of distinct bound variables. As mentioned before, and discussed fully in Chapter 11, this is intended to avoid the hard case of solving constraints.

5.2 The Languages

To introduce Elf, let us consider in Figure 5.1 an example that is essentially a Prolog program. It converts propositional formulae to negation normal form. This is very straightforward, and can be directly translated to Prolog.

First, `bool` is declared as the type of boolean truth values, and `var` as the type of propositional variables. Then the constructors `not`, `and` and `or` are the usual logical connectives. Three sample propositional variables `v1` to `v3` are defined, and the constructor `prop` turns a propositional variable into a boolean proposition. The program consists of one relation `negn`, which is first declared as a relation between two booleans. The first rule defining `negn`, named `t_nn`, eliminates double negation. The next two rules `t_na` and `t_no` flip conjunction and disjunction while carrying over a negation. The two base cases (for termination) are `t_v` and `t_nv`. Finally, `t_o` and `t_a` pass trough conjunction and disjunction without changing them.

```
bool    : type.
var     : type.

not     : b -> b.
and     : b -> b -> b.
or      : b -> b -> b.

v1      : var.
v2      : var.
v3      : var.
prop    : var -> bool.

negn    : bool -> bool -> type.

t_nn    : negn (not (not X)) Y
                <- negn X Y.
t_na    : negn (not (and X Y)) (or X1 Y1)
                <- negn (not X) X1
                <- negn (not Y) Y1.
t_no    : negn (not (or X Y)) (and X1 Y1)
                <- negn (not X) X1
                <- negn (not Y) Y1.
t_v     : negn (prop X) (prop X).
t_nv    : negn (not (prop X)) (not (prop X)).
t_a     : negn (and X Y) (and X1 Y1)
                <- negn X X1
                <- negn Y Y1.
t_o     : negn (or X Y) (or X1 Y1)
                <- negn X X1
                <- negn Y Y1.
```

Figure 5.1: Elf program to convert propositional formula to negation normal form

As an example query, consider

```
?- negn (not (and (or (not (prop v1)) (prop v2))
                  (or (prop v3) (not (prop v1))))) X.
```

which obtains the answer

```
X = or (and (prop v1) (not (prop v2)))
       (and (not (prop v3)) (prop v1))
```

with proof object

```
t_na (t_no (t_nn t_v) t_nv) (t_no t_nv (t_nn t_v)).
```

Proof objects will be discussed in more detail in Section 5.4. Essentially, a proof object is a term constructed from constants declared in an Elf signature, or equivalently the names of Elf rules. It represents a record of how the rules of a program were used to solve a particular query.

Now we can discuss Elf somewhat more systematically. The basic building block of an Elf program is a signature, that is, a sequence of constant declarations. Some of these declarations have the character of declarations for data constructors (functors) or predicates, while others have the character of declarations of inference rules or clauses. In principle, each declaration could be given all of those interpretations, but in practice few are meaningful.

This unusual way to view a program stems from the origin of Elf as a computational view of the Edinburgh Logical Framework (LF) [65], which is type-theoretic. This relationship is justified by the Curry-Howard isomorphism, and Elf is described in that way in [138].

For a tutorial introduction to Elf, see [117]. Here we give only a brief overview. The first kind of declaration is that of a constant *type* or *type family*. Types and type families are classified by *kinds*, the simplest of which is `type`. Data constructors are declared using *decl*, *type* and *atom*. Data is represented using *objects*. The dependent function type, { *var* : *type* } *type*, is elsewhere often written as $\Pi var : type.\ type$. Juxtaposition is left-associative. The square brackets denote λ-abstraction, and [*var* : *type*] *obj* is more traditionally written as $\lambda var : type.\ obj$. We adopt the shorthand [*var*] and { *var* } for those cases of abstraction or quantification where we want Elf type reconstruction to fill in the omitted types. The scope of the abstraction [...] and quantification {...} extends to the end of the declaration or enclosing parentheses. We summarize the syntax of Elf in Figure 5.2.

In Elf, predicates manifest themselves as type families and thus a type family declaration can be interpreted as a predicate declaration. When defining rules we often use the more conspicuous left-pointing arrow. One should bear in mind that this has no semantic significance, and `A <- B` and `B -> A` are parsed to the same internal representation. The backwards arrow is left associative and a Prolog rule `p :- q, r.` would be represented via the type `p <- q <- r`.

The variable name convention of Elf is much like that of Prolog. Any token beginning with an upper case letter is automatically a variable, and if an explicit quantifier is

```
famdecl ::=   famconst : kind.

kind ::=      type
            | type -> kind
            | { var : type } kind

decl ::=      const : type.

type ::=      atom
            | type -> type
            | { var : type } type

atom ::=      famconst obj ... obj

obj ::=       const
            | var
            | [ var : type ] obj
            | obj obj
```

Figure 5.2: Summary of Elf syntax

not provided, one will be added by type reconstruction. Additionally, variables that are explicitly quantified need not start with an upper case letter. We follow the convention that bound variables that will become logic variables (and thus subject to instantiation) during execution of a query are written in uppercase and bound variables that become parameters (and thus act like constants to unification) are written in lowercase.

In Elf, unlike Prolog, an underscore character does not denote an anonymous universally quantified variable, although it can sometimes be used for this purpose. Instead, it describes an existentially quantified anonymous variable, and type reconstruction is at liberty to replace an underscore with any term, depending on the available type information. Thus `{x}B` and `{x:_}B` are fully equivalent. In Prolog, since no variables are ever instantiated during parsing, this distinction does not arise.

Queries to the Elf interpreter consist of a type, possibly with free variables, which act as logic variables. The goal is to find a term of the given type. In the most common interpretation, this term represents a deduction of the judgment that is represented by the type. During the solution of a goal, the free variables in the query are instantiated as necessary, as in Prolog. As will be described in more detail later, the names of rules are used as constructors in assembling deductions.

The operational model of Elf is superficially much like that of Prolog and λProlog. A computation rule similar to Prolog's left-right atom selection rule is employed. When using a rule, the subgoals will be solved "inside-out". For example, resolving the goal `?- p.` with the rule `c : p <- q <- r` will first solve subgoal `q` and then `r`. Unification is modulo $\beta\eta\alpha$-convertibility, where certain equations are postponed as constraints. A goal of the form `A -> B` is solved by adding `A` to the set of rules available for goal reduction. The third form (universal quantification) is solved by replacing the universally quantified variable with a new constant.

One complication is that rules and types in Elf are essentially indistinguishable. In fact, the only desired distinction is that of operational behavior. For this reason, the Elf module system [66] allows type families to be defined as either *open* or *closed*. Rules defining a closed type family are used to solve goals by search. This distinction is described in more detail in [139][1].

5.3 Higher-Order Abstract Syntax

Most of the important application areas for Elf are natural extensions of applications of λProlog. The major application of λProlog is meta-programming, where the object language is typically either a programming language or a logic. This is because λProlog's simply-typed λ-expressions enable the use of a meta-programming technique known as higher-order abstract syntax. This approach was inspired by Church [23], Martin-Löf [131] and Huet & Lang [77]. A λ-calculus-based meta-language is used to represent expressions of the object language. This enables variable binding and scoping in the object language to be represented with the help of λ-abstraction in the meta-

[1] In more recent work towards a module system for Elf [66] open declarations are called *static* and closed declarations are called *dynamic* according to their role in proof search.

language. This, in turn, enables substitution and variable occurrence restrictions to be implemented using β-reduction in the meta-language, which avoids, for example, the requirement to perform explicit α-conversion to prevent capture of bound variables.

To illustrate these ideas, let us extend the above program to the first-order predicate calculus. The additional code below introduces the type `d` for the domain of quantified variables, and a sample predicate symbol `p`. The interesting constructors are **`exists`** and **`forall`**, which use abstraction to represent quantifiers. That is, for some formula Φ, $\forall x \Phi(x)$ is represented by $\forall(\lambda x.\phi(x))$, etc. The rules `t_E` and `t_A` simply pass through the quantifiers. Note how applying an abstraction to a universally quantified variable is used to descend through the body of the abstraction, and to reconstruct an abstraction from the result. The same technique is used in `t_nE` and `t_nA`, except that the quantifiers are flipped because of the descending negation.

```
dom     : type.
p       : dom -> var.

exists  : (dom -> bool) -> bool.
forall  : (dom -> bool) -> bool.

t_nE    : negn (not (exists X)) (forall X1)
                <- {y : dom} negn (not (X y)) (X1 y).
t_nA    : negn (not (all X)) (exists X1)
                <- {y : dom} negn (not (X y)) (X1 y).
t_E     : negn (exists X) (exists X1)
                <- {y : dom} negn (X y) (X1 y).
t_A     : negn (forall X) (forall X1)
                <- {y : dom} negn (X y) (X1 y).
```

As an example query, consider

```
?- negn (not (exists ([x] (not (prop (p x)))))) X.
```

has the execution trace

```
[Solving goal negn (not (exists ([x:dom] not (prop (p x))))) ?X207
Resolved with clause t_nE
Solving goal {y:dom} negn (not (not (prop (p y)))) (?X1230 y)
Introducing new parameter !y232
[Solving goal negn (not (not (prop (p !y232)))) (?X1230 !y232)
Resolved with clause t_nn
[Solving goal negn (prop (p !y232)) (?X1230 !y232)
Resolved with clause t_v
```

which obtains the answer

```
X = all ([y:dom] prop (p y))
```

with proof object

```
t_nE ([y:dom] t_nn t_v).
```

Using λProlog for higher order abstract syntax was described by Pfenning and Elliott in [141]. With this technique, λProlog has been used as an implementation language for type inference [137], meta-interpretation [61] and theorem proving [52]. These techniques are readily applicable when programming in Elf. We illustrate the techniques here with parts of the Elf type inference and evaluation code for Mini-ML, as described in [117]. Mini-ML is a small functional language based on a simply typed λ-calculus with polymorphism, products, conditionals, and recursive function definitions. We use the example of application, λ abstraction and polymorphic let. The relevant concrete syntax is as follows.

e ::=

⋮

| lambda x . e

| e_1 e_2

| let x = e_1 in e_2

⋮

τ ::=

⋮

| τ -> τ

⋮

The three relevant types rules are

of_lam
$$\frac{\Pi, x : \tau_1 \vdash e \in \tau_2}{\Pi \vdash \texttt{lambda}\ x\ .\ e \in \tau_1 \texttt{->} \tau_2}$$

of_app
$$\frac{\Pi \vdash e_1 \in \tau_2 \texttt{->} \tau_1 \qquad \Pi \vdash e_2 \in \tau_2}{\Pi \vdash e_1\, e_2 \in \tau_1}$$

of_let
$$\frac{\Pi \vdash e_1 \in \tau_1 \qquad \Pi \vdash [e_1 / x]\, e_2 \in \tau_2}{\Pi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \in \tau_2}$$

and the relevant evaluation rules are

eval_lam
$$\frac{}{\vdash \texttt{lambda}\ x\ .\ e \Rightarrow \texttt{lambda}\ x\ .\ e}$$

eval_app_lam
$$\frac{\vdash e_1 \Rightarrow (\texttt{lambda}\ x\ .\ e_1') \qquad \vdash e_2 \Rightarrow \alpha_2 \qquad \vdash [\alpha_2 / x]\, e_1' \Rightarrow \alpha}{\vdash e_1\, e_2 \Rightarrow \alpha}$$

eval_let
$$\frac{\vdash e_1 \Rightarrow \alpha_1 \quad \vdash [\alpha_1 / x] e_2 \Rightarrow \alpha_2}{\vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Rightarrow \alpha_2}$$

We define a representation function $()^+$ from Mini-ML expressions and types to Elf expressions. This is not implemented in Elf, since we consider the right-hand sides the result of parsing. The relevant part for our description is:

$$\begin{aligned}
(x)^+ &= x \\
(\texttt{lambda } x \texttt{ . } e)^+ &= \texttt{lam ([}x\texttt{:exp]} \ (e)^+) \\
(e_1\ e_2)^+ &= \texttt{app } (e_1)^+ \ (e_2)^+ \\
(\texttt{let } x = e_1 \texttt{ in } e_2)^+ &= \texttt{let } (e_1)^+ \ (\texttt{[}x\texttt{:exp]} \ (e_2)^+) \\
\\
(\tau_1 \texttt{ -> } \tau_2)^+ &= \texttt{arrow } (\tau_1)^+ \ (\tau_2)^+
\end{aligned}$$

In the defining signature, we require no syntactic category for identifiers since all Mini-ML variables correspond directly to Elf variables. Thus we only need expressions (`exp` in Elf) and types (`tp` in Elf, since `type` is a keyword). The relevant types in Elf are

```
tp     : type.
exp    : type.

arrow  : tp -> tp -> tp.

lam    : (exp -> exp) -> exp.
app    : exp -> exp -> exp.
let    : exp -> (exp -> exp) -> exp.
```

Notice how the types of `lam` and `let` are analogous to the existential and universal quantification in the previous example. Some of the programming techniques will also be similar.

In the Elf implementation of type inference, the declaration of the predicate (judgment) `of` is:

```
of  : exp -> tp -> type.
```

That is, the first argument is a Mini-ML expression and the second argument is its Mini-ML type.

The rule for `lambda` illustrates universal quantification in a goal, as well as the use of embedded implication to represent assumptions. The scope of the quantifier `{x:exp}` below extends all the way to the end of the declaration. Thus the rule has one subgoal, as expected. The rule for `app` is straightforward. The first condition in the body of the third rule ensures that the let-bound term `E1` is well-typed in isolation. The second condition uses an extension of the technique used to implement `lam`. The let-bound variable in the body is replaced by a parameter `x`, and a rule is assumed that ensures that whenever `x` is encountered while type-checking `E2`, the expression `E1`

will be type-checked in its place. Note that the type A of x is universally quantified *within* the asserted rule, so that x can be given different (Mini-ML) types at different occurrences in E2.

```
of_lam      : of (lam E) (arrow A1 A2)
                <- {x:exp} of x A1 -> of (E x) A2.
of_app      : of (app E1 E2) A1
                <- of E1 (arrow A2 A1)
                <- of E2 A2.
of_let      : of (let E1 E2) A2
                <- of E1 A1
                <- {x:exp} ({A:tp} of x A <- of E1 A)
                             -> of (E2 x) A2.
```

As is the case of the previous example, these rules, and all the others for type inference, conform to the L_λ restriction of Elf.

We begin the operational semantics with the declaration of the evaluation predicate eval:

```
eval        : exp -> exp -> type.
```

The first argument is a Mini-ML expression, and the second argument is the value of that Mini-ML expression. The first rule below simply states that an abstraction is already in normal form. The second rule first evaluates the operator until it reduces to an abstraction. This of course may fail, and the other rules for application, not given here, are tried. If it succeeds, the argument is evaluated, and Elf application is used to represent the object-level substitution of the argument through the function.

```
eval_lam        : eval (lam E) (lam E).
eval_app_lam    : eval (app E1 E2) V
                        <- eval E1 (lam E1')
                        <- eval E2 V2
                        <- eval (E1' V2) V.
eval_let        : eval (let E1 E2) V2
                        <- eval E1 V1
                        <- eval (E2 V1) V2.
```

Notice that the rule eval_app_lam does not fall into L_λ, as the variable V2 in the term (E1' V2) is existential. However, notice that in the expected use of this rule, when the last subgoal is reached, E1' will already be bound to an Elf abstraction representing the function being applied. Hence, as we shall see in Chapter 11, the hard case of constraint solving that the syntactic restriction of L_λ is intended to avoid actually doesn't arise when this program is run. This observation also applies to eval_let. In Appendix B, this application is explored in some more detail. The expression syntax of Mini-ML is given in Appendix B.1, and the natural operational semantics is given in full in Appendix B.2.

5.4 Manipulating Proofs

In previous sections we have already seen some examples of proof objects created during program execution. To be more precise, when a fact is used directly to solve a subgoal, the corresponding proof object is merely the name of that rule. If a non-base rule is used to solve a subgoal, the resulting proof object is formed by applying the name of that rule to the proof objects obtained from solving all the subgoals in the body of the rule. The proof object corresponding to a universally quantified subgoal is a function from a term of the required type to the proof object corresponding to the body. The proof object for an embedded implication is a function from a proof of the assumption to a proof of the conclusion. That is, *given* a proof of the assumption, it *produces* a proof of the conclusion.

The name of a rule, then, can be used as a constructor in a program, anywhere that a normal constructor can be used. Thus, in addition to proof objects being built automatically for a query, a program can build proof objects explicitly, and, as we shall see in the example below, the type system of Elf will ensure that the constructed object is correct. Furthermore, an automatically constructed proof object can be captured, and manipulated in a way that is verified by the type system. To capture a proof object, the **sigma** quantifier is used. Let us consider a rather trivial example. For any boolean formula Φ, if the negation normal form is $\overline{\Phi}$, then

$$\overline{\Phi \wedge \Phi} = \overline{\Phi} \wedge \overline{\Phi}.$$

Now let α be the proof object corresponding to finding $\overline{\Phi}$. Clearly, the proof object for finding $\overline{\Phi \wedge \Phi}$ is just (**t_a** α α). Let us consider how to embody this in a very simple Elf program that implements this transformation. It's declaration is:

```
transform_proof : negn F F1 -> negn (and F F) (and F1 F1) -> type.
```

and the one rule needed would be

```
transform_proof P (t_a P P).
```

The declaration states that the first argument is the proof object for the original transformation, and the second argument is the proof object for the transformation of the conjoined expression. The Elf type system then guarantees that each rule for **transform_proof** actually maintains the required property.

This example is trivial, but of course the program could be extended to handle cases like $\Phi \vee \Phi$ (also trivial) and $\neg\Phi$, which would require a recursive traversal of the original proof object, switching occurrences of **t_o** and **t_no**, etc. An example query might be:

```
?- sigma [P : negn (n (a (prop v1) (prop v2))) X]
                                    transform_proof P P1.
```

In [117], a number of aspects of the meta-theory of the natural semantics of Mini-ML are coded in Elf and explained in considerable detail. As a simple example, consider

the subject reduction property of Mini-ML: evaluating an expression does not change its type. This can be expressed as an Elf program by defining a relation **sr** between an evaluation proof object, a type inference proof object for the original expression, and another for the resulting expression. That is,

```
sr : eval E V -> of E A -> of V A -> type.
```

and to choose an interesting example rule,

```
sr_eval_app_lam : sr (eval_app_lam P3 P2 P1) (of_app D2 D1) C
                     <- sr P1 D1 (of_lam C1)
                     <- sr P2 D2 C2
                     <- sr P3 (C1 _ C2) C.
```

That is, in the case of a λ abstraction applied to some expression, the transformation is performed transforming the proof for the abstraction, obtaining `(lam C1)` and the proof for the argument, obtaining `C2`, and then using these to construct the type inference proof for the result of actually performing the substitution. To see how this works, recall from above the rule **of_lam**. The proof object constructed by applying this rule will of course have **of_lam** as its principal functor. Following the rules for constructing proof objects, as given above, the argument will be a function that accepts an expression and a proof that it has the correct argument type, and returns a proof that the body of the abstraction will have the result type. In the first subgoal in the body of **sr_eval_app_lam** above, this function is obtained as `C1`. Then it is used in the last subgoal, obtaining the first argument from Elf type reconstruction, and the second from the previous subgoal, where transformation is performed for the argument, producing `C2`. Other cases, some of which are even more complicated, are described in detail in the above reference. Another example of this kind of reasoning is given in Appendix B.3, where a definition of a Mini-ML expression that is a value is given, followed by a program that transforms the proof object for an evaluation to one showing that the resulting expression is a value.

As we shall see in Chapter 11, the creation of proof objects will have consequences for the efficiency of Elf. Creating and storing them takes time and space, and sometimes substitutions must be applied to them. For this reason, it will be useful to detect when they are not used. Additionally, while proof objects resemble simply-typed λ-expressions at the *term* level, their *types* are somewhat more complicated. As types sometimes need to be compared during constraint solving, the complexity of these types can result in considerable cost. Even when they are not being compared, creating them and substituting through them also takes time, and storing them takes space.

5.5 Programming Methodology

Considerable experience has been accumulated over the last few years in programming in λProlog. Elf is a more recent language and has been used much less extensively. However, to understand Elf programming methodology we can draw upon the experience with λProlog since most λProlog programs can be directly translated into Elf.

The major observations about "typical" λProlog programs that can be made are as follows.

- While many programs fall into the L_λ subset of the language, there are also many that do not.
- Many programs that do not fall into L_λ would not trigger the expensive case of unification. This is because they breach the L_λ syntax with an application of two existential variables where that in the function position is already instantiated to an abstraction.
- Many programs that trigger the expensive case of unification would not do so if certain readily-identifiable constraints were deferred, and would still run correctly.
- The occurs check is needed for termination and correctness in some real (non-contrived) programs. This is particularly the case with theorem proving applications.
- Many constraints involve comparison of ground terms or assignment of a term to an existential variable. Frequently, the assignment is to a *new* variable, so that the occurs check can be omitted.
- Certain programs, like some theorem provers, may turn out to need the full generality of the constraints, without delay.
- A large proportion of terms in programs are of a form that can be seen as corresponding closely to first-order terms. For example, only a few of the syntactic constructs treated in a higher order abstract syntax tend to have λ abstractions, and the heads of rules tend to match with terms of the form $(f\ X_1 \cdots X_n)$ where f is a constructor and the rest are existential variables. These often match against a term that is of the form $(g\ t_1 \cdots t_m)$ where g is a constructor and the rest are terms, about which the same observation typically holds. If f and g are different the match fails, and otherwise we must have $n = m$, and so recursive assignment takes place. However, there is a caveat: unless the occurs check can be safely omitted, assignment can result in normalization to delete occurrences of the variable being assigned.
- An important case in the use of higher order abstract syntax is application of an existentially quantified variable instantiated with an abstraction to a universally quantified variable. The dual case of an equation between an instantiated existential variable and the application of a free variable to a universally quantified variable to essentially "remove" occurrences of the latter is also important and frequent.

In addition, certain observations can be made about Elf programs with respect to λProlog.

- Many Elf programs are essentially λProlog programs with additional meta-theoretic constraints, that are evaluated on typechecking but have virtually no runtime consequences.
- Otherwise, large parts of the execution of an Elf program are just λProlog execution. This is because program execution is often divided between disjoint phases of meta-programming (essentially λProlog) and meta-theoretic reasoning (Elf's proof object manipulation).
- Another style of Elf program tends to evolve by adding a couple of arguments to a λProlog, say to build proof objects.

Furthermore, some comments can be made about the use of the distinctive features of Elf.

- A large proportion of the proof objects created are never examined, except when they are printed at the end of a computation.
- In some cases, manipulating proof objects is a major part of the program.
- In many cases a lot of runtime typechecking can be avoided.

Finally, we note some other work on the applications of Elf. A number of verified theorem-provers have been written. Additionally, in [63], Hannan and Pfenning describe using Elf to verify a compiler from the untyped, call-by-value λ-calculus to the Categorical Abstract Machine. In [7], Anderson describes the use of Elf to extract programs in a functional language from proofs in first-order logic. Proof transformations for program optimization and specialization are also represented in Elf. This work involves some of the most computationally intensive Elf code written to date.

5.6 Empirical Study

As in the previous Chapter 4, we need to understand how programs behave at run time. Thus we provide measurements for the constraint solving load of a number of Elf programs. The measurements were obtained by instrumenting the constraint solver and other components in Version 0.2 of the CMU implementation of Elf. This implementation, unlike the CLP($\mathcal{R}$) compiler instrumented for Section 4.5, is relatively naive. Hence we are able to obtain relatively precise information about the nature of the programs and queries, relatively unpolluted by the specifics of the implementation, although the results are influenced somewhat by the fact that hard constraints are delayed.

5.6.1 Properties of Programs

Since our concern is with efficient implementation (and its interaction with language design), the properties of programs that we most need to study are the dynamic properties: how frequently do various phenomena arise when typical queries are executed?

This allows us to tune data structures and algorithms. On the other hand, to evaluate the possibility of syntactic restrictions, we also need to know what occurs syntactically in programs. We begin by discussing these syntactic properties and why they are of interest. Then we go on to discuss the dynamic properties.

Static Properties

L_λ vs. general variable applications. Because of our interest in the syntactic restriction to L_λ, we need to understand how often and why programs do not fall into this subset. Even within the L_λ subset, we can observe interesting static properties of programs. For example, many programs structurally recurse through an object language expression, where the object is represented using higher-order abstract syntax.
Type redundancy. Both in λProlog and Elf there is a potential for much redundant run-time type computation. In λProlog, this is due to polymorphism (see [101]), in Elf it is due to type dependency. Such redundancy can be detected statically. However, the question about the dynamic properties of programs remains: how much type computation remains after all redundant ones have been eliminated.
Level of indexing. This is an Elf-specific property of a program. Briefly, a (simple) type is a level 0 type family. A type family indexed by objects of level 0 type is a level 1 type family. In general, an type family indexed by objects whose type involves level n families is a family of level $n+1$. For example,

```
o : type.                          % propositions, level 0.
pf : o -> type.                    % proofs of propositions, level 1.
norm : pf A -> pf A -> type.       % proof transformations, level 2.
proper : norm P Q -> type.         % proper proof transformations, level 3.
```

This is of interest because the level of indexing determines the amount of potentially redundant type computation. Empirically, it can be observed that programs at level 2 or 3 have in some respects different runtime characteristics than programs at level 1. We have therefore separated out the queries of the higher-level. This also helps to separate out the part of our analysis that is directly relevant to λProlog, where all computation happens at levels 0 and 1 (due to the absence of dependency).

Dynamic Properties

The major dynamic properties studied here are substitution, unification and constraint solving.
Substitution. Substitution can be a significant factor limiting performance. It is thus important to analyze various forms of substitution that arise during execution. When measuring these, our concern is simple: substitutions with anything other than parameters (*Uvars*) result from the fragment of the language outside L_λ, so these represent substitutions that would have had to have been performed using Elf code if the L_λ restriction had been applied. Moreover, the relative frequency of parameter substitution suggests that it is crucial for it to be highly efficient, while general substitution is somewhat less critical. A proposal regarding efficient implementation of terms has been made in [127].

Unification and Constraint Satisfaction. We measure various aspects of unification and constraint satisfaction. Terms involved in equations (disagreement pairs) are classified as *Rigid* (constant head), *Uvars* (parameters, *i.e.*, temporary constants), *Evars* (simple logic variables), *Gvars* (generalized variables, *i.e.*, logic variables applied to distinct, dominated parameters [119]), *flexible* (compound terms with a logic variable at the head, but not a Gvar), *Abst* (a term beginning with a λ-abstraction), or *Quant* (a term beginning with a Π-quantification, in Elf only). These terms will be explained in Chapter 11.

One of our goals is to determine how close Elf computations come to Prolog computations in several respects:

- How many pairs, at least at the top level, require essentially Herbrand unification? These are the Rigid-Rigid and Evar-anything cases.
- How many pairs still have unique mgus, that is, Gvar-Gvar, or admit a unique strategy for constraint simplification, that is, Gvar-Rigid, Abst-anything, or Quant-anything?
- How often do the remaining cases arise (which are postponed to avoid branching)?
- How successful is rule indexing (as familiar from Prolog) to avoid calls to unification?

5.6.2 The Measurements

The two tables in each figure give data on five areas of interest, as follows:

- *All Unifications*
  The total gives an indication of computational content, while the breakdown indicates the usefulness of first-argument indexing and the amount of deep search.
  - *Unif* Total number of subgoal/head pairs to be unified.
  - *%Ind* Percentage of above total unifications avoided by rule indexing.
  - *%S* Percentage of total unifications that succeeded.
  - *%F* Percentage of total unifications that failed.
- *Dynamic Unifications*
  It is also useful to have this information for rules assumed through embedded implication, since indexing of such rules is more complicated, and compilation has a runtime cost.
  - *Dyn* Total number of subgoal/head pairs to be unified, where the head is from a rule assumed (dynamically) through embedded implication.
  - *%Ind, %S, %F* Percentages of number of unifications with heads from dynamic rules, as above.
- *Dynamic/Assume*
  By knowing how many rules are assumed dynamically, and on average how often

they are used, we can see whether it is worthwhile to index and compile such rules or whether they should be interpreted.

Ass	Number of rules assumed by implication.
U/Ass	Normalized ratio of total unifications with dynamic rules to number of rules assumed by implication.
AU/Ass	As above, but using only those rules where the unification was not avoided through indexing.

- *Disagreement Pairs*
  We study the kinds of disagreement pairs that arise to determine which kinds of unification dominate.

Tot	Total number of disagreement pairs examined throughout the computation.
%E-?	Percentage of disagreement pairs that involved a simple Evar.
%G-?	Percentage of disagreement pairs that involved a Gvar that is *not* a simple Evar.
%R	Percentage of disagreement pairs between two Rigid terms.
%A	Percentage of disagreement pairs between two abstractions.

- *Substitutions*
  Substitutions and abstractions (the inverse of Uvar substitutions) are expensive, and the efficiency of one can be improved at the expense of the other. Furthermore, some kinds of substitutions are more costly than others. Thus it is useful to know what kinds of substitutions arise, how often both substitution and abstraction arise, and their relative frequency.

Tot	Total number of substitutions for bound variables.
%Uv	Percentage of the above where a Uvar is substituted.
Abs	Number of abstractions over a Uvar.
Abs/Uv	Normalized ratio of such abstractions to substitutions of Uvars.

5.6.3 The Programs and Results

Figures 5.3 and 5.4 show the data for basic computation queries and proof manipulation queries respectively, for the range of programs. Thus Figure 5.3 is especially applicable to the understanding of λProlog programs, while Figure 5.4 measures Elf-specific behavior.

The examples used are as follows:

- Extraction — *Constructive theorem proving and program extraction* [7]

  This example involves a large number of level 2 judgments. Indexing is particularly effective here, and assumed rules are used unusually infrequently. Note that these examples do not include any basic computation.

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Mini-ML	15333	87	13	0	1532	93	7	0	67	22.87	1.61
Canonical	177	66	28	6	8	50	50	0	3	2.67	1.33
Prop	677	60	30	10	41	44	41	15	9	4.56	2.56
F-O	359	65	28	7	33	18	82	0	17	1.94	0.07
Forsythe	2087	38	23	39	16	25	75	0	10	1.60	1.20
Lam	240	50	40	10	26	80	15	5	4	6.50	1.25
Polylam	982	65	34	1	389	88	12	1	45	8.64	1.00
Records	2459	61	31	8	274	61	39	0	28	9.79	3.79
DeBruijn	451	25	39	36	5	40	60	0	5	1.00	0.60
CLS	278	0	32	68	0	-	-	-	0	-	-

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Mini-ML	8716	47	0	52	0	6411	98	0	0.00
Canonical	427	41	8	56	0	180	96	36	0.21
Prop	1681	54	0	45	1	202	100	8	0.04
F-O	438	40	6	58	0	108	100	58	0.54
Forsythe	5812	43	0	57	0	39	100	0	0.00
Lam	874	41	0	59	0	149	86	0	0.00
Polylam	2085	48	3	50	1	7907	89	81	0.01
Records	3880	46	3	53	0	1347	100	204	0.15
DeBruijn	1554	44	1	56	0	688	97	16	0.02
CLS	2455	36	0	65	0	0	-	0	-

Figure 5.3: Basic Computation

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Extraction	873	89	11	0	165	82	17	1	54	3.05	0.54
Mini-ML	2415	73	11	16	107	87	13	0	10	10.70	1.40
CPS	162	59	41	0	72	57	43	0	48	1.50	0.65
Prop	4957	67	25	8	509	71	14	15	71	7.17	2.10
F-O	1140	69	27	4	27	0	100	0	13	2.08	2.08
Lam	369	50	44	6	36	75	22	3	12	3.00	0.75
DeBruijn	627	20	44	36	77	51	30	19	24	3.21	1.58
CLS	333	30	42	28	0	-	-	-	0	-	-

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Extraction	1580	22	9	66	6	9016	96	1124	0.01
Mini-ML	5872	17	1	76	6	3644	96	55	0.02
CPS	592	24	34	54	0	1509	100	1029	0.68
Prop	13809	35	3	63	1	12040	99	443	0.04
F-O	6800	21	1	74	5	12716	99	38	0.00
Lam	3464	22	2	74	3	1825	94	83	0.05
DeBruijn	13441	15	1	71	13	14632	99	150	0.01
CLS	5227	23	0	77	0	2	-	0	-

Figure 5.4: Proof Manipulation

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Comp	5562	90	10	0	1532	92	8	0	67	22.87	1.61
ExpComp	7200	70	10	20	1798	80	8	12	87	10.67	4.30
ExpIndComp	7200	88	10	2	1798	92	8	0	87	10.67	4.30
Trans	2159	70	11	19	107	86	14	0	10	10.70	1.40
ExpTrans	5255	29	10	59	633	15	13	72	67	9.45	8.06
ExpIndTrans	5255	76	11	13	633	84	13	3	67	9.45	8.06

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Comp	2424	43	0	57	0	445	97	0	-
ExpComp	10765	24	4	56	17	22743	100	778	0.03
ExpIndComp	4251	32	10	52	8	15801	100	778	0.05
Trans	5709	17	1	76	7	3612	96	55	0.02
ExpTrans	27342	20	4	61	16	280679	97	2522	0.01
ExpIndTrans	13482	17	8	65	12	264399	98	2522	0.01

Program	Computation	Transformation
Implicit	1.30	2.48
Explicit	8.48	155.09
Explicit-Indexed	5.80	145.89

Figure 5.5: Mini-ML comparison

- Mini-ML [117]

  An implementation of Mini-ML, including type-checking, evaluation, and the type soundness proof. Because of the large number of cases, indexing has a stronger effect than in all other examples.

- CPS — *Interpretation of propositional logics and CPS conversions* [60, 142]

  Various forms of conversion of simply-typed terms to continuation-passing and exception-returning style. Substitutions are all parameter substitutions, and unification involves an unusually large number of Gvar-anything cases. The redundant type computations are very significant in this example — all the queries are level 2 judgments.

- Canonical — *Canonical forms in the simply-typed lambda-calculus* [139]

  Conversion of lambda-terms to canonical form. A small number of non-parameter substitutions arise, but mostly unification is first-order. Here, too, there is much redundant type computation.

- Prop — *Propositional Theorem Proving and Transformation* [66]

  This is mostly first-order. In the transformations between various proof formats (natural deduction and Hilbert calculi), a fairly large number of assumptions arise, and are quite heavily used. Unification involves a large number of Evar-anything cases.

- F-O — *First-order logic theorem proving and transformation*

  This includes a logic programming style theorem prover and transformation of execution trace to natural deductions. There is rather little abstraction.

- Forsythe — *Forsythe type checking*

  Forsythe is an Algol-like language with intersection types developed by Reynolds [145]. This example involves very few substitutions, all of which are parameter substitutions. Thus the runtime behavior suggests an almost entirely first-order program, which is not apparent from the code.

- Lam — *Lambda calculus convertibility*

  Normalization and equivalence proofs of terms in a typed λ-calculus. A relatively high percentage of the substitutions are non-parameter substitutions.

- Polylam — *Type inference in the polymorphic lambda calculus* [137]

  Type inference for the polymorphic λ-calculus involves postponed constraints, but mostly parameter substitutions. Unification can be highly non-deterministic. This is not directly reflected in the given tables, as this is the only one of our examples where any hard constraints are delayed at run time (and in only 10 instances). In fact, one of these hard constraints remains all the way to the end of the computation. This indicates that the input was not annotated with

enough type information (within the polymorphic type discipline, not within the framework).

- Records — *A lambda-calculus with records and polymorphism*

  Type checking for a lambda-calculus with records and polymorphism as described in [67]. This involves only parameter substitutions, and assumptions are heavily used.

- *DeBruijn* [63]

  A compiler from untyped λ-terms to terms using deBruijn indices, including a call-by-value operational semantics for source and target language. The proof manipulation queries check compiler correctness for concrete programs. Indexing works quite poorly, and an unusually large number of Abst-Abst cases arise in unification.

- *CLS* [63]

  A second compiler from terms in deBruijn representation to the CLS abstract machine. Simple queries execute the CLS machine on given programs, proof manipulation queries check compiler correctness for concrete programs. This is almost completely first-order.

Overall, the figures suggest quite strongly that most unification is either simple assignment or first-order (Herbrand) unification, around 95%, averaged over all examples. Similarly, substitution is the substitution of parameters for λ-bound variables in about 95% of the cases. The remaining 5% are substitution of constants, variables, or compound terms for bound variables. These figures do not count the substitution that may occur when clauses are copied, or unifications or substitutions that arise during type reconstruction.

Finally, we compare the Mini-ML program with a version written using explicit substitution, to evaluate the effects of a syntactic restriction along the lines of L_λ. The computation queries had to be cut down somewhat because of memory restrictions. In Figure 5.5 we show the same data as above for the computation and transformation queries with and without explicit substitution. We also show a version with explicit substitution with the substitution code rewritten to take better advantage of indexing. Then we compare the CPU times (in seconds) for the two sets of queries for all three versions of the program, using a slightly modified[2] Elf version 0.2 in SML/NJ version 0.80 on a DEC station 5000/200 with 64MB of memory and local paging. These results show that there is a clear efficiency disadvantage to the L_λ restriction. Note that the disadvantage is greater for the transformation queries, since a longer proof object is obtained, resulting in a more complicated proof transformation. Explicit substitution increases the size of the relevant code by 30%.[3] Substitutions dominate the

[2]The modification involves building proof objects only when needed for correctness.

[3]Actually, the meta-theory was not completely reduced to L_λ, because type dependencies in the verification code would lead to a very complex verification predicate. We estimate that the code size would increase an additional 5% and the computation time by much more than that.

computation time, basically because one meta-level β-reduction has been replaced by many substitutions. These substitutions should all be parameter (Uvar) substitutions, which suggests that some (but clearly not all) of the performance degradation could be recovered through efficient Uvar substitution. See the previous footnote on why non-parameter substitutions still arise in the proof transformation examples. We will draw heavily on these results in Chapter 11.

Part III

Implementation Techniques

Chapter 6

Implementation Strategy

The remainder of this thesis deals with efficient implementation of CLP systems. In this chapter, a general strategy is outlined for dealing with the efficiency issues introduced in Chapter 3, largely by taking advantage of the observations made in that chapter and illustrated in Chapters 4 and 5. The following chapters discuss the implementation techniques in detail, illustrated by the implementation of a CLP($\mathcal{R}$) compiler. Finally, a strategy for applying these techniques to λProlog and Elf is outlined.

Consider the basic implementation model for a CLP language, as shown in Figure 6.1. The *inference engine*, using a goal and the *rule base*, controls search through the rule base. During the search, it passes constraints to the *constraint solver*, which are then added to a collected set of constraints. If the newly augmented set is consistent, the answer **yes** is returned, and otherwise **no** is returned. Whenever the inference engine needs to backtrack through the search tree, it sends a **backtrack** signal to the constraint solver for every time the most recent remaining constraint is to be removed from the current set of constraints. The set of constraints currently stored in the constraint solver at any given time will often be referred to as the *solved form*. Finally, when it is time to output an answer constraint, the inference engine gives the constraint solver the set of variables on which the solved form is to be projected, and the result is output. To avoid confusion, note that our use of the term "inference engine" is not quite standard. In discussing Prolog systems, it tends to be used to describe both the search and unification components. Here we use it to refer only to the search (resolution) component.

Our discussion of implementation will focus on the following aspects:

- Suitable data structures for dealing with the incremental nature of constraint solving in the context of backtracking.
- Organization of the constraint solver, and assignment of additional responsibility to the inference engine, such that simple constraints are executed as efficiently as possible.
- A conceptual framework for specifying delayed constraints, and resulting data structures, so that constraints may be delayed and awakened efficiently.

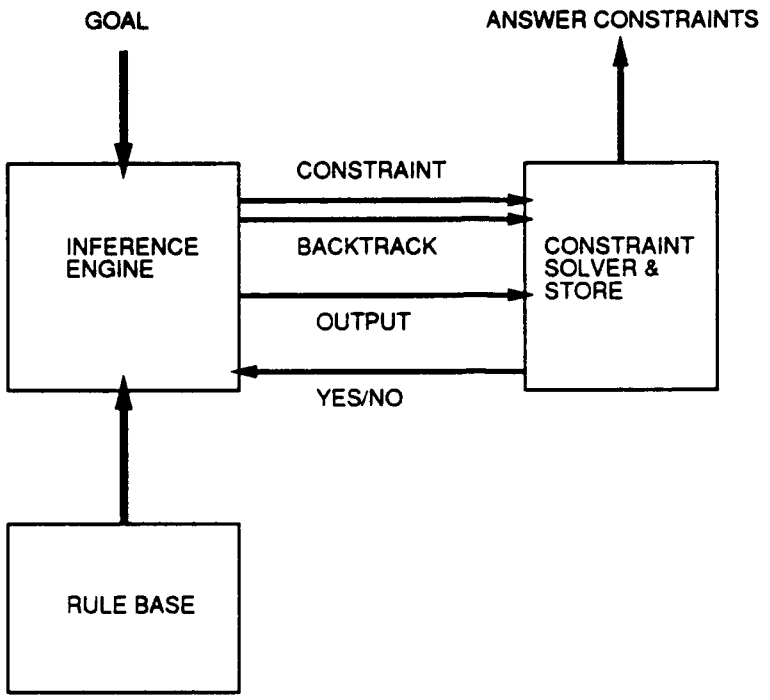


Figure 6.1: Basic Implementation Model

- Compile time optimization resulting in more efficient execution of simple cases of constraints.

In more detail, the implementation strategy can be divided into two broad categories. The first line of attack is to carefully design the run-time organization, data structures and algorithms. Then the resulting performance can be further improved upon through optimizing compilation.

6.1 Run-Time Strategy

The run-time strategy rests on the understanding that the simplest constraints occur most commonly in typical programs, and hence that it is important to solve these with minimal overhead. Powerful and general constraint-solving capability can often be very useful to programmers. However, in many domains and application areas, programmers do not make use of this generality all of the time. Programs might often contain very large numbers of constraints, most of which are quite simple. Furthermore, the collected set of constraints is typically changed by small amounts very often. It would not be acceptable if the performance of a CLP system able to solve very complex constraints was consequently very poor when solving simple constraints. Empirical and methodological justification for the strategy can be found in Chapters 3, 4 and 5.

The specifics of this strategy can be summarized as follows:

- Classify and prioritize constraints, reflecting:
  - estimated relative frequency of occurrence;
  - relative cost of solving.
- Treat constraint solving as generalized unification, thus making the Prolog case most important.
- Avoid general constraint solving machinery where possible.
- Design data structures to favor simple cases.
- Check for hard constraints that should be awakened without undue impact on the efficiency of constraint solving.
- To facilitate backtracking, store details of all *changes* to the collected constraint set, rather than backing up all or part of it at every change.

The organization and restriction of the constraint solver is addressed in Chapters 7 and 8, while data structures for incrementality and efficient backtracking are discussed in Chapter 9. These ideas were incorporated in a CLP($\mathcal{R}$) interpreter project in which the author collaborated, and further refined in the compiler-based system discussed below. The system was based on structure-sharing Prolog interpreter technology, as exemplified by C-Prolog. The performance on most Prolog programs is comparable with such Prolog interpreters, and performance for arithmetic constraints has been sufficiently good for non-trivial applications to be implemented.

6.2 Compile-Time Strategy

The usual approach to the compilation of Prolog rests on two basic ideas:

- The unification algorithm can usefully be partially evaluated with respect to the partially constructed terms appearing in rules.
- The backtracking-based execution model for Prolog can be mapped onto the stack-based paradigm for executing conventional programming languages.

Most modern Prolog compilers are based on the Warren Abstract Machine (WAM), defined by D. H. D. Warren [189]. This is an abstract machine specification including various stacks and registers, together with a special-purpose target instruction set for compiling Prolog. The reader seeking a tutorial introduction to this style of Prolog compilation should consult the book by Aït-Kaci [2]. The WAM has been used in the implementation of a number of commercial systems, many emulating the WAM at run time, others generating native code.

Since Prolog performance of CLP systems is a high priority, it is natural to want to apply the usual Prolog compilation techniques to these languages. However, the question is whether compilation can be of benefit to solving constraints more general than Prolog unification. Chapter 10 gives a detailed description of an approach to compiling CLP languages, with a particular emphasis on CLP($\mathcal{R}$). This rests partly on the assumption that certain simple kinds of constraints can be detected easily at compile time, so that with the use of peephole optimization these might be executed more efficiently than if such an analysis were to be carried out at run time, as mentioned above. For compiling CLP($\mathcal{R}$), the Constraint Logic Arithmetic Machine (CLAM) is described in Chapter 10. This is an extension of the WAM with instructions and extra structures for arithmetic constraints. These ideas have been incorporated in IBM's compiler-based implementation of CLP($\mathcal{R}$), in which the author participated. This has now been widely available and in use for over two years, and has been the platform for a number of large applications. Significantly, the performance of the CLAM-based CLP($\mathcal{R}$) system on Prolog programs is close to that of the major commercial Prolog implementations. Furthermore, there is a significant performance improvement over the interpreter mentioned above for arithmetic constraints.

It has long been observed that Prolog programs could be executed more efficiently if they were specialized with respect to certain classes of *allowed queries*. Given a class of allowed queries specified in some formal language, it is possible to analyze a program globally to determine various properties of its execution for queries that are in this class. These properties include determinism, types of variables, and modes (information about a variable having a certain level of instantiation at a certain point in an execution). The best known technique for such *global analysis* is *abstract interpretation*. This is based on the idea of executing a program on a special domain known as the *abstract domain*, which captures the kinds of properties that are desired. The program is executed with respect to a query that represents the set of allowed queries for the program, using arguments built up from the abstract domain. For an overview of abstract interpretation, the reader should consult [40], although the literature on this topic is quite large. In their respective dissertations, Van Roy [148] and Taylor [181] have shown that abstract interpretation has the potential to enable Prolog compilers to achieve performance comparable to that of compilers for conventional languages.

Abstract interpretation of CLP programs was described by Marriott and Søndergaard [114], and Jørgensen *et al* [91] describe various global analysis techniques for CLP($\mathcal{R}$), but otherwise little work has been done on global analysis for CLP languages. The approach to compiling CLP languages, described in detail in Chapter 10 rests on the the following observations:

- Local optimizations can help to solve obviously simple constraints efficiently.

- Parts of programs (say modules) tend to be called in a number of specific ways, some of which result in apparently complex constraints collapsing into simple (or simpler) cases.

- Such special cases can be detected by state-of-the-art program analysis techniques (beyond the scope of this thesis) and efficient code can be generated by multiple specialization of these modules.
- Abstract machines can be designed to take advantage of the above compilation ideas.

While a compiler based on these techniques has not yet been implemented for CLP($\mathcal{R}$), the CLAM has been extended to take advantage of the sort of information that can be obtained from global analysis of CLP($\mathcal{R}$) programs. The CLAM emulator that forms the core of the compiler-based CLP($\mathcal{R}$) system described above has been extended to obtain empirics on hand-compiled programs. This has demonstrated dramatic performance improvements for a number of programs with various allowed queries. These results are also described in Chapter 10.

6.3 Summary

This major difference between this part of this thesis and other work that has been done on CLP($\mathcal{R}$) implementation is the emphasis on handling frequently-occurring special cases of constraint solving well. This difference in emphasis results from the fact that these implementation efforts have been driven by a study of programming methodology, and thus static and dynamic properties of programs, rather than a classical (operations research-oriented) view of constraint solving.

Chapter 7

Organization of Solvers

The basic architecture for a CLP system as outlined in Figure 6.1 in Chapter 6 is in practice not adequate for two reasons. First, it does not facilitate delay, which we have argued to be essential. Second, it assumes that constraint solving is all handled by a monolithic "black box", the details of which are invisible to the inference engine and other modules. For most CLP languages, such an abstraction cannot be adhered to in practice if acceptable performance is to be attained. It needs to be broken down in two ways:

1. A typical constraint domain will consist of various subdomains, with possibly more than one kind of constraint on each. Different individual solvers will be needed to solve different kinds of constraints, to hide the complexity of solving one kind from the algorithm implementing another.

2. Details of constraint solving will have to be made available to the inference engine, at least to facilitate rule indexing.

In this chapter we develop an implementation model to facilitate delay mechanisms and for multiple constraint solvers, which may interact more closely with the inference engine. After generalizing the basic implementation model to one with delay, we explore the required properties of multiple constraint solvers for efficiency and correctness. Finally, we use the organization of the constraint solver modules in the CLP($\mathcal{R}$) interpreter (and subsequent compiler-based implementation) to illustrate how these principles are applied in practice.

7.1 Adding Delay to The Implementation Model

Figure 7.1 shows a generalization of the basic implementation model of Figure 6.1 to incorporate delay. This is still a generic model, in that it is not specialized for any particular domain. However, it illustrates some of the ideas that will be taken further below. First we have the notion of a constraint being too "hard" for the general constraint solver, and thus being stored in the delay pool. This is really an active agent, constantly monitoring the constraint solver to check when a constraint becomes

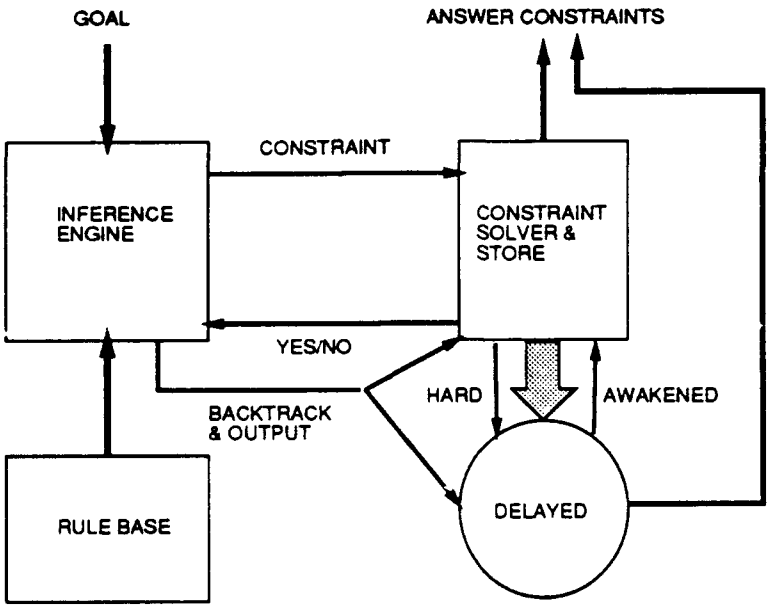


Figure 7.1: Generic Implementation Model with Delay

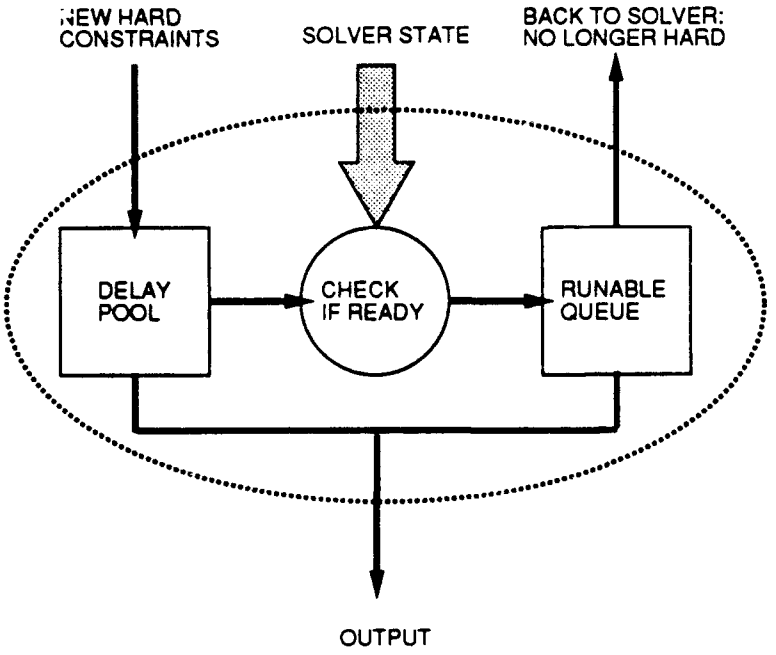


Figure 7.2: Delay/Wakeup Mechanism

"runnable", so that it can be awakened before the next inference. When a constraint is awakened it re-enters the solver as if it were a new constraint, except it is now known not to be hard (although its processing may give rise to further hard constraints). The runnable constraints must be queued because one change to the solver state can result in several constraints being awakened. The first of these can make another change to the solver state awakening more constraints, and so on. Figure 7.2 describes this in some more detail. Henceforth we dispense with representing flow of control (for modules such as backtracking and output) in these diagrams, and just note that every component of the constraint handling mechanism conceptually must be able to handle these. The detailed implementation of the delay/wakeup mechanism can actually be quite complex. The conceptual framework and implementation strategy will be described in detail in Chapter 8.

7.2 Multiple Constraint Solvers

The most straightforward use of multiple constraint solvers is to deal with composite domains: those composed from a number of simple domains. For example, the domain of CLP($\mathcal{R}$) is composed of uninterpreted functors and real arithmetic. In such cases it is desirable to combine multiple solvers because algorithms are typically available for the simple domains. Then the collected set of constraints is deemed inconsistent if and only if one of the solvers finds that it has an inconsistent store, but the whole point is that each solver should see only the constraints on that domain. For obvious efficiency reasons it is necessary to limit the number of constraints in each store. Most come directly from the input constraints, but others can be generated by the other solvers, to link all the constraints together correctly. Thus we need a formal understanding of the minimal amount of information that must be shared.

Typically, the simple domains do not share function symbols. This case has been studied quite heavily. In [130], it is shown that it is sufficient for each solver to export only the equations between pairs of variables to the other solvers, and disjunctions of such equations that are implied by its store. For solvers that cannot imply disjunctions of such equalities (so-called *convex* domains), only individual equations are transferred. The result is interesting both in theory and practice. Other, more specific, work on such combinations of solvers includes that by Kaplan [92] on the theory of arrays with constant indices, by Shostak [160] on Presburger arithmetic with uninterpreted function symbols, and by Suzuki and Jefferson [178] on Presburger arithmetic arrays and uninterpreted function symbols.

Unfortunately, the property required for this result is too restrictive for an important use of multiple solvers. Sometimes a single domain can include constraints of differing difficulty, so a low-overhead solver is desirable for the easier constraints, and a more complex, higher-overhead solver is used for the rest. As will be discussed below, this is the case for CLP($\mathcal{R}$). It is a very valuable technique for implementing CLP systems efficiently, but the above correctness result does not hold, as often function symbols are shared. Furthermore, no corresponding result is available, so that ensuring the

correctness of multiple solvers combined in this way must be done on a case by case basis.

For this reason, we formalize the exact correctness criterion. Consider a CLP system with a solver component consisting of n solvers S_1 to S_n, with respective constraint stores C_1 to C_n. The function ϕ maps a constraint to a solver index: an integer from 1 to n. The function ψ selects all the relevant constraints from the stores of the other solvers. In general this function will have to be conservative, but if we are to obtain the benefits of multiple solvers, it should not be too conservative.

When a new constraint arrives, say, c, then we require that

$$\models \tilde{\exists}(C \wedge c) \text{ iff } i = \phi(c) \wedge S_i \models \tilde{\exists}(C_i \wedge \psi(i, C_1, \cdots, C_n) \wedge c).$$

We will have nothing more to say about this correctness issue, and will not prove the correctness of the organization of solvers in CLP($\mathcal{R}$). However, it is worth looking at an example to get an intuitive feeling for what can go wrong. As was shown in Chapter 3, Prolog III tuple concatenation constraints (really word equations), give rise to integer constraints. Let us consider the execution of the query

```
?- U = X.Y, |U| = 3, |X| > 1, |Y| > 1.
```

The constraints in this query are unsatisfiable: if U is of length 3 and both X and Y are of length greater than 1, one of the latter must have non-integral length, which would not make sense. Let us consider in some more detail how such a query would be solved. When the first constraint is selected, a number of additional integer constraints result:

1. `|U| = |X| + |Y|`
2. `|U| >= 0`
3. `|X| >= 0`
4. `|Y| >= 0`

Then the remaining constraints only interact with the integer solver, and in conjunction with the four integer constraints resulting from the original word equation, have no integer solution. The interesting point is that Prolog III does not have an integer solver. Now it is not sufficient to mark the length of any word variable to be integral, since the rational arithmetic solver only determines that there is a rational solution. In the example above a rational solution exists — in fact there are infinitely many, and this is the problem. Because non-integer solutions are actually not generated in this case, Prolog III fails to detect that the query above is unsatisfiable. In fact this is quite consistent with the stated behavior of Prolog III, since word equations are delayed until the first argument of a concatenation is of known length. Hence we have to go deeper to see what the problem is. Consider the query

```
?- U = X.Y, |X| = 2, |U| > 2, |U| < 3.
```

The query is again unsatisfiable, but the word equation is awakened by the integer constraint `|X| = 2`, so a definitive answer would be expected. However, the integer equations generated do have a rational solution, so Prolog III again fails to detect that the query is unsatisfiable. Finally, it is reasonable to ask what model of solving word equations could be supported by just solving rational arithmetic constraints. For an equation of the form `X = Y + Z` to have an integer solution whenever it has an arithmetic solution, any two of the variables must be ground. This means that only local propagation can be supported in this way, in which case the arithmetic equations on lengths of word equations become quite useless.

In summary, while the benefits of using multiple communicating constraint solvers are considerable in practice, the system designer must consider the correctness issues carefully. Implementing a CLP system with such a constraint solving structure is a non-trivial engineering problem.

7.3 Interpreting CLP($\mathcal{R}$)

The organization of constraint solving in a CLP($\mathcal{R}$) interpreter provides a good opportunity to study the runtime aspects of constraint solving. The material in this section is discussed from a somewhat different viewpoint in [88].

Here the generic implementation model is modified with the aims of:

1. solving Herbrand (unification) constraints with the efficiency of a Prolog interpreter;
2. performing simple arithmetic test and local propagation efficiently;
3. solving linear equations without concern for unrelated linear inequalities.

To satisfy the first objective, it was clear that the inference engine would have to be integrated with a Prolog-style unifier for Herbrand constraints. The engine is an adaptation of a structure-sharing Prolog interpreter. There are two central data structures, both of which are stacks. The major step performed by the engine is the creation of bindings of certain variables; this is activated either in the reduction step for a subgoal, or when certain kinds of equations are encountered. The following section elaborates on this. As will be described below, when arithmetic constraints are encountered, they are passed to the arithmetic constraint solver, which is really a cluster of three constraint solvers, as follows:

- an *interface*, which evaluates complex arithmetic expressions and transforms constraints to a small number of standard forms;
- an *equation solver*, which deals with linear arithmetic equations that are too complicated to be handled in the unifier and interface;
- an *inequality solver*, which deals with linear inequalities;

In this case, as has been mentioned before, nonlinear equations are considered hard constraints, and are delayed.

The interface between the unifier and linear solvers transforms constraints into a standard form. In this transformation, the input constraint is broken up into parts that are handled by different modules of the solver. The interface does some constraint satisfaction on its own, thus further lessening the need to use the solver. The equation solver maintains its equations in *parametric solved form*, that is, some variables arising from program execution are described in terms of parametric variables. The equation solver may need to send some equations to the inequality solver. The inequality solver may communicate back to the equation solver. This happens when *implicit equalities*, that is, equations entailed only by non-strict inequalities, have been computed. The nonlinear constraint handler obtains nonlinear equations to be delayed from the interface. Based on information in the linear equality solver, it sometimes awakens these constraints and ships them to that linear equation solver. The output module then needs information from the engine/unifier, both linear solvers, and the nonlinear handler. The details of output are beyond the scope of this thesis: see [84] for an account of output in CLP($\mathcal{R}$). Figure 7.3 summarizes the activities allocated to the various modules. In what follows, we describe the conceptual details behind the implementation

7.3.1 Unification

Equations between non-arithmetic terms are handled by a standard unification algorithm to ensure that Prolog programs execute with Prolog speed. Furthermore, this unification algorithm is closely associated with the inference engine so that terms can be used for rule indexing. Many other kinds of equations, however, can also be handled using the basic binding mechanism inherent in a unification algorithm. To define what these equations are, first define that a *solver variable* is one that appears in the collection of constraints held by the arithmetic constraint solver. The kinds of constraints that are solved directly within the unifier are then described as follows:

- an equation between two non-solver variables;
- an equation between a non-solver variable and a solver variable;
- an equation involving an uninterpreted functor;
- an equation or inequality between two numbers, and
- an equation between a non-solver variable and a number.

The table in Figure 7.4[1] summarizes the binding mechanism used in the unification algorithm in the engine. As is usual, a variable can be either bound by pointing to another term structure or to another variable forming a chain of references. We note,

[1] All variables mentioned there have been fully dereferenced. The empty slots represent the symmetric cases.

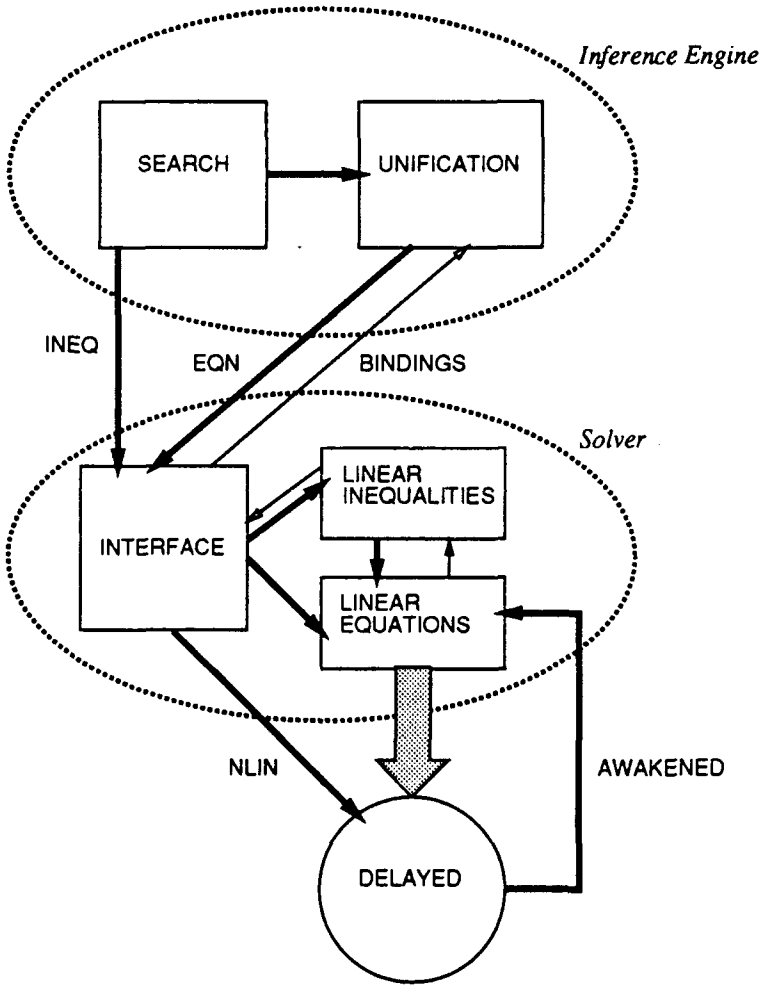


Figure 7.3: CLP($\mathcal{R}$) Solver Organization

CLP($\mathcal{R}$) Unification Table					
T2	T1				
	var	s_var	number	func	arith func
var	*bind*				
s_var	$T1 \rightarrow T2$	solver			
number	$T1 \rightarrow T2$	solver	$T1 = T2$?		
func	$T1 \rightarrow T2$	n/a	n/a	$T1 = T2$?	
arith func	interface	interface	interface	n/a	interface

Legend:

bind	- Bind T1 to T2, or bind T2 to T1
T1 → T2	- Bind T1 to T2
var	- ordinary free variable
s_var	- solver variable
func	- an uninterpreted functor
arith func	- an arithmetic functor
$T1 = T2$?	- are the topmost symbols equal?
n/a	- not applicable (corresponds to a type error)

Figure 7.4: Unification Table for CLP($\mathcal{R}$) Interpreter

however, that unlike in Prolog where a variable can only be free or bound to a term, variables in CLP($\mathcal{R}$) can belong to one of several distinct classes. In addition, a variable can also be bound by means of an *implicit binding*, as explained below.

Some of the different cases covered in the unification table are:

- *non-solver variable and non-solver variable;*
  Choose one variable, and this choice may depend on the current run-time structure, and then bind this variable to the other.
- *non-solver variable and solver variable;*
  To avoid invoking the constraint solver, the non-solver variable is bound to the solver variable, rather than vice versa. This choice is crucial for efficiency.
- *functor and any term;*
  In case the term is a non-solver variable, a straightforward binding of this variable to the other term is performed. In case the term contains a functor, the principal functors of both terms are checked to see if they are equal. If so, then their corresponding arguments are recursively unified.

- *number and number;*
  Because we use a floating-point implementation of numbers, we say that two numbers are equal if they are sufficiently close to one another.

- *non-solver variable and number;*
  Since a non-solver variable does not appear in the solver, the variable can be simply bound to the number.

- *solver variable and solver variable or number;*
  The interface, and then perhaps the constraint solver, are invoked.

- *arithmetic term and anything;*
  Here also the interface, and then perhaps the constraint solver, are invoked.

This should become clearer as a result of the examples later in this section. Now let us consider the special class of bindings between a variable and a number. Such a binding can occur in

- the unifier, when a non-solver variable is equated with a number that appears in the program;

- the interface, when a non-solver variable is equated to a directly evaluable arithmetic expression;

- the equation solver, when a system of equations is solved and it follows that certain variables must take specific values;

- the inequality solver, as a consequence of an implicit equality, that is, an equation entailed by a number of non-strict inequalities.

The bindings in the first two cases are of an explicit nature, and have their counterparts in standard logic programming systems. In the remaining two cases, the bindings occur in an implicit manner. For fast access to all these values, they are collected together in a central data structure, an *implicit binding* array, and this structure is accessible from all modules in the system. Thus the implicit binding mechanism plays the role of a cache.

Now that we have described the mechanism, let us consider its significance. The empirical results in Section 5.6 show that a very large fraction of disagreement pairs are between a non-solver variable and an arithmetic variable (%VP). These are essentially arithmetic equations, but the binding mechanism we have described does not treat them that way, and thus drastically reduces the number of constraints added to the equality solver. Furthermore, especially in the larger programs, the number of syntactic disagreement pairs tends to dwarf the number of arithmetic constraints, showing the importance of handling syntactic unification with minimal overhead.

Finally, we need to consider why this approach of doing as much constraint solving as possible by unification cannot reasonably be taken further for CLP($\mathcal{R}$). To deal with an equation of the form $x = t$, where t is an arithmetic term and x is a non-solver

variable, it would be quickest simply to bind the variable x to some representation of t. Such equations are, of course, always consistent with the constraints already stored in the solver, and hence the solver need not be invoked. It would become necessary to dereference such variables x in any term encountered later. The solver would then be invoked only when the equation at hand, after having been dereferenced, does not contain a non-solver variable as one operand. This implementation scheme, however, has two main drawbacks:

- *Cyclic bindings;*
  To implement dereferencing, the problem of cyclic references must be dealt with. This problem is analogous to the "occurs check" problem when dealing with non-arithmetic terms. An important difference, however, is that an arithmetic equation whose representation contains a cyclic binding is not necessarily unsatisfiable. Further, it is not possible to argue, as is done for most Prologs, that cyclic bindings are pathological events and so the problem can be ignored.

- *Thrashing behavior when backtracking;*
  The use of arithmetic bindings can lead to long binding chains, which in turn can cause a thrashing-like behavior when backtracking takes place. For example, consider the program

```
loop(0).
loop(I) :-
        I > 0,
        p,
        loop(I - 1).

?- loop(N).
```

  whose intent is to perform some operation p a total of N times, where N is given as a number. Let I_1, I_2, etc, denote the various occurrences of the variable I created during execution. In the first iteration, I_1 will be bound to the number N; then I_2 will be bound to the term $I_1 - 1$, I_3 to $(I_1 - 1) - 1$, and in general, I_j will be bound to a term of height $j - 1$. Now consider the unit rule and the recursive call above, and note that in the j^{th} iteration, an equation will be generated to compare the term $I_j - 1$ with 0. The cost of evaluating this equation, because of the bindings associated with I_j, is proportional to j. Thus the overall cost of executing the program, ignoring the cost of executing p, is proportional to N^2 in the best case. In contrast, if these equations are handled by a constraint solver, the cost can be linear in N.

Arithmetic bindings are not employed in the CLP($\mathcal{R}$) system because of these two problems.

7.3.2 The Interface

This module is called from the engine whenever a constraint contains an arithmetic term. The interface first simplifies the input constraint by evaluating the arithmetic expressions. If, as a result of this, the constraint is completely ground, then the appropriate test is made. If exactly one non-solver variable appears in the constraint and if the constraint is an equation, then an implicit binding is made, providing that it is easy to solve for this variable. In all other cases, the solver is invoked.

Before the interface invokes the solver, it first transforms the simplified constraint into a set of smaller constraints in canonical form. A constraint in canonical form is one of:

- an equation of the form
$$\sum_{i=1}^{n} c_i X_i = d$$
- an inequality of the form
$$\sum_{i=1}^{n} c_i X_i > d$$
- an inequality of the form
$$\sum_{i=1}^{n} c_i X_i \geq d$$
- an equation of the form $Variable = c * Variable * Variable$

where c, c_i and d are real numbers, and all X_i are variables.

7.3.3 Linear Equations and Inequalities

The equality module is invoked in one of three ways.

1. From the interface with a linear equation in a form described above.
2. From the inequality solver with an implicit equality,
3. From the nonlinear constraint handler with an equation that has been awakened.

The equality solver returns *true* if the satisfiable constraints already stored in the solver as a whole are consistent with the input equation, and *false* otherwise.

The central data structure in the equality solver is a tableau that stores, in each row, a representation of some variable in the form of a linear combination of parametric variables. The basic method of solution is a form of Gaussian elimination. A critical operation is that of simplifying substitutions, that is, the replacement of a chosen parametric variable by an equivalent parametric expression. Another critical operation is the communication between the equality and inequality solvers.

The Equation Tableau

The set of solver variables sent from the interface is partitioned by the equality solver into two sets of variables: *parametric* and *non-parametric* variables. Linear equations are stored in *parametric solved form*: using a matrix notation for convenience, this is a possibly empty collection of equations of which the i^{th} one is of the form:

$$X_i = b_i + c_{i,1} * T_{i,1} + c_{i,2} * T_{i,2} + \cdots + c_{i,n} * T_{i,n}$$

where $n \geq 0$, X_i is a non-parametric variable, the $c_{i,j}, 1 \leq j \leq n$, are real number coefficients, and the $T_{i,j}, 1 \leq j \leq n$, are parametric variables. The variable X_i is called the *subject* of the equation.

Adding a New Linear Equation

Call the initial (empty) parametric solved form θ_0. We now describe, given a parametric solved form θ_{j-1} and an input linear equation E_j, $j \geq 1$, how to determine whether $\theta_{j-1} \wedge E_j$ is solvable. If so, we obtain a new solved form θ_j.

Let C_j be the result of substituting[2] out each non-parametric variable X in E_j, where X appears in the tableau θ_{j-1}, with the corresponding parametric form for X.

(1) If C_j is now of the form $0 = 0$, then $\theta_j = \theta_{j-1}$, that is, the input constraint is already implied by θ_{j-1}.

(2) If C_j is of the form $0 = c$ where c is a non-zero real number, then the collection $E_1 \wedge E_2 \wedge \ldots \wedge E_j$ is inconsistent, and so θ_j does not exist. We simply return negatively.

(3) If C_j contains new variables, not currently in the solver, say $X_1, X_2, \ldots, X_m$, $m \geq 1$, then choose one of them, say X_1, and rewrite C_j with X_1 as the subject. X_1 becomes a non-parametric variable and the remaining variables $X_2, \ldots, X_m$ become new parametric variables. The resulting equation is then added to θ_{j-1} to obtain θ_j.

(4) If C_j contains no variables other than parametric variables, then choose one, say T, and rewrite C_j with T as the subject. For efficiency reasons, the choice of such a T is not arbitrary:

- we choose T, if possible, such that it does not appear in the tableau of the inequality solver;
- if, however, all the variables in C_j appear in the inequality solver, then we choose, if possible, T such that T is a *basic* variable (the precise definition of basic is not needed here) in the inequality solver;
- if, however, all the variables in C_j appear in the inequality solver as nonbasic variables, then we may arbitrarily choose one of them to be T.

[2]Substitutions involve both replacement of terms for variables and collection of like terms.

Now the resulting equation, call it C'_j, defines a substitution that enables *parameter elimination.* That is, θ_j is obtained from θ_{j-1} by substituting out the variable T using this substitution, and then adding the equation C'_j to the tableau. Note that the variable T, once eliminated in this way, will never become a parametric variable again except possibly as a result of backtracking. We will see later that the equation C'_j may need to be exported to the inequality solver. If so, the truth value returned by the equality solver is the answer subsequently received from the inequality solver.

In summary, the equality solver returns affirmatively in case (1) and negatively in (2). In case (3), if $m = 1$, we must consider the possibility of nonlinear constraints being awakened as a result of the input equation. Otherwise, we simply return affirmatively. Finally, in case (4), we must address the question of whether the inequality solver need be invoked.

Awakening Delayed Constraints

When variables become ground (and this can happen in steps (3) and (4) above), it will be necessary to communicate this fact to the nonlinear constraint handler. However, it is important that the nonlinear module be invoked only *after* the current equation has been completely dealt with. This is because the nonlinear module can itself invoke the equality solver. Some complications arise in the implementation because of these two facts:

- the equality solver uses one set of global data structures, and
- the equality solver and nonlinear handler are recursively dependent on each other.

We observe here that a negative response from the equality solver may be caused by a nonlinear constraint that has been awakened.

Calling the Inequality Solver

The inequality solver is called by the equality solver if:

- we fall into case (4) above, and
- every variable in the equation C'_j appears in the tableau of the inequality solver;

in which case the equation input to the inequality solver is C'_j. The value returned by the equality solver is just that returned by the inequality solver.

The formal proof that only the above-mentioned kind of equations need be sent to the inequality solver is beyond the scope of this thesis. Intuitively, we may reason as follows: specific solutions to a parametric solved form can be obtained by simply instantiating the parametric variables with arbitrary values. In particular, we can use values consistent with the constraints about parametric variables in the inequality

solver. Thus the only new additions to the equality solver that may be affected by the inequality solver are those that make a parametric variable, which appears in the inequality solver, no longer independent from the other parametric variables. This happens only as described above. For a detailed description of linear inequality solving in CLP($\mathcal{R}$), the interested reader should consult [174].

The empirical results in Section 4.5 show that not only are arithmetic inequalities less common than explicit arithmetic equations, but also that this treatment of inequalities often avoids the use of the Simplex solver. This is because the variables involved are often ground by the time the inequalities are encountered, and they essentially just behave like tests.

7.3.4 Some Examples of Constraint Flow

Let us begin by considering the query

```
?- X * X + Y * Y > 0, X + Y = 10, X - Y = 8.
```

The execution of this query proceeds as follows:

1. The first constraint is broken up into the three constraints

   (a) T_1 = X * X

   (b) T_2 = Y * Y

   (c) $T_1 + T_2 > 0$

   of which the first two are delayed, and the third is passed to the inequality solver.

2. The second constraint results in the parametric equation X = 10 - Y being passed to the equality solver, which stores it.

3. The third constraint is passed to the equality solver, which rewrites it using the previously stored parametric equation, and solves it giving Y = 1.

4. This value for Y is substituted throughout the equality solver, reducing the above parametric equation to X = 9.

5. The bindings for X and Y are propagated back to the unifier.

6. The same bindings are used to awaken the two delayed nonlinear equations, which are put in the runnable queue as

   (a) T_1 = 1

   (b) T_2 = 81

7. T_1 = 1 is taken from the head of the runnable queue, and passed to the equality solver, which considers T_1 to be parametric since it was involved in an inequality. Since all variables in the equation are then parametric, it is passed to the inequality solver, which essentially rewrites its one inequality to $T_2 > -1$.

8. T_2 = 81 is taken from the runnable queue, and similarly passed to the inequality solver, which simply confirms that 81 > -1 is true.

To further illustrate the role of the unifier, let us consider the query:

```
?- Y + 3 - Z = 0, f(X,Y) = f(U,U), X = 4.
```

Its execution proceeds as follows:

1. The first constraint results in, say, the parametric equation Y = Z - 3 being stored in the equality solver, and both Y and Z being solver variables.

2. The second (Herbrand) equation is treated by the unifier, which in turn generates the equations

   (a) X = U
   This is treated by the unifier, say binding X to U.

   (b) Y = U
   This again is treated by the unifier. Since Y is a solver variable, U must be bound to Y.

3. In solving the third constraint, X deferences to Y, which is a solver variable. Hence, the equation Y = 4 must be passed to the equality solver, which uses its stored parametric equation to rewrite it, resulting in Z = 7, which is passed back to the unifier.

7.4 Summary

We have pointed out the importance of carefully organizing multiple constraint solvers in a CLP system to improve efficiency. It is of central importance that simpler, more frequently occurring classes of constraints be spared the overhead of general constraint solving. Furthermore, different algorithms must be combined to solve constraints on compound domains, and some aspects of constraint solving need to be available to the inference mechanism for reasons such as the requirement for rule indexing. It has also been pointed out that there is a correctness issue involved in combining multiple solvers, for which there seems to be no simple recipe. This is complicated by the fact that extra communication between solvers may be required for efficiency in addition to correctness. Finally, the organization of solvers in the implementations of CLP($\mathcal{R}$) demonstrate all of the three motivations for not treating a constraint solver as just a single "black box".

Chapter 8

Managing Hard Constraints

A standard compromise approach to obtaining efficient constraint solvers has been to design a partial solver. That is, one that solves only a subclass of constraints, the *directly solvable* ones. These of course may or may not be consistent with the state of the solver. The remaining constraints, the *hard* ones, are simply delayed from consideration when they are first encountered; a hard constraint is reconsidered only when the constraint store contains sufficient information to reduce it into a directly solvable form. It may then be processed, resulting in success or failure, or even more hard constraints. In the CLP($\mathcal{R}$) system, for example, nonlinear arithmetic constraints are classified as hard constraints, and they are delayed until they become linear.

The key implementation issue is how to process efficiently just those delayed constraints that are affected as a result of a new input constraint. Specifically, the cost of processing a change to the current collection of delayed constraints should be related to the number and size of the delayed constraints affected by the change, and not to all the delayed constraints. The following two items seem necessary to achieve this end.

First is a notion of how far a delayed constraint is from being awakened. For example, it is useful to distinguish the delayed CLP($\mathcal{R}$) constraint $X = max(Y, Z)$, which awaits the grounding of Y and Z, from the constraint $X = max(5, Z)$, which awaits the grounding of Z. This is because, in general, a delayed constraint is awakened by not one but a conjunction of several input constraints. When a subset of such input constraints has already been encountered, the runtime structure should relate the delayed constraint to just the remaining kinds of constraints that will awaken it.

The other item is some data structure, call it the *access structure*, which allows immediate access to just the delayed constraints affected as the result of a new input constraint.

There are two main elements in this chapter. First is a framework for the specification of *wakeup degrees* which indicate how far a hard constraint is from being awakened. Such a formalism makes explicit the various steps a CLP system takes in reducing a hard constraint into a directly solvable one. The second element is a runtime structure that involves a global pool of the delayed constraints, and an access structure for that pool. A preliminary discussion of this work appears in [89].

8.1 Delay Mechanisms

The idea of dataflow computation, see e.g. [9], is perhaps the simplest form of a delay mechanism since program operations can be seen as directional constraints with fixed inputs and outputs. In its pure form, a dataflow graph is a specification of the data dependencies required by such an operation before it can proceed. Extensions, such as the I-structures of [10], are used to provide a delay mechanism for lazy functions and complex data structures such as arrays in the context of dataflow.

Carlsson [22] describes an implementation technique for `freeze(X, G)`. While an implementation of *freeze* can be used to implement more sophisticated annotations like **wait** and **when**, this will be at the expense of efficiency. This is mainly because the annotations can involve complicated conjunctive and disjunctive conditions. Since *freeze* takes just one variable as an argument, it is used in a complicated manner in order to simulate the behavior of a more complex wakeup condition.

In general, the present implementation techniques used for Prolog systems have some common features:

- The delay mechanism relies on a modification of the unification algorithm [22]. This entails a minimal change to the underlying Prolog engine. In CLP systems, this approach is not directly applicable since there is, in general, no notion of unification.
- Goals are awakened by variable bindings. Each binding is easily detectable (during unification). In CLP systems, however, detecting when a delayed constraint should awaken is far more complicated in general. In this paper, this problem is addressed using wakeup degrees, described in the next subsection.
- The number of delayed goals is not large in general. Thus, implementations in which the cost of awakening a goal is related to the number of delayed goals [22], rather than the number of awakened goals, can be acceptable. In a CLP system, the number of delayed constraints can be very large, and so such a cost is unacceptable. However, it should be noted that both the number of delayed goals and the number of delayed constraints varies dramatically between programs.

8.2 Wakeup Systems

Presented here is a conceptual framework for the specification of operations for the delaying and awakening of constraints. We begin by formalizing the notions of delayed and directly solvable constraints.

A hard constraint will be defined here as a primitive relation symbol, together with a mapping from its arguments to variables in the constraint solver. So, for example, the CLP($\mathcal{R}$) constraint

```
X + Y + Z * V = 3
```

would, internally, have to be broken up into the constraints

```
X + Y + T1 = 3
mult(T1, Z, V)
```

where all of the arguments of *mult* are variables.

Formally, each hard/delayed constraint δ is a tuple $\langle \Psi, \theta \rangle$ where Ψ is an n-ary hard constraint symbol and θ is a substitution of variables potentially in Σ for the argument variables A_1 to A_n of this instance of Ψ. Hence, in the example above, the second constraint is the informal representation of the formal hard constraint

$$\langle mult, \{A_1 \leftarrow T1, A_2 \leftarrow Z, A_3 \leftarrow V\} \rangle.$$

The state of the constraint solver at any given time be denoted by a tuple $\langle \Delta, \Sigma \rangle$, where Δ is the set of delayed constraints (meta-variable δ), and Σ is the set of directly solvable (and hence solved) constraints (meta-variable σ), otherwise known as a solved form.

To indicate how far a hard constraint is from being awakened, we associate with each constraint symbol Ψ a *wakeup system*. Intuitively, a wakeup system corresponds to a finite state automaton, where the final state corresponds to a hard constraint having become directly solvable, and the other states, all of them initial, indicate how far a delayed constraint is from being directly solvable. All of these states are known as *wakeup degrees*. We will denote these using the possibly subscripted meta-variable $\mathcal{D}$, and the degree corresponding to directly solvable constraints is distinguished with the name *awakened*. Formally, for a given Ψ, the set

$$\mathcal{DS}_\Psi = \{\mathcal{D}_0, \cdots, \mathcal{D}_n, awakened\}$$

is a set of wakeup degrees for Ψ. For given Σ, each degree represents a subset of Δ such that all of $\mathcal{DS}_\Psi$ represents a partition of Δ. We will say $\delta \mapsto_\Sigma \mathcal{D}$ to mean that delayed constraint δ is mapped to degree $\mathcal{D}$ with respect to Σ. A delayed constraint in *awakened* is ready to be removed from Δ, and processed by the solver.

The point is that we want to execute the various finite state automata for each of the delayed constraints, in order to manage their awakening (and, on backtracking, resuspension) efficiently. Taking the analogy further, we need to describe the transition conditions. These are really the main point of the exercise, because they are intended to be relatively inexpensive tests to determine when a constraint is coming closer to being awakened.

Associated with each wakeup degree $\mathcal{D}$ is a collection of pairs $\langle \mathcal{W}, \mathcal{N} \rangle$, where each $\mathcal{W}$ is a *generic wakeup condition* and each $\mathcal{N}$ is a wakeup degree called the *new degree*. Each generic wakeup condition is a condition, expressed in some formal language, on the argument variables of any delayed constraint assigned to the degree $\mathcal{D}$. Such a condition is satisfied by a solved form Σ for a particular delayed constraint $\delta = \langle \Psi, \theta \rangle$ if $\Sigma \models \mathcal{W}\theta$. Here $\models$ may be the entailment relationship used for constraint solving, or it may be a different one especially for awakening constraints. Note that $\mathcal{W}\theta$ will frequently be referred to as a *dynamic wakeup condition*. In terms of the automaton analogy, δ makes a transition from degree $\mathcal{D}$ to degree $\mathcal{N}$ under these circumstances.

We will need to impose some structural conditions on wakeup systems so that they make sense, but first we will give an example. To facilitate this and other examples in this thesis, we need to develop a language as mention above, for describing the function $\mapsto_\Sigma$ that partitions a Δ for a particular Σ, and also the generic wakeup conditions. We emphasize that this choice of language is arbitrary, and the appropriate choice may vary depending on the domain.

Let the *meta-constants* be a new class of symbols, and hereafter, these symbols are denoted by α, β and γ. A meta-constant is used as a template for a (regular) constant. Define that a *meta-constraint* is just like a constraint except that meta-constants may be written in place of constants. A meta-constraint is used as a template for a (regular) constraint.

To describe the partition function for a particular symbol Ψ, we simply assign to each wakeup degree other than *awakened* a description of the hard constraints placed in that degree for a given Σ. Each such description is a pair $(t, \mathcal{C})$ where

- t is a term of the form $\Psi(t_1, \ldots, t_n)$ where each t_i is either a variable, constant or meta-constant,
- $\mathcal{C}$ is a conjunction of meta-constraints whose variables and meta-constants, if any, appear in t. We allow the additional meta-constraint $nonground(X)$ which asserts that the variable X is not ground.

In our language, generic wakeup conditions have the same form as the meta-constraint component of a degree description.

For the entailment relationship, let us use CLP($\mathcal{R}$) constraint satisfiability. Then $\mapsto_\Sigma$ assigns a delayed constraint $\langle \Psi, \theta \rangle$ to the degree $\mathcal{D}$ if the description of $\mathcal{D}$ is $(\Psi(t_1, \ldots, t_n), \mathcal{C})$, and $\tilde{\exists}[\Sigma \models^* (A_1 = t_1 \wedge \cdots \wedge A_n = t_n \wedge \mathcal{C})\theta]$. Here the existential closure is intended to deal with the meta-constants as if they were variables. Hence, an equation between a variable and a meta-constant is satisfied if the variable is ground with respect to Σ, and then the meta-constant stands for this ground value.

Of course, we require that these descriptions form a partition of the possible instances of a Ψ constraint under any Σ. In CLP($\mathcal{R}$) for example, the wakeup system for the hard constraint symbol *pow*, where $pow(X, Y, Z)$ denotes $X = Y^Z$, might include a degree assigned the description

$$pow(A, B, \alpha), \quad \alpha \neq 0 \wedge nonground(A) \wedge nonground(B).$$

An instance of a *pow* constraint would be assigned to the corresponding degree if the first two arguments were not uniquely defined by Σ and the third was determined as a non-zero real number.

Consider once again the CLP($\mathcal{R}$) constraints involving *pow*. These constraints may be partitioned into classes represented by the following wakeup degrees:

$$\begin{array}{lcll} \mathcal{D}_0 & : & pow(A, B, C), & nonground(A) \wedge nonground(B) \wedge nonground(C) \\ \mathcal{D}_1 & : & pow(\alpha, B, C), & nonground(B) \wedge nonground(C) \\ \mathcal{D}_2 & : & pow(A, \alpha, C), & nonground(A) \wedge nonground(C) \\ \mathcal{D}_3 & : & pow(A, B, \alpha), & nonground(A) \wedge nonground(B) \end{array}$$

For the degree $\mathcal{D}_0$, an example wakeup condition is $C = \alpha$. This indicates that when a constraint, e.g. $Z = 4$, is entailed by the constraint store, a delayed constraint such as $pow(X, Y, Z)$ is reduced to $pow(X, Y, 4)$. This reduced constraint would be assigned the new degree $\mathcal{D}_3$. Another example wakeup condition is $A = 1$, indicating that when a constraint such as $X = 1$ is entailed, a delayed constraint of the form $pow(X, Y, Z)$ can be reduced to $pow(1, Y, Z)$. This reduced constraint, which is in fact equivalent to the constraint $Y = 1 \vee (Y \neq 0 \wedge Z = 0)$, would be assigned the degree *awakened* if the solver could deal with disjunction (which in CLP($\mathcal{R}$) it cannot). We note that stipulating that a variable is nonground in this way is going to be rather cumbersome. Hence we will leave the stipulation out, and assume it to apply whenever nothing else is said about a variable in a description. We exemplify some other uses of wakeup conditions below.

8.3 Structural Requirements of a Wakeup System

For the remainder of the discussion we need the concept of a path from one wakeup degree to another, and that of such a path being satisfied by a solved form.

Definition 1 (Finite Path) *A finite path from degree $\mathcal{D}$ to degree $\mathcal{D}'$ is a sequence of degrees $\mathcal{D}_j$ and connecting generic wakeup conditions $\mathcal{W}_{\mathcal{D}_j}$ associated with them:*

$$\mathcal{D}_1, \mathcal{W}_{\mathcal{D}_1}, \mathcal{D}_2, \cdots, \mathcal{D}_{n-1}, \mathcal{W}_{\mathcal{D}_{n-1}}, \mathcal{D}_n$$

for some $n \geq 1$. If $n = 1$ then this is a null path. We write $\Sigma \mid\!\models_\delta p$ for $\delta = \langle \Psi, \theta \rangle$ and a finite path p of length n if either

1. *$n = 1$, or*
2. *for all $j : 1 \leq j < n$, $\Sigma \models \mathcal{W}_{\mathcal{D}_j}\theta$;*

and for all $\langle \mathcal{W}, \mathcal{N} \rangle$ associated with $\mathcal{D}_n$, $\Sigma \cup \{\sigma\} \not\models \mathcal{W}\theta$

Informally, we say that the path p is *enabled* by Σ for δ. We will need to disallow the possibility of infinite paths being enabled, so we define them formally also.

Definition 2 (Infinite Path) *A infinite path from degree $\mathcal{D}$ is a sequence of degrees $\mathcal{D}_j$ and connecting generic wakeup conditions $\mathcal{W}_{\mathcal{D}_j}$ associated with them:*

$$\mathcal{D}_1, \mathcal{W}_{\mathcal{D}_1}, \mathcal{D}_2, \mathcal{W}_{\mathcal{D}_2}, \cdots$$

We write $\Sigma \mid\!\models_\delta p$ for $\delta = \langle \Psi, \theta \rangle$ and an infinite path p if for all $j : 1 \leq j$, $\Sigma \models \mathcal{W}_{\mathcal{D}_j}\theta$.

Now we are ready to state the structural conditions on wakeup systems. First, we require that every transition of every delayed constraint toward being awakened can be represented somewhere in the wakeup degree — essentially a soundness property.

Definition 3 (Soundness) *A wakeup system for Ψ is sound if for all Δ, Σ, $\delta = \langle \Psi, \theta \rangle \in \Delta$, and σ whenever $\delta \mapsto_\Sigma \mathcal{D}$ and $\delta \mapsto_{\Sigma \cup \{\sigma\}} \mathcal{D}'$ we have that all of the following hold:*

1. *There exists a path p from $\mathcal{D}$ to $\mathcal{D}'$ such that $\Sigma \cup \{\sigma\} \mid\models_\delta p$;*
2. *For all $\mathcal{D}''$ different from $\mathcal{D}$ such that there exists a path p' from $\mathcal{D}$ to $\mathcal{D}''$ and $\Sigma \cup \{\sigma\} \mid\models_\delta p'$, we have $\mathcal{D}' = \mathcal{D}''$;*
3. *There are no infinite paths q starting at $\mathcal{D}$ such that $\Sigma \cup \{\sigma\} \mid\models_\delta q$.*

The exact formulation of this condition needs some explanation. What we desire is that the for any hard constraint δ and the addition of any constraint to Σ, the degree assignment function $\mapsto_\Sigma$ and the wakeup system agree on how this addition affects the status of δ. This is captured directly in the first major condition. The complication is that it is not possible to define any suitably small granularity of constraints added to Σ such that there will only be a single corresponding transition in the wakeup system for the individual hard constraint. This will *usually* be the case, but there will be cases where a single, very simple constraint added to the solved form triggers a chain or, as we have defined it above, a *path* of transitions. these paths will, in general, not be unique, since it is necessary to allow for the transitions occurring in any order, but of course they must all end up in the same degree. This is the reason for the rather complicated second major condition. As an example of where the complications arise, consider the sequence of constraints

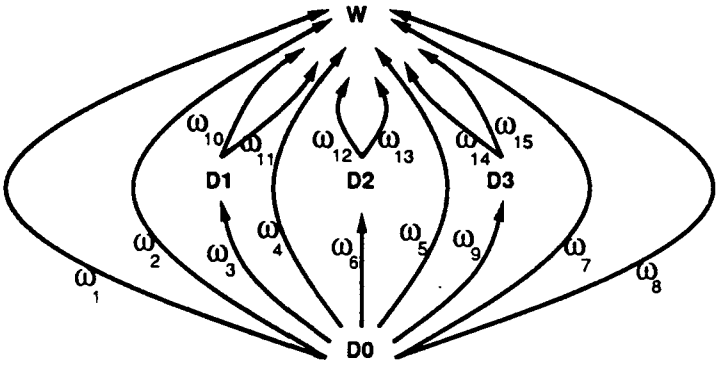
```
pow(X,Y,Z), Y = Z, Z = 5
```

where the final constraint causes the *pow* constraint to be awakened straight from being in the "least awakened" degree. Finally, as we will see in Chapter 11, it will sometimes be necessary to allow cycles in wakeup systems (that is, non-null paths that begin and end at the same degree). However, the transition enabled by any solver state for each delayed constraint should be finite, so we must not allow the possibility of an infinite path being enabled at the starting degree. Hence, we rule it ou' in the third major condition.

Our second structural requirement is that all of the wakeup degrees and dynamic wakeup conditions have some potential use, in a way that is locally sensible. We call this minimality since we want to exclude useless additional components.

Definition 4 (Minimality) *A wakeup system for Ψ is minimal if for all $\mathcal{D}$ and all $\langle \mathcal{W}, \mathcal{N} \rangle$ associated with $\mathcal{D}$ there exist Σ, σ and $\delta = \langle \Psi, \theta \rangle$ such that all of the following hold:*

1. $\delta \mapsto_\Sigma \mathcal{D}$
2. $\delta \mapsto_{\Sigma \cup \{\sigma\}} \mathcal{N}$
3. $\Sigma \cup \{\sigma\} \models \mathcal{W}\theta$



Legend:

D0 : $pow(A, B, C)$ D1 : $pow(\alpha, B, C), \alpha \neq 0 \wedge \alpha \neq 1$
D2 : $pow(A, \alpha, C), \alpha \neq 0 \wedge \alpha \neq 1$ D3 : $pow(A, B, \alpha), \alpha \neq 0 \wedge \alpha \neq 1$

$\omega_1 : A = 0$	$\omega_2 : A = 1$	$\omega_3 : A = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$\omega_4 : B = 0$	$\omega_5 : B = 1$	$\omega_6 : B = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$\omega_7 : C = 0$	$\omega_8 : C = 1$	$\omega_9 : C = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$\omega_{10} : B = \alpha$	$\omega_{11} : C = \alpha$	$\omega_{12} : A = \alpha$
$\omega_{13} : C = \alpha$	$\omega_{14} : A = \alpha$	$\omega_{15} : B = \alpha$

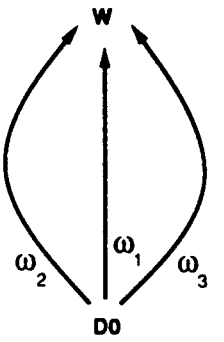
Figure 8.1: Wakeup degrees for $pow/3$

8.4 Example Wakeup Systems

The illustration in Figure 8.1 contains an example wakeup system for the CLP($\mathcal{R}$) constraint symbol *pow*. A wakeup degree is represented by a node, a wakeup condition is represented by an edge label, and the new degree of a reduced constraint is represented by the target node of the edge labeled with the wakeup condition that caused the reduction. Similarly, multiplication in CLP($\mathcal{R}$), denoted by the symbol *mult*, is shown in Figure 8.2.

Generic wakeup conditions can be used to specify the operation of many existing systems which delay constraints. In Prolog-like systems whose constraints are over terms, awaiting the instantiation of a variable X to a ground term can be represented by the wakeup condition $X = \alpha$. Awaiting the instantiation of X to a term of the form $f(\ldots)$, on the other hand, can be represented by $\exists Y.X = f(Y)$. We now give some examples on arithmetic constraints. In Prolog-III, for example, the wakeup condition $X \leq \alpha$ could specify that the length of word must be bounded from above before further processing of the word equation at hand. For CHIP, where combinatorial problems are the primary application, an example wakeup condition could be $\alpha \leq X \wedge X \leq \beta \wedge \beta - \alpha \leq 4$ which requires that X be bounded within a small range. For CLP($\mathcal{R}$), an example wakeup condition could be $X = \alpha * Y + \beta$, which requires that a linear relationship hold between X and Y.

As another example, consider equational constraints involving two dimensional arrays. Let $\mathit{array}(A, X, Y, E)$ denote the equation $A[X, Y] = E$, which might sensibly be delayed until all of A, X and Y are ground, at which time it becomes a simple



Legend: $\begin{cases} D0 : mult(A, B, C) \\ \omega_1 : A = \alpha \\ \omega_2 : B = \beta \qquad \omega_3 : C = \gamma \end{cases}$

Figure 8.2: Wakeup degrees for $mult/3$

equational constraint. Then a suitable wakeup system is shown in Figure 8.3.

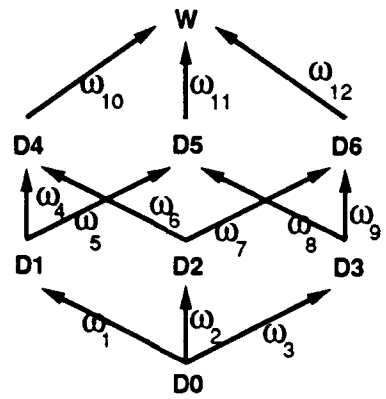
It might be expected that the wakeup system for word equations would be interesting. However, assuming a sufficiently powerful decision algorithm for the arithmetic constraints associated with word equations, we obtain only the very simple system in Figure 8.4, where $concat(A, B, C)$ denotes the equation $A.B = C$. Initially, one might expect that there should be a separate degree for the situation where the length of B is known, as this makes it easier to determine the length of A. However, if the decision procedure for the associated arithmetic constraints is sufficiently powerful, the length of A will always be supplied by that algorithm as soon as possible.

In CLP(Σ^*) [187], constraints of the form $x \in e$, where x is a string and e is a regular expression on strings, are delayed until they have a finite number of solutions. In general, this is the case when x is ground or e is ground and has no closure operators. To express this in our framework, we need to break these constraints into the three forms $x \in y^*$, $x \in y.z$ and $x \in y + z$, where x, y, and z must be syntactically either variables or strings. Denoting these constraint by *clos*, *conc* and *or* respectively, the wakeup system is shown in Figure 8.5.

8.5 The Runtime Structure

Here we present an implementational framework in the context of a given wakeup system. There are three major operations with hard constraints, which correspond to the actions of delaying, awakening and backtracking:

1. adding a hard constraint to the collection of delayed constraints;
2. awakening delayed constraints as the result of supplying a directly solvable constraint, and



Legend:

D0 : $array(A, B, C, D)$

D1 : $array(\alpha, B, C, D)$ D2 : $array(A, \beta, C, D)$

D3 : $array(A, B, \gamma, D)$ D4 : $array(\alpha, \beta, C, D)$

D5 : $array(\alpha, B, \gamma, D)$ D6 : $array(A, \beta, \gamma, D)$

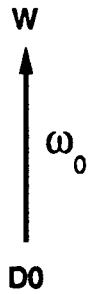
$\omega_1 : A = \alpha$ $\omega_2 : B = \beta$ $\omega_3 : C = \gamma$

$\omega_4 : B = \beta$ $\omega_5 : C = \gamma$ $\omega_6 : A = \alpha$

$\omega_7 : C = \gamma$ $\omega_8 : A = \alpha$ $\omega_9 : B = \beta$

$\omega_{10} : C = \gamma$ $\omega_{11} : B = \beta$ $\omega_{12} : A = \alpha$

Figure 8.3: Wakeup degrees for $array/4$

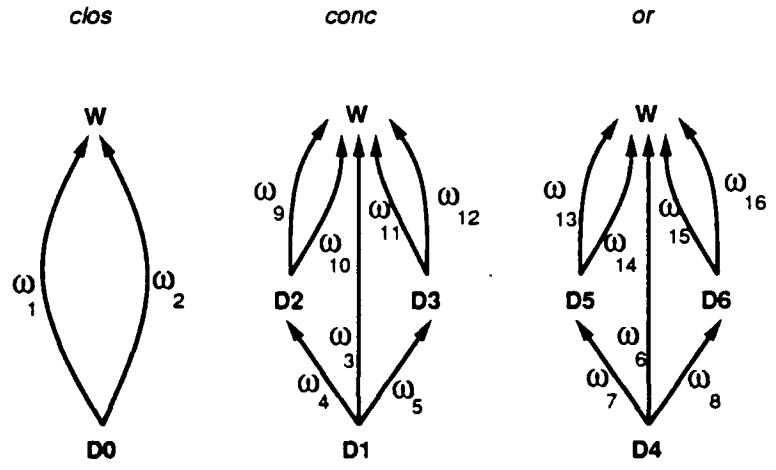


Legend:

D0 : $concat(A, B, C)$

$\omega_0 : |A| = \alpha$

Figure 8.4: Wakeup degrees for $conc/3$



Legend:

D0 : $clos(A, B)$
D1 : $conc(A, B, C)$ D2 : $conc(A, \beta, C)$ D3 : $conc(A, B, \gamma)$
D4 : $or(A, B, C)$ D5 : $or(A, \beta, C)$ D6 : $or(A, B, \gamma)$

$\omega_1 : A = \alpha$ $\omega_2 : B = \epsilon$
$\omega_3 : A = \alpha$ $\omega_4 : B = \beta$ $\omega_5 : C = \gamma$
$\omega_6 : A = \alpha$ $\omega_7 : C = \beta$ $\omega_8 : A = \gamma$
$\omega_9 : A = \alpha$ $\omega_{10} : C = \gamma$
$\omega_{11} : A = \alpha$ $\omega_{12} : B = \beta$
$\omega_{13} : A = \alpha$ $\omega_{14} : C = \gamma$
$\omega_{15} : A = \alpha$ $\omega_{16} : C = \beta$

Figure 8.5: Wakeup degrees for *clos*/2, *conc*/3 and *or*/3 in CLP(Σ^*)

3. restoring the entire runtime structure to a previous state, that is, restoring the collection of delayed constraints to some earlier collection, and restoring all auxiliary structures accordingly.

We deal with the third operation, as well as other aspects of incrementality and backtracking, in Chapter 9. The first of our two major structures is, for the time being, a pool of delayed constraints. Thus implementing the first operation, delaying a hard constraint, simply requires adding a record to the pool, with the current form of the constraint, and a field indicating its degree in the appropriate wakeup system.

Now consider the second operation. In order to implement this efficiently, it is necessary to have some access structure mapping an entailed constraint γ to just those delayed constraints affected by γ. Since there are in general an infinite number of entailed constraints, a finite classification of them is required. The classification we use is the set $\Omega_{\Delta,\Sigma}$ of all dynamic wakeup conditions for all the delayed constraints in Δ for the current Σ, noting that delayed constraints may have dynamic wakeup conditions in common. That is.

$$\Omega_{\Delta,\Sigma} = \{\mathcal{W}_{\mathcal{D}}\theta \quad | \quad \delta = \langle \Psi, \theta \rangle \in \Delta \ \wedge \ \delta \rightarrowtail_{\Sigma} \mathcal{D}\}.$$

Of course this set will change every time Δ or Σ change.

We now specify an access structure that maps a dynamic wakeup condition into a doubly linked list of nodes. Each node contains a pointer to a delay pool element containing a delayed constraint[1]. Corresponding to each occurrence node is a reverse pointer from the delay pool element to the occurrence node. For a generic wakeup condition $\mathcal{W}$ we refer to the dynamic wakeup condition $\mathcal{W}\theta$ as $\mathcal{DW}$. Call the list associated with $\mathcal{DW} \in \Omega_{\Delta,\Sigma}$ a *$\mathcal{DW}$-list*, and call each node in the list a *$\mathcal{DW}$-occurrence node*.

Initially the access structure is empty. The following specifies what is done for the basic operations.

8.5.1 Delaying a new hard constraint

This operation adds new hard constraint $\delta = \langle \Psi, \theta \rangle$ to Δ, such that $\delta \rightarrowtail_{\Sigma} \mathcal{D}$. First add a new delay pool element including its wakeup degree, $\mathcal{D}$. Let $\mathcal{W}^1_{\mathcal{D}}, \cdots, \mathcal{W}^n_{\mathcal{D}}$ be the generic wakeup conditions of $\mathcal{D}$. Then:

- Augment $\Omega_{\Delta,\Sigma}$ with all of the $\mathcal{DW}^1_{\mathcal{D}}, \cdots, \mathcal{DW}^n_{\mathcal{D}}$.
- For each $\mathcal{DW}^i_{\mathcal{D}}$, insert into the $\mathcal{DW}^i_{\mathcal{D}}$-list of the access structure a new occurrence node pointing to the delay pool element for δ.
- Set up reverse pointers from the δ node to the new occurrence nodes.

[1]The total number of occurrence nodes is generally larger than the number of delayed constraints.

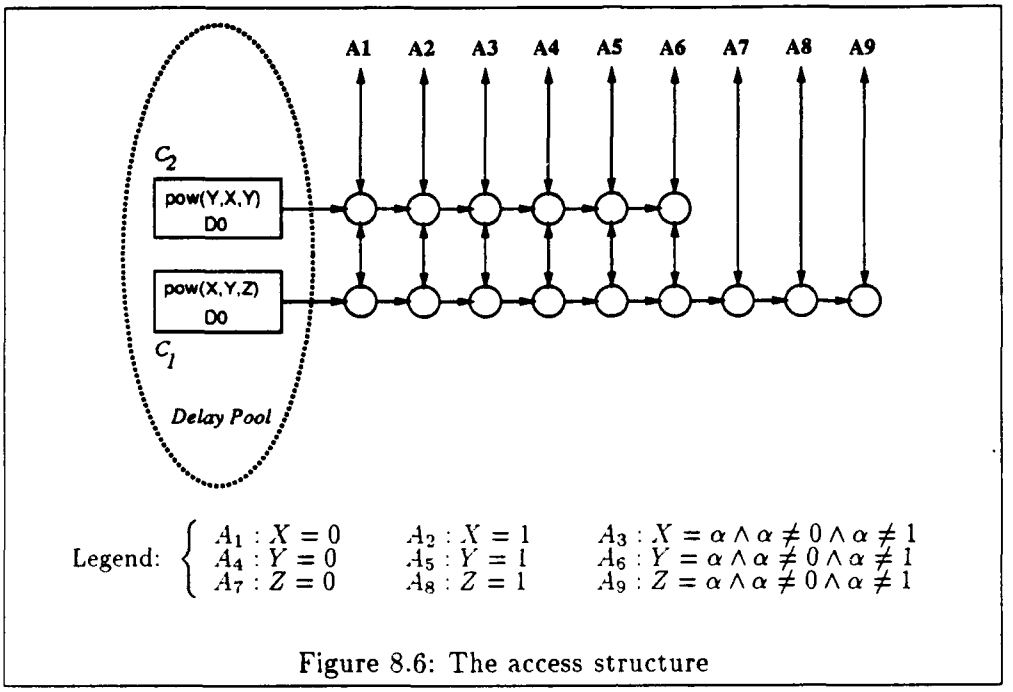


Figure 8.6: The access structure

8.5.2 Responding to Changes in Σ

When $\Sigma \models \mathcal{DW}$ for some $\mathcal{DW} \in \Omega_{\Delta,\Sigma}$, and this can happen whenever Σ is augmented, we need to update our structures.

Consider the $\mathcal{DW}$-list L. Then consider in turn each delayed constraint pointed to by the occurrence nodes in L. For each such δ node, perform the following.

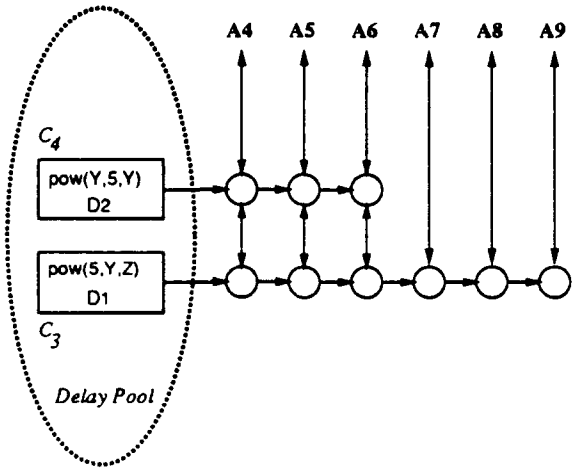
- Delete all occurrence nodes pointed to by the δ node., and delete $\mathcal{DW}$ from $\Omega_{\Delta,\Sigma}$.
- Update the node with the new wakeup degree of δ using the function $\mapsto_\Sigma$. Then perform the modifications to the access structure as described above when a new delayed constraint is added.

Figure 8.6 illustrates the entire runtime structure after storing the two hard constraints $pow(X, Y, Z)$ and $pow(Y, X, Y)$ in that order. Note that we are representing the hard constraints informally. Figure 8.7 illustrates the structure after a new input constraint adds $X = 5$ to the solved form.

8.5.3 Optimizations

An important optimization is to merge $\mathcal{DW}$-lists. Let there be lists corresponding to the dynamic wakeup conditions $\mathcal{DW}_1, \ldots, \mathcal{DW}_n$. These lists can be merged into one list with the condition $\mathcal{DW}$ if

- adding any delayed constraint δ results in either (a) no change in any of the n lists, or (b) every list has a new occurrence node pointing to δ;



Legend: $\begin{cases} A_4 : Y = 0 & A_5 : Y = 1 & A_6 : Y = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1 \\ A_7 : Z = 0 & A_8 : Z = 1 & A_9 : Z = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1 \end{cases}$

Figure 8.7: The new access structure

- for any Σ we have $\Sigma \models \mathcal{DW}$ iff $\Sigma \models \mathcal{DW}_i$ for some $1 \leq i \leq n$.

In the example of Figure 8.6, the three lists involving X can be merged into one list which is associated with the dynamic wakeup conditions $X = \alpha$. Similarly for Y and Z.

This optimization is very frequently applicable, and of great importance. Recall that the dynamic wakeup conditions must be monitored constantly, so it is essential that they be as simple as possible. It seems likely that delay with timely awakening will be prohibitively expensive in systems where this optimization cannot be successfully applied.

8.5.4 Summary of the Access Structure

The access structure maps a finite number of dynamic wakeup constraints to lists of delayed constraints. The constraint solver is assumed to identify those conditions for which an entailed constraint is an instance. The cost of the basic operations can be summarized as follows.

1. Adding a new constraint δ simply involves determining its wakeup degree, creating a delay pool entry, creating new occurrence nodes corresponding to δ and the setting of pointers between the new delay pool entries and occurrence nodes. The cost here is bounded by the number of generic wakeup conditions associated with (the degree of) δ.

2. Changing the degree of a constraint δ involves replacing δ with δ', determining the degree of δ', and deleting and inserting a number of occurrence nodes. Since the occurrence nodes are doubly-linked, each such insertion and deletion can be done in constant time. Therefore the total cost here is bounded by the number of generic wakeup conditions associated with δ and δ'.

3. The cost of backtracking is important, and will be discussed together with the scheme for implementing it, in Chapter 9.

In short, the cost of one primitive operation on delayed constraints (delaying a new hard constraint, upgrading the degree of one delayed constraint, including awakening the constraint, and undoing the delay/upgrade of one hard constraint) is bounded by the (fixed) size of the underlying wakeup system. The total cost of an operation (delaying a new hard constraint, processing an entailed constraint, backtracking) on delayed constraints is proportional to the size of the delayed constraints affected by the operation.

Chapter 9

Incremental Constraint Solvers

A major challenge in implementing CLP systems stems from the required utility of the constraint solvers. To be somewhat precise, a constraint solver maintains constraints with respect to three kinds of operations for adding, removing and solving constraints as follows:

- A state $\mathcal{S} = \langle \mathcal{S}_1, \cdots, \mathcal{S}_n \rangle$ where, for each i, $\mathcal{S}_i = \{c_1, \cdots, c_{m_i}\}$. The c_j are constraints, and $\mathcal{S}_n$ represents the set of constraints added since the most recent choice point.
- A function *cp* that maps $\langle \mathcal{S}_1, \cdots, \mathcal{S}_n \rangle$ to $\langle \mathcal{S}_1, \cdots, \mathcal{S}_n, \{\} \rangle$.
- A function *bt* that maps $\langle \mathcal{S}_1, \cdots, \mathcal{S}_n \rangle$ to $\langle \mathcal{S}_1, \cdots, \mathcal{S}_{n-1} \rangle$.
- A function *solve* that maps a state $\mathcal{S} = \langle \mathcal{S}_1, \cdots, \mathcal{S}_n \rangle$ and a constraint c to
  - state $\langle \mathcal{S}_1, \cdots, \mathcal{S}_n \cup \{c\} \rangle$ and answer *true* if $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n \cup \{c\}$ is satisfiable
  - the old state $\mathcal{S}$ and answer *false* otherwise.

The original state is $\mathcal{S} = \langle \{\} \rangle$. The function *cp* establishes a choice point – a state to which the computation will backtrack on a subsequent failure. The function *bt* actually effects such backtracking – rolling back to the state before the most recent choice point. The function *solve*, of course, tries to augment the solver state with a new constraint, checking that it is consistent with the constraints already there.

In this chapter we discuss the practical implications of this required utility, and a strategy for dealing with them. We discuss Prolog from this viewpoint, and then consider in detail the techniques for making the CLP($\mathcal{R}$) constraint solvers incremental. Much of this material is presented from a different viewpoint in [89] and [88].

9.1 Practical Implications of Incrementality

We should note that, in principle, any constraint solving algorithm can be used without modification to provide this functionality. It can simply be re-invoked on the constraints

in $\mathcal{S}$ every time *solve* is invoked, and the states can be stored "outside" this solver. However, the following methodological and empirical observations about the expected nature of program execution show that such a naive approach, no matter how sophisticated the actual solving algorithm, is unlikely to be practical. These observations stem from Part II of this thesis.

In a given computation:

1. The number of constraints handled by *solve* can be extremely large (say tens of thousands or more).
2. The number of constraints in the solver state $\mathcal{S}$ can also be extremely large, but typically, because of backtracking, it will not contain all of the constraints handled.
3. The number n of sets $\mathcal{S}_i$ of constraints between choice points can vary substantially, because some programs are much more deterministic than others.
4. For the same reason, the number of choice points can vary substantially.
5. While the set of constraints collected in the solver state may be very complicated, a large proportion of the individual constraints handled may be very simple.

Now, returning to the possibility of re-invoking a standard constraint algorithm for every use of *solve*. Assuming (optimistically) that this algorithm takes time proportional to the number of constraints, the cost of using *solve* n times will be of order n^2. This is unacceptable for a programming language.

The objectives of the design of a constraint solver are then to find an appropriate decision algorithm and representation of state such that:

1. Individual invocations of *solve* should be affected more by the size and complexity of the constraint added, rather than of the constraints already stored in the state. We will call this the *incrementality criterion.*
2. The overhead of such invocations should not be excessive.
3. The space needed is not substantially more than linear in the size and number of constraints stored, with low overhead.
4. The *cp* function is implemented with very low overhead (virtually none) in space and time.
5. The cost of the *bt* function is dependent more on the amount the store was changed in adding the constraints now being be removed than on the size of the entire store. This is just another facet of the incrementality criterion.

The phrase *solved form* is typically used to describe such a representation of the constraint store. In general, the incrementality criterion is a rather vague notion. In its strictest form, it is virtually unattainable. Clearly, since the consistency of constraints

with the store is being tested, the cost will be somewhat affected by the number and complexity of the *related* constraints. That is, if all the variables in a new constraint are new to the store, the size of the store should be irrelevant. On the other hand, if many constraints on the same variables are already being stored, that is an unreasonable requirement. The notion of incrementality for some algorithms and representations may be made precise, but in general it will be an empirical property.

9.2 Incrementality and Prolog

Before moving on to CLP in general, let us consider to what extent unification in Prolog is incremental. It it assumed that the reader is somewhat familiar with the standard Prolog implementation techniques, especially the WAM.

We begin with the *solve* operation. Variable bindings in a Prolog interpreter essentially constitute an efficient solved form. When a new equation is encountered, the bindings of only some of the variables will have to be looked up, and likewise for the variables in the terms they are bound to, and so on. However, if free variables are considered not to be in the solved form, the solved form is unchanged when a new equation is dealt with: it is only augmented. For example, consider the equation

```
f(X, g(V)) = f(h(Y, Z), U)
```

in the context of a solved form including the bindings

```
U = g(W)
W = g(g(g(Y)))
Y = g(g(Z))
```

but without any bindings for variables `X` or `V`. Since `X` is new, it is simply bound to `h(Y, Z)` and the bindings of `Y` and `Z` do not have to be considered at all — because the occurs check is omitted. The binding of `U` must be looked up to ensure that the principal functor can be a `g`, as it already is. Since `V` is new, it can simply be bound to `W`, whose binding again need not be considered. In summary, the cost of such a unification is only related to the size of the solved form in so far as parts of the solved form are truly relevant to the unification. Nevertheless, unification is exponential in the worst case.

Now we consider the operations involving backtracking. Recall that choice point records are inserted into the runtime stack at nondeterministic branches, and some bindings are trailed[1] so they can be undone on backtracking. However, backtracking is reasonably cheap, since:

- Bindings between activation records point from the newer variable to the older. This is useful if the binding is made before the first choice point after the creation of the new activation record. The binding doesn't need to be trailed, since

[1]The trail is a stack containing choice point records interleaved by a sequence of pointers to changed variable locations that must be restored on backtracking.

the entire activation record will be removed on backtracking, thus removing the bindings.

- Bindings that are created pointing from above the most recent choice point record are never trailed, for the same reason.
- The *cp* operation involves only creating a choice point record, and affects the cost of subsequent bindings somewhat.
- The *bt* operation is relatively fast, since the trail simply lists locations where bindings have to be deleted. This is because the solved form is never changed beyond being further instantiated.

As we will see below, many of these fortunate properties are lost when other kinds of constraints have to be solved. Nevertheless, it is important to understand the impact of these properties on the performance of Prolog systems.

9.3 Strategy for Incremental Solvers

In general, to satisfy the above criteria, constraint solvers must typically rely on a solved form, with the implementation having the following properties:

1. When a new constraint is handled by *solve*, the relevant parts of the solved form must be found quickly. That is, the solved form is essentially indexed on variables. Then the index is used to look up the parts relevant to the variables in the new constraint. Ideally the index is an extension of Prolog variable bindings.
2. The satisfiability check usually involves attempting to construct a new solved form, augmented with the new constraint. This should affect as little of the solved form and supporting structures as possible. However, in the worst case, the entire solved form may have to be examined, and may also have to be extensively modified.
3. Since the solved form is, in general, modified rather than just further instantiated, simple trailing is insufficient. A tagged value trail is needed. On backtracking, the tag is used to determine what the inverse operation is, the usual address is used to determine where it is to be performed, and the value field, which may involve a complex data structure, is used to perform the inverse operation.
4. In general, parts of the solved form will be changed many times between choice points. Hence, it is often best not to store all of the changes made for the purposes of backtracking. It may be better to backup an appropriate chunk of the solved form when making the first change after a choice point. Then, subsequent changes will not require trailing until another choice point is encountered. This works well when appropriately small chunks can be identified, so they are not too expensive to save.

5. Certain cross-referencing structures are too large and complex, and are changed too often to be worth trailing. It is better to rebuild these on backtracking. This has the advantage that no cost is incurred unless unless backtracking actually takes place.

For an interesting discussion of incrementality in constraint solving, see [55]. There is an important problem that we have not discussed so far: to date little work on decision algorithms has taken incrementality into account. This means that for some domains with a well known, efficient decision algorithm, at may be the case that either:

- there is no known incremental algorithm,
- the incremental algorithms are extremely slow, or
- there is a good incremental algorithm, but it is completely different to the best decision algorithm overall.

This factor limits the kinds of CLP languages that can be made practical, and hence needs to be considered by the language designer.

9.4 Incrementality in Four Modules of CLP($\mathcal{R}$)

We now return to our major case study, and discuss the aspects of the implementations of the various solver modules in the two CLP($\mathcal{R}$) implementations that are related to incrementality.

9.4.1 Unification

Unification in CLP($\mathcal{R}$) is sufficiently close to Prolog unification that it inherits the incrementality discussed above. However, even here the trailing (for backtracking) becomes somewhat more involved, since it is no longer the case that bindings must simply be undone. In particular, variables with implicit bindings (see Subsection 7.3.1) may have to go back to being of type s_var.

9.4.2 Linear Equations

The algorithm for solving linear equations in CLP($\mathcal{R}$) was presented in Subsection 7.3.3. The parametric solved form is represented mainly by a tableau of linked-lists. We also need a *cross-reference* table mapping each parametric variable to the list of its occurrences within the main tableau. Additional structures include a tagged trail stack for backing up parametric forms that are about to be changed for the first time since the last choice point. We now briefly outline the critical operations in the equality solver given this method of representation.

Parametric Substitutions

Given that a parametric variable T is to be replaced in the tableau, the first operation is to obtain, from the list in the cross-reference table corresponding to T, the list of occurrences of T in the tableau. The actual substitutions will have a cost proportional to the length of this list.

An important consideration is how much this substitution enlarges the tableau. A straightforward heuristic for minimizing this number is to choose T such that its frequency in the tableau is minimal. At the present stage of our implementation, we have not found this to be necessary, and this suggests the frequencies tend to be uniformly distributed over the variables T we choose.

Backtracking

In CLP($\mathcal{R}$), as foreshadowed above, substitutions actually change the form of a parametric equation. Thus when each row is changed for the first time after a choice point, the previous form, or some representation of it, must be stored. Given the data structures above, it is clear that not only do the parametric forms themselves have to be saved for backtracking, but also the support structures, such as the cross-reference table, must be saved. Thus some complicated administrative mechanisms are also required, two of which need to be discussed.

First consider the parametric forms themselves. If we assume that the parametric form of a variable is, in general, going to be short, then it is reasonable to copy the entire equation when it is changed after a choice point. The extra space used is not a problem since the number of parameters is small, and this method is very simple and hence fast. The alternative is to trail each change in an equation. While this "fine-grained" method usually benefits space utilization, it can suffer both from added complexity and reduced efficiency because many changes can go by between one choice point and the next. Next consider the cross-reference table. As with parametric forms, we could save entire entries corresponding to variables that are just about to be changed in some way. Instead, we choose to *reconstruct* the appropriate entries whenever parametric forms are restored upon backtracking. Since equations are typically small, reconstructing the cross-reference table from the restored equation and the current equation is cheap and furthermore, this method does not incur any overhead when backtracking does not occur.

Example

Figure 9.1 shows the major solver structures after the constraints `X = Y + Z` and `W = 2 * Z + 1` have been solved, with variables `Y` and `Z` chosen as parametric, and the others as non-parametric. Figure 9.1 shows the state after a choice point has been encountered and the additional constraint `W = 5` has been solved. Notice how the old parametric forms have been saved but their index structure has not. In the trail fragments shown, `P` denotes that the variable was newly bound to a parametric form, and `C` denotes that a parametric form was changed. In the stack fragments, `S` denotes that the variable is

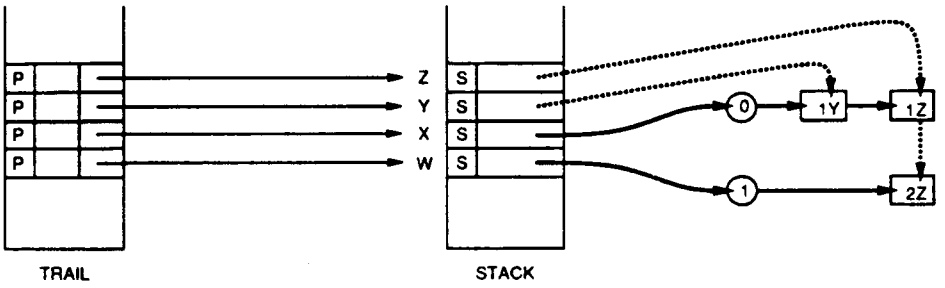


Figure 9.1: Solver state after two equations

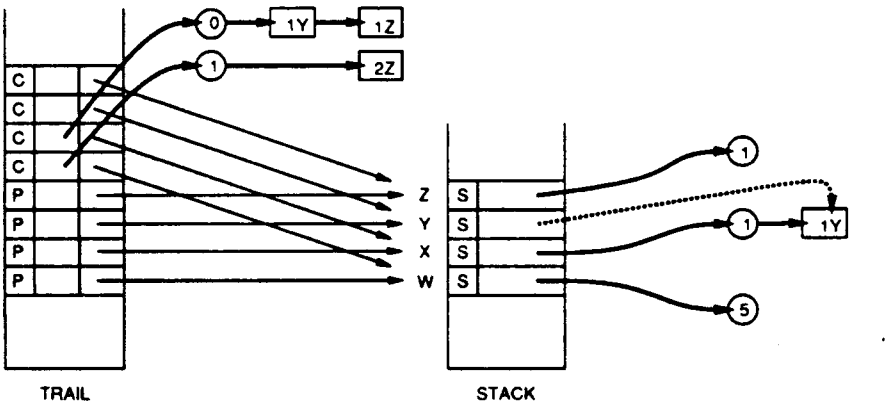


Figure 9.2: Solver state after choice point and third equation

a solver variable, bound to a parametric form. We omit the details of distinguishing between parametric and non-parametric variables, and of implicit bindings.

9.4.3 Linear Inequalities

The details of the linear inequality solver are beyond the scope of this thesis — see [86] and [174]. It is based on an incremental formulation of the first phase of the two-phase simplex method. The constraints that are actually sent to this solver are described in Chapter 7. In the worst case, of course, the algorithm can be exponential in the size and number of these constraints. However, each new constraint tends to take, on average, less than one pivot to be incorporated. The internal data structures of this solver are very similar to those of the equality solver.

9.4.4 The Delay Pool

The main issue is maintaining the delay pool and access structure in the presence of backtracking. For example, if changes to the structure were trailed using some adaptation of Prolog techniques, then a cost proportional to the number of entries can be incurred even though no delayed constraints are affected. We return to the exponentiation example in Chapter 8 as our running example.

The delay pool itself is implemented using a stack. Thus delaying a hard constraint simply requires a push on this stack. Additionally, the stack contains hard constraints that are reduced forms of constraints deeper in the stack. For example, if the hard constraint $pow(X, Y, Z)$ were in the stack, and if the input constraint $Y = 3$ were encountered, then the new hard constraint $pow(X, 3, Z)$ would be pushed, together with a pointer from the latter constraint to the former. In general, the collection of delayed constraints contained in the system is described by the sub-collection of stacked constraints that have no inbound pointers.

When a new hard constraint is being delayed, entering the appropriate record into the delay pool consists of just pushing it onto the delay stack. When a new constraint is entailed, and some delayed constraint is affected, placing the form C' of C in the delay pool changes somewhat. C' is pushed onto the stack, setting up a pointer from C' to C.

Figure 9.3 illustrates the entire runtime structure after the two hard constraints $pow(X, Y, Z)$ and $pow(Y, X, Y)$ were stored, in this order, and then a new input constraint makes $X = 5$ entailed. Note that this one equation caused the pushing of two more elements, these being the reduced forms of the original two. The top two constraints now represent the current collection of delayed constraints. This is equivalent to the diagram in Figure 8.7.

Restoring the stack during backtracking is easy because it only requires a series of pops. For the access structure, no trailing/saving of entries is performed; instead, they are *reconstructed* upon backtracking. Such reconstruction requires a significant amount of interconnection between the global stack and access structure. In this runtime structure, the overhead cost of managing an operation on the delayed constraints is proportional to the size of the delayed constraints *affected* by the operation, as opposed

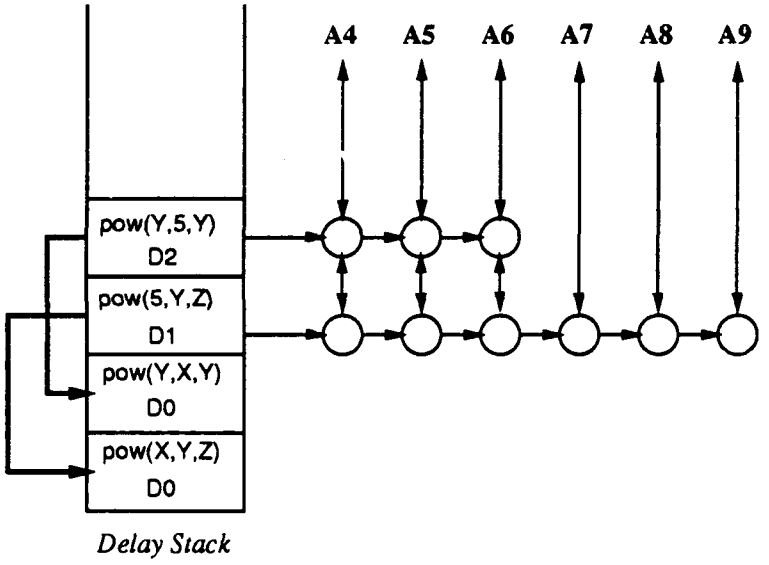


Figure 9.3: Delay stack and access structure

to all the delayed constraints. In more detail, the primitive operation of backtracking is the following:

- Pop the stack, and let C denote the constraint just popped.
- Delete all occurrence nodes pointed to by C.
- If there is no pointer from C (and so it was a hard constraint that was newly delayed) to another constraint deeper in the stack, then nothing more need be done.
- If there is a pointer from C to another constraint C' (and so C is the reduced form of C'), then perform the modifications to the access structure *as though* C' were being pushed onto the stack. These modifications, described above, involve computing the dynamic wakeup conditions pertinent to C', inserting occurrence nodes, and setting up reverse pointers.

Note that the access structure obtained in backtracking may not be structurally the same as that of the previous state. What is important, however, is that it depicts the same *logical* structure as that of the previous state.

Additional efficiency can be obtained by not creating a new stack element for a reduced constraint if there is no choice point (backtrack point) between the changed degrees in question. This saves space, saves pops, and makes updates to the access structure more efficient.

Another optimization is to save the sublist of occurrence nodes deleted as a result of changing the degree of a constraint. Upon backtracking, such sublists can be inserted

into the access structure in constant time. This optimization, however, sacrifices space for time.

Finally, let us reconsider the complexity. The cost in backtracking of popping a node C, which may be the reduced form of another constraint C', involves deleting and inserting a number of occurrence nodes. The cost here is bounded by the number of generic wakeup conditions associated with C and C'. In short, the cost of one primitive operation on delayed constraints (delaying a new hard constraint, upgrading the degree of one delayed constraint, including awakening the constraint, and undoing the delay/upgrade of one hard constraint) is bounded by the (fixed) size of the underlying wakeup system. The total cost of an operation (delaying a new hard constraint, processing an entailed constraint, backtracking) on delayed constraints is proportional to the size of the delayed constraints affected by the operation.

9.5 Summary

It is essential that the designer and implementor of a CLP system understand the consequences of the incrementality requirement imposed by the operational model. This chapter emphasizes those consequences and gives some guidance as to how they may be dealt with. However, no recipe can be provided for dealing with this problem, and in many cases it cannot be solved. Dealing with incrementality is likely to remain a major engineering challenge in realizing practical CLP systems.

Chapter 10

Compilation

When considered at a very high level, the challenges that arise in compiling CLP languages correspond closely to those that arise in compiling Prolog. These can be summarized as:

1. Suitably mapping the operational model to the linear sequential structures of von Neumann machines.

2. Improving the efficiency of constraint solving by partially evaluating the constraint solving algorithm with respect to the partially constructed constraints appearing in the static program.

The extra challenges posed by CLP in the first respect have already been discussed at some length in earlier chapters of this thesis. The second challenge is more problematic, as partially evaluating a constraint solver with respect to individual constraints is not always of great benefit. It is useful for unification because of the highly syntactic nature, but for domains where the constraints are much less syntactic in nature, this is usually not the case. Certainly some overheads of interpretation will be removed, and some peephole optimizations will be made possible, and this is very useful. However, in many cases, constraint solving can only be made substantially more efficient if special cases are detected. Global analysis of programs to obtain type and mode information is the major technique for obtaining such information.

In this chapter we will illustrate these concepts by describing in detail some techniques for compiling CLP($\mathcal{R}$). In this particular instance our objective will be to develop an abstract machine architecture for CLP($\mathcal{R}$) that allows us to:

1. Demonstrate that for the Prolog subset of CLP($\mathcal{R}$) it is possible to attain performance that is close to that of state-of-the-art Prolog compilers.

2. Remove basic interpretation overheads from general arithmetic constraints.

3. Take advantage of present and future work on global analysis for CLP($\mathcal{R}$) programs, with multiple specialization of procedures. (That is, determine the calling patterns with which a procedure is to be used, and generate multiple specialized code sequences for different patterns where appropriate.)

Preliminary discussions of various aspects of this work appear in [91] and [87].

Abstract machines have been used for implementing programming languages for many reasons. Retargetability is one: only an implementation of the abstract machine needs to be made available on each platform. Another is simply convenience: it is easier to write a native code compiler if the task is first reduced to compiling for an abstract machine that is semantically closer to the source language. The best abstract machines sit at just the right point on the spectrum between the conceptual clarity of the high-level source language and the details of the target machine. In doing so they can often be used to express programs in exactly the right form for tackling the efficiency issues of a source language. For example, the Warren Abstract Machine (WAM) [189] revolutionized the execution of Prolog, since translating programs to the WAM exposed many opportunities for optimization that were not apparent at the source level. An example from further afield is the technique of optimizing functional programs by first converting them to Continuation Passing Style (CPS) [99]. CPS conversion has lead to highly optimizing compilers for Scheme and Standard ML. The benefit from designing an appropriate abstract machine for a given source language can be so great that even executing the abstract instruction code by interpretation can lead to surprisingly efficient implementations of a language. For example, many commercial Prolog systems compile to WAM-like code. Certainly more efficiency can be obtained from native code compilation, but the step that made Prolog usable was that of compiling to the WAM.

Because we wish to attain Prolog performance for the Prolog subset of CLP($\mathcal{R}$), it seems reasonable that the abstract machine should be based on the WAM. Other implementation models have been proposed (and even used) for Prolog recently, but these tend to be concerned more with aspects of the final realization than just the compilation model. We will concentrate on those aspects that support the compilation model.

After reviewing some aspects of Prolog compilation and the Warren Abstract Machine (WAM), we describe the design of the core of the Constraint Logic Arithmetic Machine (CLAM). The Core CLAM has been used as the basis of a widely distributed compiler-based CLP($\mathcal{R}$) system, in which CLAM code is interpreted by an emulator written in C. As will be demonstrated in Section 10.7, this system achieves almost the performance of the best commercial Prolog systems, and the speed of constraint-solving is improved significantly. Finally, some of the interesting global properties of CLP($\mathcal{R}$) programs that can be detected by global analysis (itself beyond the scope of this thesis) are introduced, and the CLAM is extended to take advantage of these. The performance improvement is estimated by coding the specialized CLAM programs by hand.

10.1 Prolog Compilation and the WAM

As was mentioned in Chapter 2, the Warren Abstract Machine [189] (WAM) revolutionized the compilation of Prolog. We review some of the basic ideas here, but readers

not familiar with the WAM should consult the tutorial by Aït-Kaci [2]. The WAM is particularly suitable for software emulation of compiled code. The essential idea behind it is that the instructions can be used to represent variants of the unification operation – specialized by partially evaluating unification with respect to the terms in the program. Furthermore, it is particularly successful in mapping the depth-first search of Prolog with a left-right atom selection rule to the conventional von-Neumann model of program execution.

The major data structures of the WAM are as follows:

- *stack*
  For storing activation records, consisting of variable bindings and return information.

- *heap*
  For storing complex structures and those variable bindings that need to be kept after their activation record has been trimmed through tail recursion.

- *trail*
  For keeping pointers to variables that need to be unbound on backtracking. Also contains *choice point records*, which keep track of which rule is to be used next on backtracking.

- *registers*
  For arguments of procedure calls both at call and return, and also for temporary values.

The basic types of instructions are:

- *partial construction and unification of terms*
  These are used to construct a simple or complex term, or get it from a memory location, and put it in a register. They also unify such terms with the contents of a register.

- *control*
  These include procedure call, choice points and backtracking.

- *indexing*
  These typically check whether the first argument of a call is instantiated enough to use table lookup rather than backtracking to find the appropriate rule.

Commercial implementations variously interpret WAM code using an emulator written either in a high level language or machine code, or compile to native code. Most of those using an emulator use many additional instructions obtained from combining pairs or groups if instructions commonly found in sequence, in order to reduce the fetch and decode overhead.

Considerable work has been done on extending the WAM for different kinds of unification or different control strategies (see [22], for example).

10.2 The Core CLAM

Presented here is a set of basic instructions that, in addition to those of the WAM, are sufficient to execute CLP($\mathcal{R}$) programs in general. In order to understand the design of the instructions, it is necessary to recall some key aspects of the constraint solver. Mainly, we will describe basic instructions that organize constraints for the solver. Some specialized versions of these instructions are also described. We then discuss the necessary runtime support.

10.2.1 Instructions for Arithmetic Constraints

Recall that a *linear parametric form* is $c_0 + c_1V_1 + \cdots + c_nV_n$, where $n \geq 1$, each c_i is a real number and each V_i is a distinct *solver variable*. A linear equation, say $V = c_0 + c_1W_1 + \cdots + c_nW_n$, equates a solver variable and a parametric form. The variables W_i are known as *parameters* and V is a non-parameter. Linear inequalities are stored in the form $lpf \geq 0$ or $lpf > 0$ where lpf is a linear parametric form. The solver ensures that every solver variable appearing in an equation either is a parameter, or appears as a non-parameter in at most one equation. It also ensures that every variable in the inequalities is parametric if it appears in an equation.

The basic instructions of the core CLAM build and manipulate parametric forms:

- `initpf`[1] `c0`
  initializes a parametric form with constant `c0`.
- `addpf_va{lr}`[2] `ci, Vi`
  adds the term `ci * Vi` to the parametric form[3]. The two versions of the instruction (`var` or `val`) correspond to whether V_i is a local variable appearing for the first time. If so, we need to create storage for the variable, and we may also simplify constraint solving. The variable V_i may be a parameter, in which case c_i is added to the coefficient of V_i in the parametric form, or already have a parametric representation lpf, in which case $c_i * lpf$ is added.
- `solve_eq0`
  signifies the end of the construction of the linear form. The equation $c_0 + c_1V_1 + \cdots + c_nV_n = 0$ is solved by the equation solver.
- `solve_ge0` and `solve_gt0`
  Similar to the above, but forming an inequality or strict inequality instead of an equation.

For example, consider the constraint $5 + X - Y = 0$, where X is a new variable. It could be compiled as shown below. We also indicate how the instructions are executed in the case where Y is non-parametric, say $Y = Z + 3.14$ is in the solver.

[1] The "pf" stands for parametric form.

[2] This brace notation indicates addpf_val or addpf_var

[3] V_i is either a register or a local variable in the stack; the distinction is not important for this paper.

`initpf`	`5`	*lpf*:	5
`addpf_var`	`1, X`	*lpf*:	$5 + 1 * X$
`addpf_val`	`-1, Y`	*lpf*:	$1.86 + 1 * X - 1 * Z$
`solve_eq0`		*solve*:	$1.86 + 1 * X - 1 * Z = 0$

Recall that a constructed parametric form contains only parametric and new variables. Also recall that the process of solving an equation built using such a parametric form roughly amounts to:

(a) finding a parameter V in the form to become non-parametric,

(b) writing the equation into the normalized form $V = lpf$,

(c) substituting out V using *lpf* in all other constraints, and finally

(d) adding the new constraint $V = lpf$ to the solver.

Suppose there is a new variable in the equation to be compiled. Then it will always appear in the parametric form at runtime, so it can be chosen in step (a). Since this choice is made at compile time, much of the work of step (b) can also be compiled away. Step (c) is not needed since the variable is new. Hence all we need is a new instruction for step (d), one for which the solver always returns *true*. A similar simplification can be made for the compilation of inequalities that contain a new variable. Three new instructions are needed:

- `solve_no_fail_eq V`
- `solve_no_fail_ge V`
- `solve_no_fail_gt V`

For example, the above constraint $5 + X - Y = 0$, where X is new, can be better compiled into the instructions

`initpf`	`-5`	*lpf*:	-5
`addpf_val`	`1, Y`	*lpf*:	$-1.86 + 1 * Z$
`solve_no_fail_eq`	`X`	*add*:	$X = -1.86 + 1 * Z$

Finally, there are a number of simple enhancements we can make to the instruction set to facilitate commonly occurring cases. The cases where the constant is zero or the coefficient is 1 or -1 occur in the majority of instances, so special instructions can be expected both to reduce the code size and cut down on decode time. This can be done using

- `initpf_0` start a new linear form with constant 0;
- `addpf_va{lr}_{+-} Vi` add a term consisting of a variable with coefficient ± 1.

Nonlinear constraints can be separated out at compile time, as discussed in Section 10.3, so that they appear in one of a few particular forms: $X = Y \times Z$, $X = pow(Y, Z)$,

$X = abs(Y)$, $X = sin(Y)$, $X = cos(Y)$. Since CLP($\mathcal{R}$) delays the satisfiability of nonlinear constraints until they become linear, the role of these instructions is to check whether the constraints are runnable, and accordingly either delay them or invoke the appropriate solver. Instructions are provided for creating a nonlinear constraint in any one of its degrees, so for example there are five instructions for *pow* corresponding to the various wakeup degrees:

- `pow_vvv Vi, Vj, Vk` for $V_i = pow(V_j, V_k)$ where V_i, V_j, V_k are variables.
- `pow_cvv Vj, Vk` for $c = pow(V_j, V_k)$
- `pow_vcv Vi, c, Vk` for $V_i = pow(c, V_k)$
- `pow_vvc Vi, Vj, c` for $V_i = pow(V_j, c)$
- `pow_cvc c0, Vj, c2` for $c_0 = pow(V_j, c_2)$

(In fact, an extra degree of freedom results from allowing `V` instead of `v` so that a variable can be initialized.) For example $X = pow(3, Y)$ compiles to `pow_vcv X, 3, Y`. The remaining forms of a *pow* constraint, e.g. $8 = pow(2, X)$, are equivalent to linear equations and hence the exponentiation is evaluated at compile time, and replaced with a linear equation, e.g. $X = 3$. There is also a class of variants of these instructions, such as `pow_Vcv`, which allow a variable (the one indicated by `V`) to be initialized. The other nonlinears are handled similarly.

Finally, note that numbers in complex terms can be treated much like constants, with the `putnum`, `getnum`, `uninum` and `bldnum` instructions.

10.2.2 Runtime Issues

The CLAM requires the same basic runtime support as the WAM, with the addition of the arithmetic constraint solver and its data structures, extended unification and additional backtracking support.

Data Structures

Some data structures needed to support the CLAM are a routine extension of those for the WAM — the usual register, stack, heap and trail organization. The main new structures pertain to the solver. Variables involved in arithmetic constraints have a *solver identifier*, which is used to refer to that variable's location in the solver data structures.

The modifications to the basic WAM architecture are:

- *Solver identifiers*
  Arithmetic variables need to be represented differently from Prolog variables. In addition to the usual WAM cell data types, one more is required. Cells of this type contain a solver identifier. The basic unification algorithm needs to be augmented to deal with this new type.

- *Tagged trail*
  In the WAM, the trail merely consists of a stack of addresses to be reset on

backtracking. For CLP($\mathcal{R}$), the trail is also used to store changes to constraints. Hence a tagged value trail is required. The tags specify what operation is to be reversed, and the value component, if present, contains any old data to be restored.

- *Choice points*
  Choice points are expanded slightly so as to save the "high water mark" for solver identifiers and inequalities. That is, the tableau entries beyond which all entries should be deleted on backtracking.

- *Linear form accumulator*
  A linear constraint is built up using one instruction for the constant term, and one for each linear component. During this process, the partially constructed constraint is represented in an accumulator. One of the `solve` instructions then passes the constraint to the solver. We can think of this linear form accumulator as a generalization of the accumulator in classical computer architectures, accumulating a linear form instead of a number.

Unification

Unification in the CLAM is again similar to that in the WAM, except that solver identifiers are included, as was shown previously in Figure 7.4. However, the arithmetic expressions and arithmetic constants of that table do not appear here, as they are separated out using the special arithmetic instructions. Thus, the CLAM unification table shown in Figure 10.1 is a simplified version of the earlier table.

Backtracking

As mentioned above, backtracking is somewhat more complicated because many different operations need to be trailed, and backtracking may require operations other than mere unbinding. The cases are:

- simple (un)bindings
- restoring a linear equation or inequality to empty
- restoring a linear equation or inequality to a previous (non-empty) form, as stored in the trail
- rolling back a delayed nonlinear constraint that has come closer to being awakened, or has been awakened

CLP($\mathcal{R}$) Unification Table			
T2	T1		
	var	s_var	func
var	*bind*		
s_var	$T1 \rightarrow T2$	solver	
func	$T1 \rightarrow T2$	n/a	$T1 = T2$?

Legend:

bind	– Bind T1 to T2, or bind T2 to T1
T1 → T2	– Bind T1 to T2
var	– ordinary free variable
s_var	– solver variable
func	– an uninterpreted functor
$T1 = T2$?	– are the topmost symbols equal?
n/a	– not applicable (corresponds to a type error)

Figure 10.1: Unification Table for CLAM Execution

10.3 Basic Code Generation

The basic code generation issues for CLP($\mathcal{R}$) can be divided into three groups:

1. Constraint level — implicit and explicit
2. Rule level
3. Procedure level (indexing)

Here we concentrate on the first and second groups. In terms of code generation, we have to deal with constraints in two forms: implicit half-constraints in the argument of a predicate, and explicit constraints in the body of a rule. A half-constraint is an arithmetic term that is going to become half of an arithmetic equation at run time. The compilation of an individual CLP($\mathcal{R}$) rule proceeds in three stages:

1. separation of implicit arithmetic half-constraints as explicit constraints,
2. rewriting and reordering of explicit constraints for optimization purposes, and
3. transforming explicit constraints to a canonical form for direct code generation.

The difference between implicit and explicit constraints is actually very minor. Implicit half-constraints are treated as much as possible like terms in arguments of predicates in Prolog programs. In CLP($\mathcal{R}$), the interesting part in both cases is dealing

with the arithmetic subterms (if any) of a term. If the entire term is an arithmetic term, we simply have an arithmetic constraint, which can be treated as described below. Arithmetic constants, in terms of code generation, are handled exactly like constants when in the context of an uninterpreted functor, using an analogous set of instructions. The first stage transformation is as follows. Where a complex arithmetic term appears as an argument of a functor, the approach is to replace it with a new variable, and add an explicit equation to the rule between the new variable and the arithmetic term. If the term was in the head, the obvious place for the new constraints is just after the head. If in a body atom, the obvious place is just before the atom. There may be reasons for putting them elsewhere, but these will be discussed below. It should be noted that for code generation purposes, a variable appearing immediately under an uninterpreted functor is treated as a general variable. In many cases its involvement in arithmetic constraints elsewhere in the rule may suggest treating it differently, and this is a useful optimization, but it is not essential, because of the way unification works.

As an example of how arithmetic subterms are removed in the first phase transformation, the rule

```
p(X, Y + Z, f(4), g(Y - Z)) :- q(h(X + Z)), r(Y - X).
```

would be transformed to

```
p(X, T1, f(4), g(T2)) :-
          T1 = Y + Z,
          T2 = Y - Z,
          T3 = X + Z,
          q(h(T3)),
          T4 = Y - X,
          r(T4).
```

The aim of the second stage is to examine all the explicit arithmetic constraints in the rule together, along with information about intervening subgoals, to replace them with a more efficient set of constraints.

If we ignore subgoals, there are a number of basic transformations that can be useful:

1. *separation of common subexpressions*
   This is very similar to the issue in imperative languages, but in CLP($\mathcal{R}$) we potentially have more freedom, since the subexpression can be dealt with before, between or after its original occurrences. For example, the constraint sequence

   ```
   X + Y + Z = 3, Y + Z + 7 > 0
   ```

   can be transformed to any of

   ```
   X + T = 3, T = Y + Z, T + 7 > 0
   T = Y + Z, X + T = 3, T + 7 > 0
   X + T = 3, T + 7 > 0, T = Y + Z .
   ```

As is the case in imperative languages, of course, common subexpressions can be quite well hidden. However, some of these will be taken care of by the transformations described below. For example,

```
X + Y + Z = 2 * Z + 5 + 2 * Y
```

is a linear equation, and in preparation for generating linear code in the third stage would be transformed by collecting variables to:

```
X - Y - Z - 5  = 0 .
```

Those that are hidden under nonlinearity or truly multiple constraints cannot, however, be dealt with so easily.

2. *re-ordering of constraints*
Certain orders of constraints result in inefficiency. For example, for the sequence

```
X = Y * Z, T + Z = 4, T - Z = 0
```

the most obvious code generation would result in the nonlinear constraint being delayed and then almost immediately awakened by the subsequent pair of constraints, which ground Z to 2 and hence make the first constraint linear. The order

```
T + Z = 4, T - Z = 0, X = Y * Z
```

would be much more favorable. Even if we do not wish to do the necessary analysis to realize that Z would be grounded, it is reasonable to put nonlinears as late as possible in a constraint sequence. This will be especially important below, where we actually generate simple nonlinears by pulling them out of more complex constraints: this tells us where to put them.

The situation can be more extreme:

```
X = Y * Z, Y = 2
```

should become not merely

```
Y = 2, X = Y * Z
```

but rather

```
X = 2 * Z, Y = 2
```

and of course

```
X = Y * Z, Y = 2, Z = 3
```

should be rewritten all the way to

```
X = 6, Y = 2, Z = 3
```

and in the last two transformations the simple equations can be discarded if the variable does not appear anywhere else.

3. *constraint fusion*
   Where certain variables are used to communicate between two constraints, and for no other purpose, it may be possible to combine the constraints. For example, in the sequence

```
X = Y + Z + W, Z > 0
```

   if the `Z` is not used elsewhere we just need

```
X - Y - W > 0
```

All of these transformations can become more complicated if there are subgoals between the constraints considered. This is because, as undesirable as it is, the user may have been relying on the specific level of instantiation or restriction of certain variables for control or side-effects. In fact, the first is virtually unavoidable in general, because of depth-first search. This means that a constraint should never be moved beyond a subgoal containing "related" variables, and this property (at least somewhat conservatively) is easy to check. Moving a constraint forward across a related subgoal can not cause problems if the subgoal uses rules that are free of side-effects and non-logical predicates, and this can be checked globally, as discussed below.

Now it remains to consider explicit arithmetic constraints. As we have seen above, the CLAM provides instructions for almost the most general form of linear equation or inequality, but only the most simple nonlinear equations. Of course, any CLP($\mathcal{R}$) arithmetic constraint can be transformed such that these instructions are sufficient (in fact, more than sufficient) but in general there are a number of ways to do this for any one constraint. The simplest way to generate code for a general arithmetic constraint is to break it at every internal node of its parse tree. This results in a large number of constraints of the form

$$\mathcal{P}(b_0, \mathcal{O}(b_1, \cdots, b_n))$$

where, for each i, b_i is either a variable or numeric constant, $\mathcal{P}$ is either equality or inequality, and $\mathcal{O}$ is an arithmetic operation. This would naturally suggest single instructions for each of the operations.

For example, the obvious parse tree for the inequality

```
X + Y + Z * W >= X + U - V + 5
```

suggests the inefficient breakup

```
T1 = X + Y
T2 = Z * W
T3 = T1 + T2
T4 = U - V
T5 = X + T4
T6 = 5
T7 = T5 + T6
T3 >= T7
```

However, the more general linear arithmetic instructions described above are intended to provide a more efficient vehicle for a class of constraints that occurs relatively frequently in programs. The intention is that linear subterms of arithmetic constraints should not be broken up, but treated together by a single chain of appropriate instructions. This potentially complicates the issue somewhat, because certain algebraic transformations on constraints may produce longer runs of linear components. For example, the constraint

```
(X + Y) * (U + V) = Z
```

could be broken up directly into the simple constraints

```
T1 = (X + Y)
T2 = (U + V)
T1 + T2 - Z = 0
```

or by rewriting it using the distributive law as

```
X * U + X * V + Y * U + Y * V = Z
```

it can be broken up into

```
T1 = X * U
T2 = X * V
T3 = Y * U
T4 = Y * V
T1 + T2 + T3 + T4 - Z = 0
```

Now the question is whether this is actually desirable. It tends to increase the number of constraints overall, increase the number of nonlinear constraints (potentially delayed) that are dependent on each variable, and produces a long linear equation rather than a number of short ones, but the components of the long one tend to be dependent on potentially delayed nonlinear constraints. In short, it seems to be of little value. However, the answer is less obvious when a linear term is multiplied by a constant. For example,

```
X + 3 * (Y + Z) = U
```

is probably best expanded to

```
X + 3 * Y + 3 * Z = U
```

as the disadvantage of having to use instructions with explicit coefficients (rather than the implicit 1) is outweighed by not having to deal with the apparent nonlinearity. However, for a constraint with more variables in the linear term, like

```
X + 3 * (Y1 + Y2 + Y3 + Y4 + Y5 + Y6) = U
```

it is no longer clear that it is better to multiply through in advance. It is important, however, to put the instructions for the apparently nonlinear term after the arguments become available. That is, the order

```
T1 - Y1 - Y2 - Y3 - Y4 - Y5 - Y6 = 0
T2 = 3 * T1
X + T2 - U = 0
```

will have little overhead, whereas

```
T3 = T1 * T2
X + T3 - U = 0
T2 - Y1 - Y2 - Y3 - Y4 - Y5 - Y6 = 0
T1 = 3
```

where none of Y1 to Y6 have values yet would be unacceptable, as a nonlinear constraint would first be delayed, and then awakened by the last constraint.

Because of the tradeoffs involved, a relatively simple scheme of flattening has been used to date: the distributive law is usually not applied, but commutativity and associativity are used extensively. As a more complicated example,

```
X * Y * Z >= 3 + cos(U + V) + U1 + V1
```

would be expanded to

```
T1 = X * Y
T2 = T1 * Z
T3 - U - V = 0
T4 = cos(T3)
T2 - T4 - U1 - V1 - 3 >= 0
```

and, returning to our earlier example,

```
X + Y + Z * W >= X + U - V + 5
```

should be transformed to

```
Y + T1 - U + V - 5 >= 0
T1 = Z * W
```

In summary, CLP($\mathcal{R}$) code generation is of course an extension of Prolog code generation. Beyond the usual issues carried over from Prolog, the remaining ones are largely generalizations of those arising from arithmetic expressions in conventional programming languages. Finally, special role of linearity in CLP($\mathcal{R}$) results in code being generated in ways that favor linear equations.

10.4 Global Optimization of CLP($\mathcal{R}$)

The kinds of program analysis required to utilize the specialized CLAM instructions include those familiar from Prolog — most prominently, detecting special cases of unification and deterministic predicates. Algorithms for such analysis have become familiar; see [40] for example. The extension to constraints involves a reasonably straightforward extension to these algorithms.

10.4.1 Modes and Types

One of the principle mechanisms for enhancing efficiency is to avoid invoking the fully general solver in cases where the constraints are simple. For example when they are equivalent to tests or assignments. Given some fixed calling pattern for a predicate, including information about which variables are ground or unconstrained at call time we can determine which constraints can be executed as tests or assignments.

Consider the following simple program, whose structure is typical of recursive definitions in CLP($\mathcal{R}$):

```
sum(0, 0).
sum(N, X) :-
      N >= 1,
      N' = N - 1,
      X' = X - N,
      sum(N', X').
```

and consider executing the query `sum(7, X)`. The first constraint encountered, $N \geq 1$, can be implemented simply as a test since the value of N is known. The second constraint, $N' = N - 1$, can be implemented simply as an evaluation and assignment to N' because N' is a new variable and the value of N is known. These observations continue to hold when executing the resulting subgoal `sum(6, X')` and for each subsequent subgoal. Hence for any query `sum(`c`, X)`, where c is a number, the constraint $N \geq 1$ is *always* a test, and $N' = N - 1$ is always an evaluation/assignment.

Even when the above properties do not hold, some simplifications can be made. For example, consider the constraint $V = I * R + V_0$ where the values of I and V_0 will be known when the constraint is encountered. Since I will be known, it is clear at compile time that the constraint will be linear and hence there is no need to separate out the multiplication. Furthermore, the constraint becomes a linear equation on two variables since V_0 is also known.

More information about obtaining mode and type information for CLP($\mathcal{R}$) programs can be found in [47, 90, 114]. In particular, Jørgensen [90] has automated the analyses.

10.4.2 Redundancy

An obvious way to enhance efficiency is to minimize the number of constraints in the solver. Toward this aim we want to eliminate redundant constraints, and an obvi-

ous starting point is to eliminate newly encountered constraints that are redundant. However, such constraints are rare. What is more common is that new constraints make some *earlier* collected constraints redundant. We concentrate on linear constraints in this subsection since detecting redundancy involving nonlinear constraints is intractable.

We now discuss two forms of redundancy: one associated with variables and the other with constraints.

Redundancy of Variables

We say that a variable is redundant at a certain point in the computation if it will never appear again. The elimination of these variables from the solver produces a more compact representation of the constraints on the variables of interest. Eliminating these variables not only saves space, but also reduces solver activity associated with these variables when adding new constraints. Every variable becomes redundant eventually; thus to gain the most compact representation we may project the constraints onto the remaining variables. (For example, if the only variables of interest are S and T, the constraints $T = T_1, T_1 = T_2 + T_3 - T_4, T_4 - T_3 = 1, T_2 = S + 2$ can be better represented by $T = S + 1$.) However, projection can be expensive, especially when inequalities are involved. It is only worthwhile in certain cases, one of which is identified below. We note that identifying redundant variables could be achieved at runtime as part of general garbage collection. However, greater benefit can be obtained by utilizing CLAM instructions to remove these variables in a timely fashion.

Consider the `sum` program above. After compiling away the simple constraints as described above, the following constraints, among others, arise during execution of the goal `sum(7, X)`:

$$\begin{array}{lrcl} (1) & X' & = & X - 7 \\ (2) & X'' & = & (X - 7) - 6 \\ (3) & X''' & = & ((X - 7) - 6) - 5 \\ \ldots & & \ldots & \end{array}$$

Upon encountering the second equation $X'' = X' - 6$ and simplifying into (2), note that the variable X' will never occur in future. Hence equation (1) can be deleted. Similarly upon encountering the third equation $X''' = X'' - 5$ and simplifying into (3), the variable X'' is redundant and so (2) can be removed. In short, only one equation involving X need be stored at any point in the computation.

Redundancy of Constraints

Of concern here are constraints that become redundant as a result of new constraints. One class of this redundancy that is easy to detect is *future redundancy* defined in [91], where a constraint added now will be made redundant in the future, but it does not effect execution in between these points.

Consider the **sum** program once again, and executing the goal `sum(N, X)` using the second rule. We obtain the subgoal

```
N >= 1, N' = N - 1, sum(N', X')
```

and note that we have omitted writing the constraint involving X. Continuing the execution we have two choices: choosing the first rule we obtain the new constraint $N' = 0$, and choosing the second rule we obtain the constraint $N' \geq 1$ (among others). In each case the constraint $N \geq 1$ is made redundant. The main point of this example is that the constraint $N \geq 1$ in the second rule should be implemented simply as a test[4], and not added to the constraint store.

Detecting redundant variables and future redundant constraints can in fact be done without dataflow analysis. One simple method involves unfolding the predicate definition (and typically once is enough), and then, in the case of detecting redundant variables, simply inspecting where variables occur last in the unfolded definitions. For detecting a future redundant constraint, the essential step is determining whether the constraints in an unfolded predicate definition imply the constraint being analyzed. Some further discussions appear in [91].

The experimental analyzer implemented by Jørgensen [90] indicates that many CLP($\mathcal{R}$) programs can be analyzed quite effectively.

10.5 The Extended CLAM

Above we described the core CLAM and associated runtime structures. Here we present some new instructions that implement some key optimization steps in arithmetic constraint solving. Specific kinds of global analysis are required in order to utilize these instructions.

To take advantage of inferred modes, a new data type, *fp_val*, is provided to represent arithmetic variables whose value is known to be ground. They can be thought of as "un-boxed" variables, except that some tag discipline is still needed in certain cases if heap garbage collection is to be used. (In CLP($\mathcal{R}$) fp_vals are in fact stored in adjacent pairs of registers, or in adjacent stack locations.)

[4] In general, these tests are not just simple evaluations.

The new instructions are:

• `litf`	`c, FPi`	load a numeric constant into an fp_val;
• `getf`	`Vi, FPj`	convert a solver variable to an fp_val;
• `putf`	`FPi, Vj`	convert an fp_val to a solver variable;
• `stf`	`FPi, S`	put an fp_val on the stack frame (offset S);
• `ldf`	`S, FPi`	read an fp_val from stack frame (offset S);
• `mvf`	`FPi, FPj`	copy one fp_val to another;
• `addf`	`FPi, FPj, FPk`	add fp_vals; similarly for `mulf`, `subf`, `divf`;
• `addcf`	`FPi, c, FPk`	add a constant to an fp_val; similarly for `mulcf`, `subcf`, `divcf`;
• `jeqf`	`FPi, L`	jump to label `L` if `FPi` is zero;
• `jgtf`	`FPi, L`	jump to label `L` if `FPi` is positive.
• `jgef`	`FPi, L`	jump to label `L` if `FPi` is nonnegative.

For example, when N is known in advanced to be a ground arithmetic variable, and N' is known to be a free arithmetic variable, the constraints $N \geq 1, N' = N - 1$ can be compiled into

```
        subcf    N, 1, Tmp
        jgef     Tmp, cont
        fail
cont:   mvf      Tmp, N'
```

We add new instructions for creating parametric forms using fp_val variables:

• `initpf_fp`	`FPi`	begin a parametric form obtaining constant from an fp_val;
• `addpf_fp_va{rl}`	`FPi, Vi`	add a linear component obtaining the coefficient from an fp_val.

The constraint $V = I * R + V_0$, when I and V_0 are known to be ground arithmetic variables, may then be compiled into

```
initpf_fp       V0
addpf_fp_val    I, R
addpf_val-      V
solve_eq0
```

We finally remark that the instructions for nonlinears are also augmented to make use of variables stored in fp_vals.

For redundant variables, we add the following instructions to the CLAM:

• `addpf_va{lr}e`	`ci, Vi`
• `addpf_va{lr}e_{+-}`	`Vi`
• `addpf_fp_va{lr}e`	`FPi, Vi`

As before, these augment the current parametric form with an entry involving V_i, but now they indicate that the variable V_i is redundant (i.e. will not occur again) and can be eliminated (hence the "e"). Returning to the **sum** example above, a possible sequence of instructions for the iterations is:

```
(1)  init_pf              -7
     addpf_val_+          X
     solve_no_fail_eq0    X'
(2)  init_pf              -6
     addpf_vale_+         X'
     solve_no_fail_eq0    X''
(3)  init_pf              -5
     addpf_vale_+         X''
     solve_no_fail_eq0    X'''
...  ...
```

Notice that a different set of instructions is required for the first equation from that required for the remaining equations. Hence the first iteration needs to be unrolled to produce the most efficient code.

Now let us briefly consider implementation of the instructions. Eliminating a variable X from the constraint solver is quite simple: if X is a non-parametric variable, then we just remove the linear form associated with X (as in the above example). If, however, X is a parameter, then to eliminate X we must (i) find an equation containing X, (ii) rewrite the equation with X as the subject, (iii) substitute out X everywhere else using this equation, and finally, (iv) remove the equation. Thus when X is a parameter, there is a trade-off between having one less equation and performing the work toward this aim (because this work is essentially equivalent to adding one new equation). For example, removing an equation is not worthwhile if execution immediately backtracks after its removal.

The following illustrates a typical execution sequence. Suppose the solver contained $X = Y + 1, T = U + Y - 1$ and a new constraint $Y + X - T = 0$ is encountered. Suppose further that it is known that Y does not appear henceforth and so can be eliminated. A straightforward implementation would (a) write the new constraint into parametric form $U - Y = 0$, (b) substitute out U everywhere by Y, (c) add the new constraint $U = Y$, and finally (d) using the information that Y is redundant, process the three resulting equations $X = Y + 1$, $T = 2 * Y - 1$, $U = Y$ in order to eliminate Y in the manner described above. A much better implementation will (a) write the new constraint into parametric form $U - Y = 0$, and (b) substitute out Y everywhere by U (instead of vice versa).

For redundant constraints, we add the new instructions

- **solve_no_add_eq0**
- **solve_no_add_ge0**
- **solve_no_add_gt0**

that behave like the **solve** class of instructions, but do not add the new constraint. In general this task involves significantly less work than the usual constraint satisfiability check and addition since we do not have to detect implicit equalities and may avoid substitutions.

We finally remark that, in our experiments, implementing future redundancy has lead to the most substantial efficiency gains compared to the other optimizations discussed here. The main reason is that inequalities are prone to redundancy, and the cost of maintaining inequalities in general is already relatively high. Equations in contrast are maintained in a form that is essentially free of this kind of redundancy.

10.6 Summary of Main CLAM Instructions

In Figure 10.2, the main CLAM instructions other than those directly inherited from the WAM are tabulated.

10.7 Empirics

Here we wish to demonstrate:

- the performance of the CLAM-based CLP($\mathcal{R}$) compiler for Prolog programs,
- the performance of the same compiler for programs involving arithmetic constraints, and
- the potential for improvement from a highly optimizing CLP($\mathcal{R}$) compiler, based on global analysis, generating extended CLAM code.

Throughout this section, all timings (in seconds) were obtained on an IBM RS 6000/530 workstation running AIX Version 3.0. The C compiler used throughout was the standard AIX C compiler with '−O' level optimization. The systems tested are

- The CLP($\mathcal{R}$) interpreter, written entirely in C, whose inference engine uses standard Prolog structure sharing techniques.
- The CLP($\mathcal{R}$) compiler system, executing core CLAM code. The CLAM code is interpreted, using an emulator written in C.
- An emulator, as above, for the full CLAM, executing handwritten code.
- Quintus Prolog version 3.0b, a widely-used commercial system.
- Two C programs, for comparison with a CLP($\mathcal{R}$) program compiled into CLAM code.

In Figure 10.3 we compare CLP($\mathcal{R}$) with Quintus Prolog. The programs chosen are a naive reverse benchmark (six times on a 150 element list, built using a normal functor rather than the list constructor), a program for converting boolean formulae into

Mnemonic	Arguments	Explanation
`initpf`	`c0`	initialize accumulator *lpf* with constant `c0`
`initpf_0`		initialize accumulator *lpf* with constant 0
`initpf_fp`	`FPi`	initialize *lpf* with constant from fp_val
`addpf_va{lr}`	`ci, Vi`	add `ci * Vi` to *lpf* in accumulator
`addpf_va{lr}_{+-}`	`Vi`	add linear component, with 1 or -1 coefficient
`addpf_fp_va{rl}`	`FPi, Vi`	add linear component, with coefficient from fp_val
`addpf_va{lr}e`	`ci, Vi`	like `addpf_va{lr}`, eliminating V_i if possible
`addpf_va{lr}e_{+-}`	`Vi`	like `addpf_va{lr}_{+-}` , eliminating V_i if possible
`addpf_fp_va{lr}e`	`FPi, Vi`	like `addpf_fp_va{lr}` , eliminating V_i if possible
`solve_eq0`		invoke equation solver on $lpf = 0$
`solve_ge0`		invoke inequality solver on $lpf \geq 0$
`solve_gt0`		invoke inequality solver on $lpf > 0$
`solve_no_fail_eq`	`V`	simply add $V = lpf$ to solver
`solve_no_fail_ge`	`V`	simply add $V \geq lpf$ to solver
`solve_no_fail_gt`	`V`	simply add $V > lpf$ to solver
`solve_no_add_eq0`		check $lpf = 0$, do not add to solver
`solve_no_add_ge0`		check $lpf \geq 0$, do not add to solver
`solve_no_add_gt0`		check $lpf > 0$, do not add to solver
`litf`	`c, FPi`	load numeric constant into fp_val
`getf`	`Vi, FPj`	convert solver variable to fp_val
`putf`	`FPi, Vj`	convert fp_val to solver variable
`stf`	`FPi, S`	put fp_val on stack frame (offset S)
`ldf`	`S, FPi`	read fp_val from stack frame (offset S)
`mvf`	`FPi, FPj`	copy one fp_val to another
`addf`	`FPi, FPj, FPk`	add fp_vals; similarly for `mulf`, `subf`, `divf`
`addcf`	`FPi, c, FPk`	add a constant to an fp_val; similarly for `mulcf`, `subcf`, `divcf`, `subfc`, `divfc`
`jeqf`	`FPi, L`	jump to label `L` if `FPi` is zero
`jgtf`	`FPi, L`	jump to label `L` if `FPi` is positive
`jgef`	`FPi, L`	jump to label `L` if `FPi` is nonnegative
`pow_vvv`	`Vi, Vj, Vk`	for $V_i = pow(V_j, V_k)$ here V_i, V_j, V_k are variables
`pow_cvv`	`Vj, Vk`	for $c = pow(V_j, V_k)$
`pow_vcv`	`Vi, c, Vk`	for $V_i = pow(c, V_k)$
`pow_vvc`	`Vi, Vj, c`	for $V_i = pow(V_j, c)$
`pow_cvc`	`c0, Vj, c2`	for $c_0 = pow(V_j, c_2)$ similar groups of instructions for *cos*, *sin*, *abs* and multiplication

Figure 10.2: Summary of Main CLAM Instructions

program	*Quintus 3.0b*	*CLP($\mathcal{R}$) interpreter*	*CLP($\mathcal{R}$) compiler*
nrev	0.66	1.91	0.79
dnf	0.61	1.89	0.78
zebra	1.33	2.58	1.57
8 queens	45.2	94.3	51.7

Figure 10.3: *Prolog benchmarks*

disjunctive normal form, a program that solves a standard logic puzzle by combinatorial search, and obtaining all solutions to the 8 queens problem. The programs in themselves are not interesting, and hence the code is not shown here. The important point is that they test major aspects of a Prolog inference engine. The purpose of this comparison is first to indicate the relative speeds of the inference engines of the two CLP($\mathcal{R}$) systems, and more importantly, to give evidence that the CLAM can be implemented without significantly compromising Prolog execution speed.

The main part of this section deals with the constraint aspects of the CLAM. We use the program in Figure 10.4 in four different ways, as shown in Figure 10.5, so as to utilize different constraint solving instructions

Given the calling pattern associated with the first query, the program can be compiled as though it were a simple recursive definition. Similarly for the second query, though a different recursive definition is obtained. Figures 10.6 and 10.7 contain C functions that implement these two definitions. The third query essentially propagates constraints, and hence it cannot be compiled so simply. Similarly for the fourth query, which essentially carries out a search.

In Figure 10.8, timings are tabulated for the interpreter, core CLAM and full CLAM. To separate the effects of the mode-based optimizations from those of the redundancy-based optimizations, Figure 10.8 also contains timings for full CLAM code that does not take advantage of redundancy. The first two timing columns illustrate the benefit of compilation over interpretation. For the next two columns, the program is specialized with respect to the modes corresponding to the four queries.

10.8 Extended Examples

In this section we return to the mortgage program to show, in detail, CLAM code for the general program as well as versions specialized for the various kinds of queries. We begin with the general program in Figure 10.13. Then we give high-level pseudocode for the four specializations (Figures 10.9, 10.10, 10.11 and 10.12) and the respective pieces of CLAM code (Figures 10.14, 10.15, 10.16 and 10.17).

```
mortgage(P, T, I, R, B) :-
     T > 1,
     T1 = T - 1,
     P >= 0,
     P1 = P * I - R,
     mortgage(P1, T1, I, R, B).
mortgage(P, T, I, R, B) :-
     T = 1,
     B = P * I - R.
```

Figure 10.4: *Program for reasoning about mortgage repayments*

```
Q1 : ?- mortgage(100000, 360, 1.01, 1025, B).
Q2 : ?- mortgage(P, 360, 1.01, 1025, 12625.9).
Q3 : ?- R > 0, B >= 0, mortgage(P, 360, 1.01, R, B).
Q4 : ?- 0 <= B, B <= 1030, mortgage(100000, T, 1.01, 1030, B).
```

Figure 10.5: *Four queries for the mortgage program*

```
double mg1(p, t, i, r)
double p, t, i, r;
{
  if (t == 1.0) return (p*i - r);
  else if (t > 1.0 && p >= 0.0)
    return mg1(p*i - r, t - 1.0, i, r);
  else exit(1);
}
```

Figure 10.6: *C function for Q_1*

```
double mg2(t, i, r, b)
double t, i, r, b;
{
  double p;
  if (t == 1.0) return ((b + r)/i);
  else if (t > 1.0) {
    p = (mg2(t - 1.0, i, r, b) + r)/i;
    if (p >= 0.0) return p; else exit(1);
  } else exit(1);
}
```

Figure 10.7: *C function for* Q_2

query	CLP($\mathcal{R}$) *interpreter*	CLP($\mathcal{R}$) *core CLAM*	CLP($\mathcal{R}$) *full CLAM but no redundancy*	CLP($\mathcal{R}$) *full CLAM*	*C program*
Q_1	0.10	0.05	0.0042	0.0042	0.0010
Q_2	2.08	1.78	0.0054	0.0054	0.0016
Q_3	2.35	1.84	1.05	0.0700	n/a
Q_4	4.20	2.05	1.78	0.0640	n/a

Figure 10.8: *Timings for Mortgage program*

```
mortgage(P,T,I,R) =
  if T > 1
    T1 ← T − 1
    if P ≥ 0
      P1 ← P * I − R
      return mortgage(P1,T1,I,R)
    else fail
  else if T = 1
    return P * I − R
  else fail

?-P ← 100000
  T ← 360
  I ← 1.01
  R ← 1025
  B ← mortgage(P,T,I,R)
```

Figure 10.9: High-level pseudocode: mortgage specialized for Q_1

```
mortgage(T,I,R,B) :-
  if T > 1
    T1 ← T − 1
    P1 ← mortgage(T1,I,R,B).
    if P ≥ 0
      return P
    eqn( P1 = P * I - R )
    else fail
  else if T = 1
    V1 ← (B + R)/I
    return V1
  else fail

?-T ← 360
  I ← 1.01
  R ← 1025
  B ← 12625.9
  P ← mortgage(T,I,R,B)
```

Figure 10.10: High-level pseudocode: mortgage specialized for Q_2

```
mortgage(P,T,I,R,B) :-
  if T > 1
    T1 ← T − 1
    fr_ineq( P >= 0 )
    eqn( P1 = P * I - R )
    mortgage(P1,T1,I,R,B)
  else if T = 1
    eqn( B = P * I' - R )
  else fail

?-T ← 360
  I ← 1.01
  mortgage(P,T,I,R,B)
```

Figure 10.11: High-level pseudocode: mortgage specialized for Q_3

```
mortgage(P,T,I,R,B) :-
  fr_ineq( T > 1 )
  eqn( T1 = T - 1 )
  if P ≥ 0
    P1 ← P * I − R
    mortgage(P1,T1,I,R,B)
  else fail
mortgage(P,T,I,R,B) :-
  eqn( T = 1 )
  V1 ← P * I − R
  eqn( B = V1 )

?-ineq( 0 <= B )
  ineq( B <= 1030 )
  P ← 100000
  I ← 1.01
  R ← 1030
  mortgage(P,T,I,R,B)
```

Figure 10.12: High-level pseudocode: mortgage specialized for Q_4

```
mg    try          mg1, 5
      trust        mg2
mg1   initpf       -1
      addpf_val_+  #T
      solve_gt0
      initpf       1
      addpf_val    -1, #T
      addpf_var_+  #tmp1
      solve_eq0
      initpf_0
      addpf_val_+  #P
      solve_ge0
      mult_Vvv     #tmp3, #P, #I
      initpf_0
      addpf_val_+  #R
      addpf_val    -1, #tmp3
      addpf_var_+  #tmp2
      solve_eq0
      getvar       #P, #tmp2
      getvar       #T, #tmp1
      jump         mg
mg2   gettnum      1, #T
      mult_Vvv     #tmp1, #P, #I
      initpf_0
      addpf_val_+  #R
      addpf_val    -1, #tmp1
      addpf_val_+  #B
      solve_eq0
      proceed
```

Figure 10.13: Core CLAM code for general mortgage program

```
mg      subcf           #T, 1, #tmp
        jgtf            #tmp, mg1
        jeqf            #tmp, mg2
        fail
mg1     mvf             #tmp, #T
        jgef            #P, cont
        fail
cont    mulf            #P, #I, #tmp
        subf            #tmp, #R, #P
        jump            mg
mg2     mulf            #P, #I, #tmp
        subf            #tmp, #R, #B
        proceed
```

Figure 10.14: Extended CLAM code for mortgage program, specialized for Q_1

```
mg      save            0
        subf            #T, 1, #tmp
        jgtf            #tmp, mg1
        jeqf            #tmp, mg2
        fail
mg1
        mvf             #tmp, #T
        callp           mg
        addf            #P, #R, #tmp
        divf            #tmp, #I, #P
        jgtf            #P, cont1
        jeqf            #P, cont1
        fail
cont1
        restore
        proceed
mg2
        addf            #R, #B, #tmp
        divf            #tmp, #I, #P
        restore
        proceed
```

Figure 10.15: Extended CLAM code for mortgage program, specialized for Q_2

```
mg      subcf           #T, 1, #tmp
        jgtf            #tmp, mg1
        jeqf            #tmp, mg2
        fail
mg1     subcf           #T, 1, #T
        jgtf            #T, cont1
        fail
cont1   initpf_0
        addpf_val_+     #P
        solve_no_add_ge0
        initpf_0
        addpf_fp_val    #I, #P
        addpf_val_-     #R
        solve_no_fail_eq #tmp3
        getvar          #P, #tmp3
        jump            mg
mg2     subcf           #T, 1, #tmp4
        jeqf            #tmp4, cont2
        fail
cont2   subfc           0, #I, #tmp2
        initpf_0
        addpf_val_+     #R
        addpf_fp_val    #tmp2, #P
        addpf_val_+     #B
        solve_eq0
        proceed
```

Figure 10.16: Extended CLAM code for mortgage program, specialized for Q_3

```
mg
        try             mg1, 9
        trust           mg2
mg1
        initpf          -1
        addpf_val_+     #T
        solve_no_add_gt0
        initpf          -1
        addpf_val_+     #T
        solve_no_fail_eq #tmp
        jgtf            #P, cont1
        jeqf            #P, cont1
        fail
cont1
        mulf            #P, #I, #tmp1
        subf            #tmp1, #R, #P
        gettvar         #T, #tmp
        jump            mg
mg2
        initpf          -1
        addpf_val_+     #T
        solve_eq0
        mulf            #P, #I, #tmp1
        subf            #tmp1, #R, #tmp2
        initpf_fp       #tmp2
        addpf_val_-     #B
        solve_eq0
        proceed
```

Figure 10.17: Extended CLAM code for mortgage program, specialized for Q_4

Chapter 11

Efficient Implementation of Elf

The objective of this chapter is to give some specific proposals for an efficient implementation of Elf, based on the ideas and techniques presented in the rest of this thesis. Many of the comments made about Elf will also be applicable to λProlog. This chapter does not describe any existing implementation, but the prescriptions are based heavily on the empirical study described in Section 5.6.

We begin by discussing the problem of implementing CLP languages with equality constraints on typed λ-expressions. Then we give an overview of higher-order unification, the standard technique for solving these constraints for the simply-typed λ-calculus, as described by Huet [76], and extended by Elliott to dependent types [50]. We restrict the constraint problem as described in Chapter 5 to avoid the hardest cases, and thus describe a modified form of higher-order unification, and show how this fits in with the methodology for managing hard constraints as described in Chapter 8. Finally, based on empirical evidence we describe how to prioritize constraint solving so that the frequent simple cases are solved with relatively low overhead.

11.1 Higher-Order Unification

To understand the relevant points of higher-order unification, we need to take another look at first-order unification. Recall that the Robinson unification algorithm for first-order terms is usually formulated as a set of rewrite rules on a set of *disagreement pairs*, each of which is a pair of terms, and a most general substitution for the variables is required making the terms in each such pair equal. This is called a most general unifier (mgu), which is guaranteed to exist if the equality constraints are satisfiable, and to be unique modulo variable renaming. Note that the mgu is a solved form for the set of equality constraints in the sense that it is a representation of certain eliminable variables as terms possibly including non-eliminable, or parametric variables.

Higher-order unification can be described in a similar way, but there are complications. The notion of uniqueness for solutions is now modulo not only re-naming existential variables, but also $\alpha\beta\eta$-convertibility. As we have already discussed, in this respect most general unifiers do not exist. If we require answers in solved form as

described above, the best we can do is an algorithm that will enumerate the possibly infinite set of pre-mgus. That is, each solution will be an instance of at least one of the unifiers. As has already been mentioned in Chapter 5, this has unpleasant practical consequences. However, if we relax the solved form requirement, a pre-unification algorithm is available. If a set of constraints is satisfiable, the algorithm terminates with a solution in the usual form, together with some equality constraints that do not conform to the solved form, but are known to be satisfiable. For reasons that will be made clear below, these constraints are called $Flex - Flex$ pairs. Thus unification is now, at a pragmatic level, more like general constraint solving, since a store is needed to maintain the set of substitutions and Flex-Flex pairs. At times, Flex-Flex pairs will disappear in the forward direction as more information becomes available, and then have to be restored on backtracking. Note that these constraints are not "hard" or "delayed" in the usual sense, as their satisfiability is guaranteed rather than assumed, but many of the same implementation issues arise.

The pre-unification problem is still undecidable, and so this algorithm still may not terminate when no solution exists. This is because it is still non-deterministic, requiring a search through possibly infinite sets of unifiers for some sub-problem, for one that corresponds to a solution of another part. If none exists, the search may be infinite, but if a solution does exist, it will eventually be found. This problem arises because of some other kinds of disagreement pairs, the so-called $Flex - Rigid$ pairs, and $Flex - Gvar$ pairs, a special case of $Flex - Flex$ pairs. (and of course their symmetric cases). To illustrate these situations, consider two constraints c_1 and c_2, where c_1 has an infinite set of pre-mgus, c_2 has a finite one, but c_1 and c_2 are jointly unsatisfiable. The pre-unification algorithm will choose the first solution to c_1, and then search through the solutions to c_2 to find one that is compatible. None will be found, and the same will happen for the second solution to c_1, and the third, and so on. The procedure doesn't terminate, since it never runs out of solutions to c_1 to try, and none are consistent with any of the solutions to c_2.

In Chapter 5 it was argued that Flex-Rigid and Flex-Gvar pairs will occur rather rarely, and in Section 5.6 the empirical results bore this out, as they occurred in only one of the examples. Furthermore, it was shown that these can be deferred until they become simpler through additional information becoming available. This means that we can classify them as hard constraints, and delay them until they are simplified sufficiently to avoid this problem. Then unification itself will always terminate, but of course the usual potential problems of search control that are associated with delay are introduced.

There is an additional complication, in that Flex-Flex pairs can actually give rise to hard constraints when further information becomes available. This will be discussed further in Section 11.2.

It should be noted that Elliott's extension of Huet's pre-unification algorithm, and Pfenning's extension of Miller's algorithm for the L_λ subset are incomparable in terms of which disagreement pairs they actually solve. This is summarized in the set inclusion diagram in Figure 11.1. Of course neither of them solve Flex-Flex pairs, or for the most part Flex-Gvar pairs, although the latter can be solved by the Miller/Pfenning

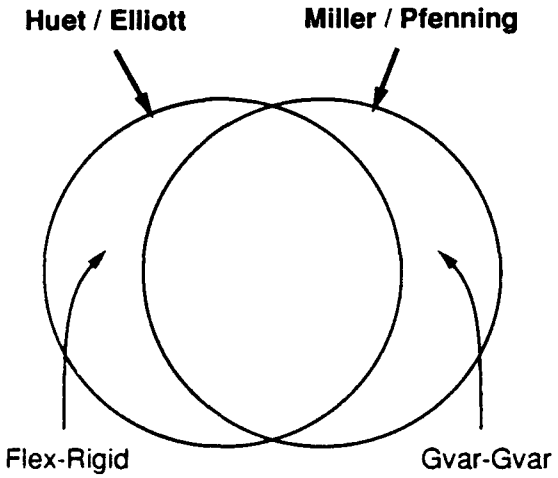


Figure 11.1: Comparison of solutions in Huet's and Miller's algorithms

algorithm as an optimization for certain special cases. We should also note that in this discussion the Flex case does not include the Gvar case, unlike in Huet's usage.

Now, let us list the various forms fo terms in disagreement pairs. For higher-order unification for dependent types, the interested reader is referred to [51] — we restrict the discussion to the simply-typed subset. An Evar is an existential variable, and a Uvar is a temporary constant introduced by pi-quantification. An Evar E is said to *depend* on a Uvar x in a goal if E is bound inside the scope of x in the goal.

- **Flex**
  $FM_1M_2\cdots M_n, n \geq 0$ where F is an Evar and the M_i are terms, and the Gvar conditions below are not satisfied.

- **Gvar**
  $Fx_1x_2\cdots x_m, 0 \leq m \leq n$ where F is an Evar, the x_i are Uvars, F does not depend on any x_i, and the x_i are all distinct.

- **Rigid**
  $fM_1M_2\cdots M_n, n \geq 0$ where f is a constant or a Uvar and the M_i are terms.

Note that, to simplify the discussion, types have been omitted, and the the other classes of disagreement pairs have been omitted since they are dealt with by straightforward recursive unifications.

11.2 Hard Constraints

In this section, our aim is to describe the management of hard constraints in Elf in terms of the framework developed in Chapter 8. We will work towards this goal somewhat gradually.

Core Elf Unification Table			
	Gvar	Flex	Rigid
Gvar	unify		
Flex	delay	delay	
Rigid	unify	delay	unify

Figure 11.2: Core unification table for Elf

The situation is described in Figure 11.2.

Because a Flex-Flex pair, which is not a hard constraint, can lead to a hard constraint we include Flex-Flex pairs in the wakeup system. Hence, ideally, we need five wakeup degrees in addition to *awakened*, These are for Flex-Flex, Flex-Rigid, Rigid-Flex, Flex-Gvar, and Gvar-Flex. We note that it is not desirable to combine symmetric cases. This is because the transitions of the two sides of the equation depend on the binding of different variables, and we need to distinguish them for efficient implementation.

The transitions between these three forms of expressions that we need to consider are as follows. Note that we do not consider leading abstractions.

1. *Flex* $\Rightarrow$ *Rigid*
   The head F is bound to
   $$\lambda x_1 \cdots \lambda x_k.\, g N_1 \cdots N_m$$
   where g is a constant or a Uvar and the N_i are terms. The resulting Rigid term will, of course, be of the form
   $$g P_1 \cdots P_l.$$

2. *Gvar* $\Rightarrow$ *Rigid*
   Same as *Flex* $\Rightarrow$ *Rigid.*

3. *Flex* $\Rightarrow$ *Gvar*

   (a) All of the arguments are bound to universal variables, such that the partial permutation property holds. (This is very unlikely, and expensive to check for, so it has not been implemented to date).

   (b) The head F is bound to
   $$\lambda x_1 \cdots \lambda x_k.\, G y_1 \cdots y_m$$
   where G is an existential variable, each y_j is either a Uvar or one of the x_i, and the resulting term is a Gvar. That is,
   $$G z_1 \cdots z_l$$
   such that the z_i are all distinct Uvars, and G does not depend on any of them.

4. *Gvar ⇒ Flex*
   The head F is bound to

$$\lambda x_1 \cdots \lambda x_k.\, G N_1 \cdots N_m$$

   where G is a an existential variable and the N_i are terms, such that the Gvar criteria are now violated. The resulting Flex term will be of the form

$$G P_1 \cdots P_l.$$

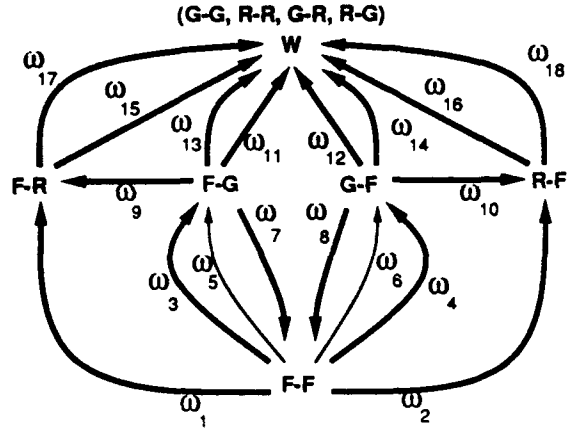
Notice that the above transitions admit the possibility of cycles. A Flex term can turn into a Gvar term when more information becomes available, and with still more information may turn back into a Flex term, all without backtracking. This makes the wakeup system cyclic, as shown in Figure 11.3. Note that the two arcs shown using a thinner line correspond to the case that is expensive and unlikely, and might be omitted. We describe the generic wakeup conditions in symmetric pairs, to avoid notational clutter, and ignore the term that does not change in each pair. The language in our description of the degrees and wakeup conditions is different to that used in Chapter 8, to make it more intuitive in this context.

11.3 Easy Cases of Constraints

Finally we consider details of the actual implementation of higher-order unification, given the empirical data of section 5.6. Recall that that most unification was either simple assignment or first-order (Herbrand) unification, around 95%, averaged over all examples. Similarly, substitution was the substitution of parameters for λ-bound variables in about 95% of the cases. The remaining 5% were substitution of constants, variables, or compound terms for bound variables.

The aim is to find a representation for Elf terms, substitutions and constraints that is suitable for the general problem at hand but is especially suited to the frequent simple cases. The obvious representation is that corresponding to first-order abstract syntax: a DAG with special nodes for application, abstraction etc. This is problematic because of the frequency of the Rigid-Rigid case in unification — remembering that unification avoided by indexing represents otherwise uncounted instances of Rigid-Rigid unification.

Representing application in the obvious way, as shown in the example in Figure 11.4, is particularly problematic. The term shown in the example is $F M_1 M_2 M_3 M_4$. Consider the problem of classifying a disagreement pair. The classifications are mostly based on the nature of the head: in particular whether it is bound to a constant or not. Furthermore, in the frequently occurring Rigid-Rigid case, it is necessary to know *which* constant: as the pair is only decomposed into argument pairs if the heads are identical. In this representation, obtaining information about the head is too expensive. It is necessary to have the head immediately accessible, and then the arguments might as well be a list. This representation, for the same example, is shown in Figure 11.5.



F	$FM_1M_2\cdots M_n$
G	$Fx_{\phi_1}x_{\phi_2}\cdots x_{\phi_m}$
R	$fM_1M_2\cdots M_n$

F -> R	$(\omega_1, \omega_2, \omega_{13}, \omega_{14}, \omega_{15}, \omega_{16})$	$F = \lambda x_1\cdots\lambda x_k.\, gN_1\cdots N_m$
F -> G	(ω_5, ω_6)	$M_1 = x_1 \wedge \cdots \wedge M_n = x_n$
		$\wedge \quad Gvar(Fx_1\cdots x_n)$
F -> G	$(\omega_3, \omega_4, \omega_{11}, \omega_{12}, \omega_{17}, \omega_{18})$	$F = \lambda x_1\cdots\lambda x_k.\, Gy_1\cdots y_m$
		$\wedge \quad (\lambda x_1\cdots\lambda x_k.\, Gy_1\cdots y_m)M_1\cdots M_n \rightarrow M'$
		$\wedge \quad Gvar(M')$
G -> R	(ω_9, ω_{10})	$F = \lambda x_1\cdots\lambda x_k.\, gN_1\cdots N_m$
G -> F	(ω_7, ω_8)	$F = \lambda x_1\cdots\lambda x_k.\, GN_1\cdots N_m$
		$\wedge \quad (\lambda x_1\cdots\lambda x_k.\, GN_1\cdots N_m)M_1\cdots M_n \rightarrow M'$
		$\wedge \quad \neg Gvar(M')$

Figure 11.3: Wakeup system for Elf

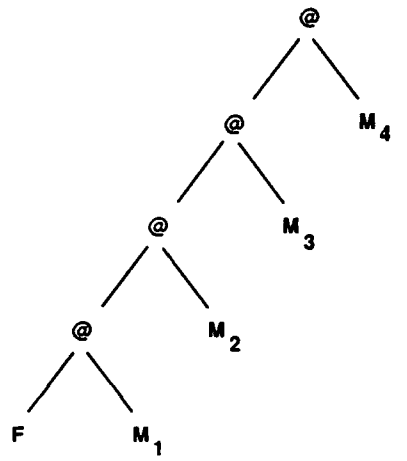


Figure 11.4: Conventional term representation for Elf

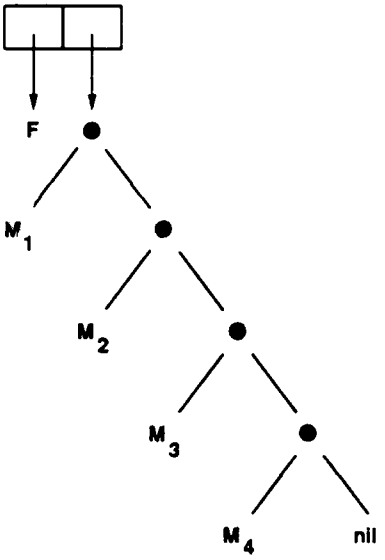


Figure 11.5: Functor/Arguments term representation for Elf

Notice that this representation also makes it easier to make use of clause indexing on rigid term heads.

Note that this functor/argument representation can be problematic when the head is flexible, since it can then be bound to an expression which requires some normalization, producing a new head, and the old one needs to be stored for backtracking in some way. These complications are outbalanced by the efficiency improvement for the simple cases.

Note also that this discussion hinges on rigid terms being in close correspondence with constructed terms in Prolog. The constant head is the functor, and the remaining terms are the arguments. There is then also a close correspondence with Prolog indexing. In short, Prolog unification is an important special case of Elf equality constraints, and this fact affects the representation of application especially. We conclude that functor/argument representation, as was in fact used in Nadathur's original implementation of λProlog up to LP2.7, is an essential optimization for a λProlog or Elf implementation.

11.4 Other Implementation Issues

A number of other representation and implementation issues arise from a study of the empirical data in Section 5.6. Here we make a number of brief comments on those issues.

- *Term Comparison*
  In principle, an important part of term comparison in these languages is the test for α-convertibility. The Duke representation proposal [127] suggests a De Bruijn [38] representation of terms for this reason. While that suggestion may well be appropriate, we should note that the empirical study showed the comparison of two abstractions to be a rare occurrence, and hence it should be probably not be allowed to determine the choice of term representation.

- *Substitution and Abstraction*
  The obvious way to handle substitutions is to produce explicitly the new term. However, this is time-consuming since the function applied must be traversed, and space consuming, since multiple copies of a large term may have to be made. However, other representations of substitutions tend to be complicated in general. This is not the case for substitutions of Uvars. Furthermore, most of the substitutions actually performed are Uvars, and these are frequently abstracted over as well. Abstraction is typically less expensive when substitutions are not actually made, but represented in some explicit way. The Duke proposal advocates using embellished environments to represent substitutions efficiently.

- *Building Deductions*
  Building deductions is expensive. However, it need only be done when the overall

deduction is requested by the user, or when a deduction is needed for further computation. In this sense goals that do not use partial deductions for computation lead to behavior that is closer to that of λProlog.

- *Types*
  Type computations in Elf are a significant part of the overall work done, and eliminating redundant type computations is likely to be important.

- *Extended Occurs Check*
  The occurs check is particularly important in λProlog and Elf, as programs quite commonly depend on it to run correctly. However, even local analysis of rules can be used to avoid many uses of it in most Elf programs. This is because heads of rules are typically linear, so that each variable occurs only once, and thus the occurs check can largely be avoided when these variables are bound. Likewise the dependency check can often be avoided.

Part IV

Conclusions

Chapter 12

Conclusions

The main contribution of this thesis has been to demonstrate the practicality of constraint logic programming by using CLP($\mathcal{R}$) as a case study. It has been shown that paying close attention to the relationship between programming methodology, language design and efficient implementation can enable us to design CLP systems that have the expressive power to provide leverage to programmers yet be efficiently implementable.

The specific contributions can be summarized as follows:

- An understanding of CLP($\mathcal{R}$) and Elf programming methodology. This consists of an understanding of both static program structure and the empirical aspects of program execution.
- A discipline for dealing with hard constraints by delaying them at runtime.
- A strategy for structuring constraint solvers to take advantage of frequently occurring simple cases of constraints.
- An understanding of the problems of adapting constraint solvers to the operational model of CLP by making them incremental.
- Design of suitable data structures for incremental constraint solving.
- A strategy and abstract machine design for compiling CLP($\mathcal{R}$) based on the central data structure of the constraint solver.
- A strategy for making use of highly optimizing compilation of CLP($\mathcal{R}$) based on global program analysis and multiple program specialization.

The broad applicability of the techniques developed here has been demonstrated through a second case study. Elf is superficially almost totally unlike CLP($\mathcal{R}$), except for being essentially a CLP language. However, when viewed in the way that this thesis advocates, the two languages, and to varying extents the various other CLP languages, can be seen to be strongly related not only conceptually, but also pragmatically. This relationship is more obvious for most of the other CLP languages discussed, and the parallels are even stronger. The ideas and techniques in this thesis have been validated

in practice to a considerable degree. In particular, the CLP($\mathcal{R}$) systems have been used for a substantial number of non-trivial applications, as described in Chapter 4.

There is still much work to be done in the practical aspects of Constraint Logic Programming. To mention just a few important areas of research:

- Incremental constraint solving for other domains, and for wider class of constraints in the usual domains.
- Practical, highly optimizing compilers for CLP($\mathcal{R}$) and other languages.
- Control and usability issues in language design.
- Pursuit of other application areas.
- A program debugging methodology to accommodate search for causes of failure through constraint unsatisfiability.

It is hoped that this thesis, however, provides a practical guide for future developers of CLP systems.

Appendix A

Electrical Engineering Programs

A.1 Circuit Solver

This program carries out a steady state phasor analysis of RLC circuits. It is called through the predicate `circuit_solve()` which has as arguments

- the angular frequency for the analysis.
- the component list.
- the list of nodes which are to be 'grounded' — otherwise all voltages are relative.
- the 'Selection' list — a list of nodes for which computed information is to be printed.

The circuit is defined by a list of components. Each component is described by the component type, name, value and the nodes to which it is connected. The component type is used to determine the component characteristics.

```
circuit_solve(W, L, G, Selection) :-
    get_node_vars(L, NV),
    solve(W, L, NV, Handles, G),
    format_print(Handles, Selection).

get_node_vars([[Comp, Num, X, Ns]|Ls], NV) :-
    get_node_vars(Ls, NV1),
    insert_list(Ns, NV1, NV).
get_node_vars([], []).

insert_list([N|Ns], NV1, NV3) :-
    insert_list(Ns, NV1, NV2),
    insert(N, NV2, NV3).
insert_list([], NV, NV).

insert(N, [[N, V, I]|NV1], [[N, V, I]|NV1]).
```

```
insert(N, [[N1, V, I]|NV1], [[N1, V, I]|NV2]) :-
    insert(N, NV1, NV2).
insert(N, [], [[N, V, c(0, 0)]]).

solve(W, [X|Xs], NV, [H|Hs], G) :-
    addcomp(W, X, NV, NV1, H),
    solve(W, Xs, NV1, Hs, G).
solve(W, [], NV, [], G) :-
    zero_currents(NV),
    ground_nodes(NV, G).

zero_currents([[N, V, c(0, 0)]|Ls]) :-
    zero_currents(Ls).
zero_currents([]).

ground_nodes(Vs, [N|Ns]) :-
    ground_node(Vs, N),
    ground_nodes(Vs, Ns).
ground_nodes(Vs, []).
ground_node([[N, c(0, 0), I]|Vs], N).
ground_node([[N1, V, I]|Vs], N) :-
    ground_node(Vs, N).

%
% The following rules deal with two-terminal components.
%
addcomp(W, [Comp2, Num, X, [N1, N2]], NV, NV2,
           [Comp2, Num, X, [N1, V1, I1], [N2, V2, I2]]):-
    c_neg(I1, I2),
    iv_reln(Comp2, I1, V, X, W),
    c_add(V, V2, V1),
    subst([N1, V1, Iold1], [N1, V1, Inew1], NV, NV1),
    subst([N2, V2, Iold2], [N2, V2, Inew2], NV1, NV2),
    c_add(I1, Iold1, Inew1),
    c_add(I2, Iold2, Inew2).

%
% Voltage/current relationships for two-terminal components.
%
iv_reln(resistor, I, V, R, W) :-
    c_mult(I, c(R, 0), V).
iv_reln(voltage_source, I, V, V, W).
iv_reln(isource, I, V, I, W).
iv_reln(capacitor, I, V, C, W) :-
    c_mult(c(0, W*C), V, I).
iv_reln(inductor, I, V, L, W) :-
    c_mult(c(0, W*L), I, V).
iv_reln(connection, I, c(0, 0), L, W).
iv_reln(open, c(0, 0), V, L, W).
```

```
iv_reln(diode, I, V, D, W) :-
    diode(D, I, V).

%
% Three rules per diode type.
%
diode(in914, c(I, 0), c(V, 0)) :-
    V < -100, DV = V + 100, I = 10*DV.
diode(in914, c(I, 0), c(V, 0)) :-
    V >= -100, V < 0.6, I = 0.001*V.
diode(in914, c(I, 0), c(V, 0)) :-
    V >= 0.6, DV = V - 0.6, I = 100*DV.

%
% The following rules deal with transistors.
%
addcomp(W, [transistor, Num, X, [N1, N2, N3]], NV, NV3,
           [transistor, Num, X, [N1, V1, I1],
                        [N2, V2, I2], [N3, V3, I3]]):-
    transistor(X, R, Gain),
    c_add(I1, I3, IT),
    c_neg(I2, IT),
    c_add(Vin, V2, V1),
    c_mult(I1, c(R, 0), Vin),
    c_mult(I1, c(Gain, 0), I3),
    subst([N1, V1, Iold1], [N1, V1, Inew1], NV, NV1),
    subst([N2, V2, Iold2], [N2, V2, Inew2], NV1, NV2),
    subst([N3, V3, Iold3], [N3, V3, Inew3], NV2, NV3),
    subst([N4, V4, Iold4], [N4, V4, Inew4], NV3, NV4),
    c_add(I1, Iold1, Inew1),
    c_add(I2, Iold2, Inew2),
    c_add(I3, Iold3, Inew3),
    c_add(I4, Iold4, Inew4).

% We need one fact for each kind of transistor we wish to consider.

transistor(bc108, 1000, 100).

%
% The following rule deals with transformers.
%
addcomp(W, [transformer, Num, X, [N1, N2, N3, N4]], NV, NV4,
           [transformer, Num, X, [N1, V1, I1], [N2, V2, I2],
           [N3, V3, I3], [N4, V4, I4]]):-
    c_neg(I1, I2),
    c_neg(I3, I4),
    c_add(Vin, V2, V1),
    c_add(Vout, V4, V3),
    c_mult(Vout, c(X, 0), Vin),
```

```
    c_mult(I1, c(X, 0), I4),
    subst([N1, V1, Iold1], [N1, V1, Inew1], NV, NV1),
    subst([N2, V2, Iold2], [N2, V2, Inew2], NV1, NV2),
    subst([N3, V3, Iold3], [N3, V3, Inew3], NV2, NV3),
    subst([N4, V4, Iold4], [N4, V4, Inew4], NV3, NV4),
    c_add(I1, Iold1, Inew1),
    c_add(I2, Iold2, Inew2),
    c_add(I3, Iold3, Inew3),
    c_add(I4, Iold4, Inew4).

subst(X, Y, [X|L1], [Y|L1]).
subst(X, Y, [Z|L1], [Z|L2]) :-
    subst(X, Y, L1, L2).

%
% These rules define complex arithmetic.
%

c_mult(c(Re1, Im1), c(Re2, Im2), c(Re3, Im3)) :-
    Re3 = Re1*Re2 + -1*Im1*Im2,
    Im3 = Re1*Im2 + Re2*Im1.

c_add(c(Re1, Im1), c(Re2, Im2), c(Re3, Im3)) :-
    Re3 = Re1 + Re2,
    Im3 = Im1 + Im2.

c_neg(c(Re, Im), c(Re1, Im1)) :-
    Re1 = -Re, Im1 = -Im .
c_eq(c(Re1, Im1), c(Re2, Im2)) :-
    Re1 = Re2, Im1 = Im2 .

c_real(c(Re, Im), Re).
c_imag(c(Re, Im), Im).

% format_print(H, Selection)  --- Print out node information.
```

A.2 Transistor Circuit Analysis and Design

This program solves transistor amplifier design and analysis problems, and D.C. circuit analysis problems involving transistors, diodes and capacitors. The first main goal, used for dc analysis of circuits, is

```
?- dc_analysis(Vcc1, Vcc2, Circuit).
```

The parameters are two (optional) source voltages which will ensure that the node *cc1* is at voltage *Vcc1*, similarly for *cc2*, and a circuit description in the following form: a list of elements each of which is a list containing four elements, the component type, the component name, the data for the component, and a list of nodes that the component connects.

The second main goal, used for transistor amplifier design and analysis, is

```
?- full_analysis(Vcc1, Vcc2, Circuit, In, Out, Type,
          Stability, Gain, Inresist, Outresist).
```

The first three parameters are as above. The rest are the input node for the amplifier, the output node for the amplifier, and the type (emitter-follower, common-base) of the amplifier. The final four parameters are design parameters: the stability of collector currents on 50% deviation of Beta values and 10% deviation of *Vbe* values for the transistors of the amplifier, the gain of the amplifier, and finally the open-circuit input resistance and output resistance of the amplifier.

Note that several conventions are followed in this program. In the D.C. case, capacitors are considered to be open circuits. However, for small signal analysis, they are considered to be short circuits. Additionally, it is assumed that any amplifier circuit type used for a full analysis appears in the database of circuits and that all components used appear in the database of components.

```
%
%  Entry Points.
%

dc_analysis(Vcc1, Vcc2, Circuit):-
      choose_circuit(Circuit),
      solve_dc(mean, Circuit, [n(cc1, Vcc1, [_]),
            n(cc2, Vcc2, [_]), n(gnd, 0, [_])],
            Nodelist, Collector_Currents),
      current_solve(Nodelist),
      print_circuit(Circuit),
      print_value(Nodelist).

full_analysis(Vcc1, Vcc2, Circuit, In, Out, Type,
                  Stability, Gain, Inresist, Outresist):-
      % Choose a circuit template.
      circuit(Vcc1, Vcc2, Circuit, In, Out, Type),
      % Construct circuit constraints.
      solve_dc(mean, Circuit, [n(cc1, Vcc1, [_]),
            n(cc2, Vcc2, [_]), n(gnd, 0, [_])],
            Nodelist, Collector_Currents),
      current_solve(Nodelist),
      % Determine stability constraints.
      stability(Vcc1, Vcc2, Circuit, Collector_Currents, Stability),
      % Determine input resistance and gain constraints.
      solve_ss(Circuit, Collector_Currents,
            [n(cc1, 0, [_]), n(cc2, 0, [_]),
             n(gnd, 0, [_]), n(In, 1, [Iin]),
             n(Out, Vout, [])], Nodelist2),
      current_solve(Nodelist2),
      Inresist = -1 / Iin,
      Gain = Vout,
```

```
      % Determine Output resistance constraints
      solve_ss(Circuit, Collector_Currents,
            [n(cc1, 0, [_]), n(cc2, 0, [_]),
             n(gnd, 0, [_]), n(Out, 1, [Iout])], Nodelist3),
      current_solve(Nodelist3),
      Outresist = -1 / Iout,
      % Choose circuit values - all (real) choice points occur here
      choose_circuit(Circuit).

%
%  Small signal equivalent circuit analysis.
%
solve_ss([], [], List, List).
solve_ss([[Component, _, Data, Points]|Rest], CCin,
         Innodes, Outnodes):-
      connecting(Points, Volts, Amps, Innodes, Tmpnodes),
      component_ss(Component, Data, Volts, Amps, CCin, CCout),
      solve_ss(Rest, CCout, Tmpnodes, Outnodes).

component_ss(resistor, R, [V1, V2], [I, -1*I], Cc, Cc):-
      V1-V2 = R*I.
component_ss(capacitor, _, [V, V], [I, -1*I], Cc, Cc).
component_ss(transistor, [npn, Code, active, Mean, _, _],
        [Vb, Vc, Ve], [Ib, Ic, Ie], [Icol|CC], CC):-
      Mean = data(Beta, _, _, Vt),
      Vb - Ve = (Beta*Vt / Icol)*Ib,
      Ic = Beta*Ib,
      Ie + Ic + Ib = 0.


%
%  D.C. component solving.
%
solve_dc(_, [], List, List, []).
solve_dc(Kind, [[Component, _, Data, Points] | Rest],
               Inlist, Outlist, CCin):-
      connecting(Points, Volts, Amps, Inlist, Tmplist),
      component_dc(Component, Data, Volts, Amps, CCin, CCout, Kind),
      solve_dc(Kind, Rest, Tmplist, Outlist, CCout).

component_dc(resistor, R, [V1, V2], [I, -1*I], Cc, Cc, _):-
      V1-V2 = R*I.
component_dc(capacitor, _, [V1, V2], [0, 0], Cc, Cc, _).
component_dc(transistor, [Type, Code, State, Mean, Min, Max],
        Volts, [Ib, Ic, Ie], [Ic|CC], CC, mean):-
      Mean = data(Beta, Vbe, Vcestat, _),
      transistor_state(Type, State, Beta, Vbe, Vcesat, Volts,
                       [Ib, Ic, Ie]).
component_dc(transistor, [Type, Code, State, Mean, Min, Max],
```

```
        Volts, [Ib, Ic, Ie], [Ic|CC], CC, minn):-
      Min = data(Beta, Vbe, Vcestat, _),
      transistor_state(Type, State, Beta, Vbe, Vcesat, Volts,
                       [Ib, Ic, Ie]).
component_dc(transistor, [Type, Code, State, Mean, Min, Max],
        Volts, [Ib, Ic, Ie], [Ic|CC], CC, maxx):-
      Max = data(Beta, Vbe, Vcestat, _),
      transistor_state(Type, State, Beta, Vbe, Vcesat, Volts,
                       [Ib, Ic, Ie]).
component_dc(diode, [Code, State, Vf, Vbreak], Volts, Amps,
             Cc, Cc, _):-
      diode_state(State, Vf, Vreak, Volts, Amps).

%
%  Diode and transistor states / relationships.
%

diode_state(forward, Vf, Vbreak, [Vp, Vm], [I, -1*I]):-
      % forward biased
      Vp - Vm = Vf,
      I >= 0.
diode_state(reverse, Vf, Vbreak, [Vp, Vm], [I, -1*I]):-
      % reverse biased
      Vp - Vm < Vf,
      Vm - Vp < Vbreak,
      I = 0.

transistor_state(npn, active, Beta, Vbe, _, [Vb, Vc, Ve],
                                            [Ib, Ic, Ie]):-
      Vb = Ve + Vbe,
      Vc >= Vb,
      Ib >= 0,
      Ic = Beta*Ib,
      Ie+Ib+Ic = 0.
transistor_state(pnp, active, Beta, Vbe, _, [Vb, Vc, Ve],
                                            [Ib, Ic, Ie]):-
      Vb = Ve + Vbe,
      Vc <= Vb,
      Ib <= 0,
      Ic = Beta*Ib,
      Ie+Ib+Ic = 0.
transistor_state(npn, saturated, Beta, Vbe,
                      Vcesat, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb = Ve + Vbe,
      Vc = Ve + Vcesat,
      Ib >= 0,
      Ic >= 0,
      Ie+Ib+Ic = 0.
transistor_state(pnp, saturated, Beta, Vbe,
```

```
                    Vcesat, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb = Ve + Vbe,
      Vc = Ve + Vcesat,
      Ib <= 0,
      Ic <= 0,
      Ie+Ib+Ic = 0.
transistor_state(npn, cutoff, Beta, Vbe,
                    Vcesat, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb <= Ve + Vbe,
      Ib = 0,
      Ic = 0,
      Ie = 0.
transistor_state(pnp, cutoff, Beta, Vbe,
                    Vcesat, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb >= Ve + Vbe,
      Ib = 0,
      Ic = 0,
      Ie = 0.

%
%  Component connections.
%
connecting([], [], [], List, List).
connecting([P|PR], [V|VR], [I|IR], Inlist, Outlist):-
      connect(P, V, I, Inlist, Tmplist),
      connecting(PR, VR, IR, Tmplist, Outlist).

connect(P, V, I, [], [n(P, V, [I])]) :-
      !.
connect(P, V, I, [n(P, V, Ilist) | Rest], [n(P, V, [I|Ilist])|Rest]) :-
      !.
connect(P, V, I, [A|Rest], [A|Newrest]) :-
      connect(P, V, I, Rest, Newrest).

%
%  Stability analysis.
%
stability(Vcc1, Vcc2, Circuit, CollectorCurrents, Stability):-
      solve_dc(minn, Circuit, [n(cc1, Vcc1, [_]),
          n(cc2, Vcc2, [_]), n(gnd, 0, [_])],
          Nodelist1, MinCurrents),
      current_solve(Nodelist1),
      solve_dc(maxx, Circuit , [n(cc1, Vcc1, [_]),
          n(cc2, Vcc2, [_]), n(gnd, 0, [_])],
          Nodelist2, MaxCurrents),
      current_solve(Nodelist2),
      calculate(MinCurrents, MaxCurrents,
                CollectorCurrents, Stability).
```

```
calculate(MinCurrents, MaxCurrents, CollectorCurrents, Stability):-
      cal(MinCurrents, MaxCurrents, CollectorCurrents, Percents),
      maxi(Percents, 0, Stability).

cal([Min|Rin], [Max|Rax], [Ic|Rc], [Pc|Rpc]):-
      Pc = max(Ic-Min, Max-Ic),
      cal(Rin, Rax, Rc, Rpc).
cal([], [], [], []).

maxi([N1|R], N2, P):-
      M = max(N1, N2),
      maxi(R, M, P).
maxi([], P, P).


current_solve([]).
current_solve([n(_, _, L) | Rest]) :-
      kcl(L),
      current_solve(Rest).

print_value([]).
print_value([n(P, V, I) | Rest]) :-
      printf("% at % %\n", [P, V, I]),
      print_value(Rest).

print_circuit([]).
print_circuit([[Comp, Name, Data, Points] | Rest]) :-
      printf("  % at % %\n", [Comp, Name, Data]),
      print_circuit(Rest).

sum([X|T], Z) :-
      X+P = Z,
      sum(T, P).
sum([], 0).

kcl(L) :-
      sum(L, 0).

%
%  Choose circuit values.
%
choose_circuit([[Component_type, _, Data, _] | RestofCircuit]):-
      choose_component(Component_type, Data),
      choose_circuit(RestofCircuit).
choose_circuit([]).

choose_component(resistor, R):-
      resistor_val(R).
choose_component(capacitor, _).
```

```
choose_component(diode, [Code, _, Vf, Vbreak]):-
      diode_type(Code, Vf, Vbreak).
choose_component(transistor, [Type, Code, _, Mean, Min, Max]):-
      transistor_type(Type, Code, MeanBeta,
                           MeanVbe, MeanVcestat, MeanVt, mean),
      Mean = data(MeanBeta, MeanVbe, MeanVcestat, MeanVt),
      transistor_type(Type, Code, MinBeta,
                           MinVbe, MinVcestat, MinVt, minn),
      Min = data(MinBeta, MinVbe, MinVcestat, MinVt),
      transistor_type(Type, Code, MaxBeta,
                           MaxVbe, MaxVcestat, MaxVt, maxx),
      Max = data(MaxBeta, MaxVbe, MaxVcestat, MaxVt).

%
%  Database of circuits and components.
%
resistor_val(100).
resistor_val(50).
resistor_val(27).
resistor_val(10).
resistor_val(5).
resistor_val(2).
resistor_val(1).

diode_type(di1, 0.6, 100).

transistor_type(npn, tr0, 100, 0.7, 0.3, 0.025, mean).
transistor_type(npn, tr0, 50, 0.8, 0.3, 0.025, minn).
transistor_type(npn, tr0, 150, 0.6, 0.3, 0.025, maxx).

transistor_type(pnp, tr1, 100, -0.7, -0.3, 0.025, mean).
transistor_type(pnp, tr1, 50, -0.8, -0.3, 0.025, minn).
transistor_type(pnp, tr1, 150, -0.6, -0.3, 0.025, maxx).

circuit(15, 0, [
    [capacitor, c1, c1, [in, b]],
    [resistor, r1, R1, [b, cc1]],
    [resistor, r2, R2, [b, gnd]],
    [transistor, tr, [npn, tr0, active, Mean, Minn, Maxx], [b, c, e]],
    [resistor, re, Re, [e, gnd]],
    [capacitor, c2, c2, [c, out]],
    [resistor, rc, Rc, [c, cc1]],
    [capacitor, c3, c3, [e, gnd]]],
  in, out, common_emitter).

circuit(15, 0, [
    [capacitor, c1, C1, [gnd, b]],
    [resistor, r1, R1, [b, cc1]],
    [resistor, r2, R2, [b, gnd]],
```

```
    [transistor, tr, [pnp, tr1, active, Mean, Minn, Maxx], [b, c, e]],
    [resistor, re, Re, [e, gnd]],
    [capacitor, c2, C2, [c, in]],
    [resistor, rc, Rc, [c, cc1]],
    [capacitor, c3, C3, [e, out]]],
  in, out, common_base).

circuit(15, 0, [
    [capacitor, c1, C1, [in, b]],
    [resistor, r1, R1, [b, cc1]],
    [resistor, r2, R2, [b, gnd]],
    [transistor, tr, [npn, tr0, active, Mean, Minn, Maxx],
        [b, cc1, e]],
    [resistor, re, Re, [e, gnd]],
    [capacitor, c3, C3, [e, out]]],
  in, out, emitter_follower).
```

A.3 Signal Flow Graph Simulation

This program simulates the signal flow graph presented to it as input, and describes the value at the required node at each time interval by drawing a graph. The goal is of the form

```
?- flow(Spec, Output).
```

The first argument describes the signal flow graph as a list of arcs and the second is either the name of a node in the graph for which the value is to be plotted, or an empty list signifying that the value at each node is to be printed at each time interval. Each arc description is of the form `[delay, node1, node2]` or `[coeff(coefficient), node1, node2]`.

```
%
% First convert the goal to an internal representation.
%
flow(Spec, Output) :-
        convert_list(Spec, Specd, [Nodes, Sources]),
        get_node_index(Output, Nodes, _, Oindex, no), !,
        analyze(Specd, Oindex).

convert_list([S|Ss], Spec2, Info2) :-
        convert_list(Ss, Spec1, Info1),
        convert_arc(S, Spec1, Spec2, Info1, Info2).
convert_list([], [], [[], Sources]) :-
        read_sources(Sources).

convert_arc([Type, In, Out], Spec1, Spec4, Info1, Info3) :-
        get_index(Type, In, Info1, Info2, Index1, Newnode_flag1),
```

```
        get_index(not_source, Out, Info2, Info3,
                  Index2, Newnode_flag2),
        node_insert(Spec1, Spec2, Newnode_flag1),
        node_insert(Spec2, Spec3, Newnode_flag2),
        insert(Spec3, Spec4, Index2, [Type, Index1]).

get_index(source, N, [Nodes1, Sources], [Nodes1, Sources],
                  Index, no) :-
        get_node_index(N, Sources, _, Index, no).
get_index(Type, Node, [Nodes1, Sources], [Nodes2, Sources],
                  Index, Newnode_flag) :-
        get_node_index(Node, Nodes1, Nodes2, Index, Newnode_flag).

%
% Find an index to a node in the list, adding it if necessary.
%
% get_node_index(Node, Old_node_list, New_node_list, Index,
%                Newnode_flag)
%
get_node_index(X, [X|Ns], [X|Ns], 1.0, no).
get_node_index(X, [Xd|Ns], [Xd|Nds], V + 1, Newnode_flag) :-
        get_node_index(X, Ns, Nds, V, Newnode_flag).
get_node_index(X, [], [X], 1, yes).

%
% Add new nodes by setting up a new list on the specification list.
%
node_insert(S, S, no).
node_insert([S1|Ss1], [S1|Ss2], yes) :-
        node_insert(Ss1, Ss2, yes).
node_insert([], [[]], yes).

%
% Insert the arc in the new specification.
%
insert([S|Ss], [S|Ts], Index, Arc) :-
        Index > 1,
        insert(Ss, Ts, Index-1  Arc).
insert([S|Ss], [[Arc|S]|Ss], 1, Arc).

analyze(Spec, Output) :-
        getinitial(Old),
        sigflow(Old, Spec, Output).

sigflow(Old, Spec, Output) :-
        getsources(Sources),
        stepflow(Old, Sources, New, Spec, New),
        printnodes(New, Output),
        sigflow(New, Spec, Output).
```

```
getsources(Ss) :-
        readsources(Ss).
getsources(0, []).

getinitial(Old) :-
        readinitial(Old).

%
% stepflow(Oldnodes, Sources, Newnodes, Spec, Newnodes_left_to_process)
%
stepflow(Oldnodes, Sources, Newnodes, [S|Spec], [N|New]) :-
        stepflow(Oldnodes, Sources, Newnodes, Spec, New),
        calcnode(Oldnodes, Sources, Newnodes, S, N).
stepflow(Oldnodes, Sources, Newnodes, [], []).

% calcnode(Oldnodes, Sources, Newnodes, Arcs, Newnode)
calcnode(Oldnodes, Sources, Newnodes, [Arc|Arcs], New) :-
        New = New1 + New2,
        calcarc(Oldnodes, Sources, Newnodes, Arc, New1),
        calcnode(Oldnodes, Sources, Newnodes, Arcs, New2).
calcnode(Oldnodes, Sources, Newnodes, [], 0).

%
% calcarc(Oldnodes, Sources, Newnodes, Arc, New)
%
calcarc(Oldnodes, Sources, Newnodes, [coeff(C), Arc], New) :-
        New - C*Value = 0,
        find_index(Value, Newnodes, Arc).
calcarc(Oldnodes, Sources, Newnodes, [delay, NIndex], New) :-
        New - Value = 0,
        find_index(Value, Oldnodes, NIndex).
calcarc(Oldnodes, Sources, Newnodes, [source, Name], New) :-
        New - Value = 0,
        find_index(Value, Sources, Name).

%
% find_index(item, list of items, place in list)
%
find_index(V, [V|Vs], 1).
find_index(VV, [V|Vs], N) :-
        N > 0,
        NN = N-1,
        find_index(VV, Vs, NN).

% printnodes() -  print out output signal
% readsources(), readinital() -  read data from files
```

Appendix B

Natural Semantics Programs

This is an extended excerpt from the Mini-ML natural semantics code, given in full in [117]. We begin with the Mini-ML expression syntax, then give a natural operational semantics. Finally, we give a definition of what it means for a Mini-ML expression to be a *value*, and a program that transforms a deduction of an evaluation into a deduction showing that the result of the evaluation is a value.

B.1 Expressions of Mini-ML

```
exp    : type.

true   : exp.
false  : exp.
if     : exp -> exp -> exp -> exp.

z      : exp.
s      : exp.
pred   : exp.
zerop  : exp.

pair   : exp -> exp -> exp.
fst    : exp -> exp.
snd    : exp -> exp.

lam    : (exp -> exp) -> exp.
app    : exp -> exp -> exp.

let    : exp -> (exp -> exp) -> exp.

letrec : (exp -> exp) -> (exp -> exp) -> exp.
fix    : (exp -> exp) -> exp.
```

B.2 Natural Operational Semantics

Sometimes the most "natural" semantics for a programming language is "nondeterministic" in the sense that its execution would require backtracking in our chosen implementation language. However, the corresponding "deterministic" version is usually more efficient, and corresponds more closely to practical applications. Here we just give the nondeterministic (but nevertheless executable) semantics for Mini-ML (note the nondeterminism in the handling of if-then-else and application). The deterministic semantics may be found in [117].

```
eval     : exp -> exp -> type.

eval_t          : eval true true.
eval_f          : eval false false.
eval_if_t       : eval (if E1 E2 E3) V
                       <- eval E1 true
                       <- eval E2 V.
eval_if_f       : eval (if E1 E2 E3) V
                       <- eval E1 false
                       <- eval E3 V.

eval_z          : eval z z.
eval_s          : eval s s.
eval_pred       : eval pred pred.

eval_zerop      : eval zerop zerop.

eval_pair       : eval (pair E1 E2) (pair V1 V2)
                       <- eval E1 V1
                       <- eval E2 V2.
eval_fst        : eval (fst E) V1
                       <- eval E (pair V1 V2).
eval_snd        : eval (snd E) V2
                       <- eval E (pair V1 V2).

eval_lam        : eval (lam E) (lam E).

eval_app_lam    : eval (app E1 E2) V
                       <- eval E1 (lam E1')
                       <- eval E2 V2
                       <- eval (E1' V2) V.
eval_app_s      : eval (app E1 E2) (app s V)
                       <- eval E1 s
                       <- eval E2 V.
```

```
eval_app_pred_s : eval (app E1 E2) V
                        <- eval E1 pred
                        <- eval E2 (app s V).
eval_app_zerop_t : eval (app E1 E2) true
                        <- eval E1 zerop
                        <- eval E2 z.
eval_app_zerop_f : eval (app E1 E2) false
                        <- eval E1 zerop
                        <- eval E2 (app s V).


eval_let        : eval (let E1 E2) V2
                        <- eval E1 V1
                        <- eval (E2 V1) V2.

eval_letrec     : eval (letrec E1 E2) V
                        <- eval (fix E1) V1
                        <- eval (E2 V1) V2.

eval_fix        : eval (fix E) V <- eval (E (fix E)) V.
```

B.3 The Value Property and Evaluation

B.3.1 The Value Property

These rules describe what it means for a Mini-ML expression to be a value.

```
value        : exp -> type.

val_t        : value true.
val_f        : value false.

val_z        : value z.
val_s        : value s.
val_pred     : value pred.

val_zerop    : value zerop.

val_pair     : value E1 -> value E2 -> value (pair E1 E2).

val_lam      : value (lam E).
val_app_s    : value E -> value (app s E).
```

B.3.2 Transformation of Evaluations to Value Deductions

The vp relation defined here embodies the proof that the result of evaluating any Mini-ML expression is a value. It does this by transforming a deduction of an evaluation judgement into a deduction of a value judgement.

```
vp          : eval E V -> value V -> type.

vp_t        : vp (eval_t) val_t.
vp_f        : vp (eval_f) val_f.
vp_if_t     : vp (eval_if_t P2 P1) VP2 <- vp P2 VP2.
vp_if_f     : vp (eval_if_f P3 P1) VP3 <- vp P3 VP3.

vp_z        : vp (eval_z) val_z.
vp_s        : vp (eval_s) val_s.
vp_pred     : vp (eval_pred) val_pred.

vp_zerop    : vp (eval_zerop) val_zerop.


vp_pair     : vp (eval_pair P2 P1) (val_pair VP1 VP2)
                 <- vp P1 VP1
                 <- vp P2 VP2.

vp_fst      : vp (eval_fst P) VP1 <- vp P (val_pair VP1 VP2).
vp_snd      : vp (eval_snd P) VP2 <- vp P (val_pair VP1 VP2).

vp_lam      : vp (eval_lam) val_lam.

vp_app_lam      : vp (eval_app_lam P3 P2 P1) VP3
                     <- vp P3 VP3.
vp_app_s        : vp (eval_app_s P2 P1) (val_app_s VP2)
                     <- vp P2 VP2.

vp_app_pred_s : vp (eval_app_pred_s P2 P1) VP0
                     <- vp P2 (val_app_s VP0).

vp_app_zerop_t : vp (eval_app_zerop_t P2 P1) val_t.
vp_app_zerop_f : vp (eval_app_zerop_f P2 P1) val_f.

vp_let      : vp (eval_let P2 P1) VP <- vp P2 VP.

vp_letrec : vp (eval_letrec P2 P1) VP <- vp P2 VP.
```

```
vp_fix     : vp (eval_fix P) VP <- vp P VP.
```

Bibliography

[1] A. Aggoun and N. Beldiceanu. Overview of the CHIP compiler system. In Koichi Furukawa, editor, *Proc. 8th International Conference on Logic Programming*, pages 775–789, Paris, France, June 1991. MIT Press.

[2] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[3] Hassan Aït-Kaci and Patrick Lincoln. LIFE: A natural language for natural language. Technical Report ACA-ST-074-88, MCC, 1988.

[4] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):187–215, 1986.

[5] Tod Amon and Gaetano Borriello. An approach to symbolic timing verification. In *Tau '92: 2nd International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Princeton, NJ, March 1992.

[6] Tod Amon and Gaetano Borriello. An approach to symbolic timing verification. In *Proc. 29th ACM/IEEE Design Automation Conference*, pages 410–413, Anaheim, CA, USA, June 1992.

[7] Penny Anderson. *Program Development by Proof Transformation.* PhD thesis, Carnegie Mellon University, 1992. In preparation.

[8] D. S. Arnon. A bibliography of quantifier elimination for real closed fields. *Journal of Symbolic Computation*, 5:267–274, 1988.

[9] Arvind and D.E. Culler. Dataflow architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

[10] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[11] Joachim Beer. The occur-check problem revisited. *Journal of Logic Programming*, 5(3):243–261, 1988.

[12] F. Berthier. Managing Underlying Assumptions of a Financial Planning Model in CHIP. Technical Report TR-LP-39, European Computer Industry Research Centre (ECRC), Munich, Germany, November 1988.

[13] F. Berthier. A financial model using qualitative and quantitative knowledge. In F. Gardin, editor, *Proceeedings of the International Symposium on Computational Intelligence 89*, pages 1–9, Milano, Italy, September 1989.

[14] F. Berthier. Solving Financial Decision Problems with CHIP. In J.-L. Le Moigne and P. Bourgine, editors, *Proceeedings of the 2nd Conference on Economics and Artificial Intelligence - CECIOA 2*, pages 233–238, Paris, France, June 1990.

[15] K.-H. Borgwardt. Some distribution-independent results about the asymptotic order of the average number of pivot steps of the simplex method. *Mathematics of Operations Research*, 7:441–462, 1982.

[16] Alan Borning. Thinglab – a constraint-oriented simulation laboratory. Technical Report SSL-79-3, Xerox PARC, 1979.

[17] Alan Borning. The programming language aspects of ThingLab, a constraint – oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):252–387, October 1981.

[18] R. S. Boyer and J. S. Moore. The sharing of structure in theorem proving programs. *Machine Intelligence*, 7:101–116, 1972.

[19] J. M. Broek and H. A. M. Daniels. Application of constraint logic programming to asset and liability management in banks. *Computer Science in Economics and Management*, 4(2):107–116, May 1991.

[20] M. Bruynooghe. The memory management of Prolog implementations. In K. L. Clark and S.-A. Tarnlund, editors, *Proceedings of the 1st International Workshop on Logic Programming, 1980*, pages 83–98, Debrecen, Hungary, 1982. Academic Press.

[21] W. Büttner and H. Simonis. Embedding Boolean expressions into logic programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[22] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In Jean-Louis Lassez, editor, *Proc. 4th International Conference on Logic Programming*, pages 40–58, Melbourne, Victoria, Australia, May 1987. MIT Press.

[23] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[24] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[25] J. Cohen. A view of the origins and development of Prolog. *Communications of the ACM*, 31(1):26–36, 1988.

[26] A. Colmerauer. Les systemes-Q ou un formalisme pour analyser et synthesizer des phrases sur ordinateur. Technical Report 43, Dept. d'Informatique, Universite de Montreal, Canada, 1973.

[27] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Une systeme de communication homme-machine en Francais. Technical report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, France, 1973.

[28] Alain Colmerauer. PROLOG II reference manual and theoretical model. Technical report, Groupe Intelligence Artificielle, Université Aix - Marseille II, October 1982.

[29] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-84), ICOT, Tokyo*, pages 85–99, 1984.

[30] Alain Colmerauer. Opening the PROLOG-III universe. *BYTE Magazine*, 12(9), August 1987.

[31] Alain Colmerauer. Final specifications for PROLOG-III. Technical Report P1219(1106), ESPRIT, February 1988.

[32] Alain Colmerauer. An introduction to PROLOG-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[33] Alain Colmerauer. Personal Communication, CLP Workshop, Marseille, January 1991.

[34] Mary Dalrymple, Stuart M. Shieber, and Fernando C. N. Pereira. Ellipsis and higher-order unification. *Linguistics and Philosophy*, 14:399–452, 1991.

[35] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[36] J. Darlington, Y.-K. Guo, and H. Pull. A new perspective on integrating functional and logic languages. In *Fifth Generation Computer Systems*, pages 682–693, Tokyo, Japan, 1992.

[37] Ernest Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, July 1987.

[38] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[39] J. de Kleer and G. J. Sussman. Propagation of constraints applied to circuit synthesis. *Circuit Theory and Applications*, 8:127–144, 1980.

[40] S. K. Debray. *Global Optimization of Logic Programs.* PhD thesis, State University of New York at Stony Brook, 1986.

[41] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, January 1988.

[42] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, April 1989.

[43] Emanuel Derman and Christopher J. van Wyk. A simple equation solver and its application to financial modelling. *Software - Practice and Experience*, 14(12):1169–1181, December 1984.

[44] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving a Cutting-Stock Problem in Constraint Logic Programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Fifth International Conference on Logic Programming*, pages 42–58, Seattle, WA, August 1988. MIT Press.

[45] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving Large Scheduling Problems in Logic Programming. In *EURO-TIMS Joint International Conference on Operations Research and Management Science*, Paris, France, July 1988.

[46] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988.

[47] Veroniek Dumortier, Gerda Janssens, and Maurice Bruynooghe. Detection of free variables in the presence of numeric constraints by means of abstract interpretation. Technical Report CW 145, Derpartment of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A - B-3001 Leuven, Belgium, March 1992.

[48] E. W. Elcock. Absys: The first logic programming language — a retrospective and commentary. *Journal of Logic Programming*, 9(1):1–17, 1990.

[49] E. W. Elcock, J. J. McGregor, and A. M. Murray. Data directed control and operating systems. *British Computer Journal*, 15(2):125–129, 1972.

[50] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, pages 121–136. Springer-Verlag LNCS 355, April 1989.

[51] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[52] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989. Available as Technical Report MS-CIS-87-109.

[53] R. E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[54] J. M. Foster and E. W. Elcock. Absys 1: An incremental compiler for assertions — an introduction. *Machine Intelligence*, 4:423–432, 1969.

[55] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[56] Bjorn N. Freeman-Benson. *Constraint Imperative Programming.* PhD thesis, Department of Computer Science and Engineering, University of Washington, 1991.

[57] Michael M. Gorlick, Carl F. Kesselman, Daniel A. Marotta, and D. Stott Parker. Mockingbird: A logical methodology for testing. *Journal of Logic Programming*, 8(1 & 2):95–119, January/March 1990.

[58] J. Gosling. *Algebraic Constraints.* PhD thesis, Carnegie-Mellon University, 1983. Available as Technical Report CMU-CS-83-132.

[59] T. Graf, P. van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of hybrid circuits in constraint logic programming. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 72–77, Detroit, 1989.

[60] Timothy G. Griffin. Logical interpretations as computational simulations. Draft paper. Talk given at the North American Jumelage, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1991.

[61] John Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, January 1991. Available as technical report MS-CIS-91-09.

[62] John Hannan and Dale Miller. A meta-logic for functional prolgramming. In John Lloyd, editor, *Proceedings of the Workshop on Meta-Programming in Logic Programming*, pages 453–476, Bristol, England, June 1988. University of Bristol.

[63] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[64] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[65] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, To appear. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[66] Robert Harper and Frank Pfenning. Modularity in the LF logical framework. Submitted. Available as POP Report 91–001, School of Computer Science, Carnegie Mellon University, November 1991.

[67] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 131–142, January 1991.

[68] William H. Hayt and Jack E. Kemmerly. *Engineering Circuit Analysis*. McGraw Hill, 1978.

[69] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference*, pages 675–703, Melbourne, Victoria, Australia, May 1987. MIT Press. Also to appear in Journal of Automated Reasoning.

[70] Nevin C. Heintze, Joxan Jaffar, Chean Shen Lim, Spiro Michaylov, Peter J. Stuckey, Roland Yap, and Chut Gneow Yee. The CLP($\mathcal{R}$) programmers manual – version 1. Technical Report 59, Department of Computer Science, Monash University, June 1986.

[71] Jacques Herbrand. Sur la théorie de la démonstration. In W. Goldfarb, editor, *Logical Writings*. Cambridge U. P., 1971. Original Ph. D. Thesis 1930.

[72] C. Hewitt. Planner: A language for proving theorems in robots. In *International Joint Conference on Artificial Intelligence*, pages 295–301, Washington D. C., 1969.

[73] Ralph D. Hill. A 2-D graphics system for multi-user interactive graphics based on objects and constraints. In E. Blake and P. Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the Eurographics Workshop on Object Oriented Graphics*, pages 67–92. Springer Verlag, 1991.

[74] D. S. Homiak. A constraint logic programming system for solving partial differential equations with applications in options valuation. Master's project, DePaul University, 1991.

[75] Hoon Hong. Non-linear constraint solving over real numbers in constraint logic programming (introducing RISC-CLP). Technical Report 92-08, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1992.

[76] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[77] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[78] Tien Huynh and Catherine Lassez. A CLP($\mathcal{R}$) options trading analysis system. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 59–69, Seattle, Washington, U.S.A., 1988. MIT Press.

[79] Tien Huynh and Catherine Lassez. An expert decision–support system for option–based investment. *Computer Mathematics with Applications*, 20(9/10):1–14, 1990.

[80] J. Jaffar, J-L Lassez, and M. J. Maher. A theory of complete logic programs with equality. *Journal of Logic Programming*, 1(3):211–223, 1984.

[81] Joxan Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, January 1990.

[82] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. Technical Report 86/73, Monash University, Victoria, Australia, June 1986.

[83] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[84] Joxan Jaffar, Michael Maher, Peter Stuckey, and Roland Yap. Output in CLP($\mathcal{R}$). In *Proceedings of the 1992 Conference on Fifth Generation Computer Systems, Tokyo*, 1992.

[85] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference*, pages 196–218, Melbourne, Australia, May 1987. MIT Press. Revised version of Monash University technical report number 86/75, November 1986.

[86] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. Technical Report RC 16292 (#72336) 11/15/90, IBM Research Division, November 1990.

[87] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation, San Francisco*, pages 128–139, June 1992.

[88] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, July 1992.

[89] Joxan Jaffar, Spiro Michaylov, and Roland Yap. A methodology for managing hard constraints in CLP systems. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 306–316, Toronto, Canada, June 1991.

[90] Niels Jørgensen. *Abstract interpretation of constraint logic programs.* PhD thesis, Roskilde University Center, Denmark, 1992.

[91] Niels Jørgensen. Kim Marriott, and Spiro Michaylov. Some global compile-time optimizations for CLP($\mathcal{R}$). In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 420–434, San Diego, CA, October 1991. MIT Press.

[92] D. M. Kaplan. Some completeness results in the mathematical theory of computation. *Journal of the ACM*, 15(1):124–134, January 1968.

[93] N. Karmarkar. A new polynomial–time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[94] L. G. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20(1):191–194, 1979.

[95] Donald E. Knuth. *The METAFONTbook.* Addison Wesley, 1986.

[96] M. Konopasek and S. Jayaraman. Constraint and declarative languages for engineering applications: The TK!Solver contribution. *Proceedings of the IEEE*, 73(12), December 1985.

[97] R. Kowalski. Predicate logic as a programming language. In *IFIP Congress*, pages 569–574, Stockholm, 1974. North–Holland.

[98] R. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–44, 1988.

[99] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, , and N. Adams. Orbit: an optimizing compiler for scheme. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 219–233, Palo Alto, June 1986.

[100] D. Kuehner and R. Kowalski. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[101] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing logic programming languages with polymorphic typing. Technical Report CS–1991–39, Duke University, Durham, North Carolina, October 1991.

[102] Sivand Lakmazaheri and William J. Rasdorf. Constraint logic programming for the analysis and partial synthesis of truss structures. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 3(3):157–173, 1989.

[103] A. Lane. Trilogy: A new approach to logic programming. *BYTE*, 13:145–151, March 1988.

[104] Catherine Lassez, Ken McAloon, and Roland Yap. Constraint logic programming and options trading. *IEEE Expert, Special Issue on Financial Software*, 2(3):42–50, August 1987.

[105] J. H. M. Lee and M. H. van Emden. Adapting CLP($\mathcal{R}$) to floating-point arithmetic. In *Fifth Generation Computer Systems*, pages 996–1003, Tokyo, Japan, 1992. (also appears as report LP-18 (DCS-183-IR) Univ. of Victoria).

[106] W. Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison–Wesley, 1988. Based on Ph.D Thesis, UNC Chapel Hill.

[107] D. Levitt. Machine tongues X: Constraint languages. *Computer music*, 8(1):9–21, 1984.

[108] Jiarong Li. Using constraints in interactive text and graphics editing. In P. A. Duce and P. Jancene, editors, *Eurographics*, pages 197–205, Nice, France, September 1988. North-Holland.

[109] P. Lim and P. J. Stuckey. A constraint logic programming shell. In P. Deransart and J. Maluszyński, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming, Linköping, Sweden, August 20–22*, number 456 in Lecture Notes in Computer Science, pages 75–88. Springer Verlag, 1990.

[110] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[111] Donald W. Loveland. Near-Horn prolog. In Jean-Louis Lassez, editor, *Proc. 4th International Logic Programming Conference*, pages 456–469, Melbourne, Australia, May 1987. MIT Press.

[112] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[113] G. S. Makanin. The problem of solvability of equations on a free semi-group. *Math. USSR Sbornik*, 32(2), 1977 (English translation AMS 1979).

[114] Kim Marriott and Harald Søndergaard. Analysis of constraint logic programs. In Saumya Debray and Manuel Hermenegildo, editors, *Proc. of the 1990 North American Conference on Logic Programming*, pages 521–540, Austin, TX, 1990. MIT Press.

[115] C. S. Mellish. An alternative to structure sharing in the implementation of a Prolog interpreter. In K. L. Clark and S.-A. Tarnlund, editors, *Proceedings of the 1st International Workshop on Logic Programming, 1980*, pages 99–106, Debrecen, Hungary, 1982. Academic Press.

[116] Nicolas Mercouroff and Aline Weitzman. Automatic analysis of the execution time of a class of parallel recursive algorithms. *SIAM Journal of Computing, to appear*, 199?

[117] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNCS/LNAI 595.

[118] Dale Miller. A theory of modules for logic programming. In *Symposium on Logic Programming*, pages 106–114, Salt Lake City, Utah, 1986. IEEE.

[119] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop*, pages 253–281, Tübingen FRG, December 1991. Springer-Verlag LNCS 475.

[120] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proc. 27th Annual Meeting of the Association for Computational Linguistics*, pages 247–256, Columbia University, 1986. Association for Computational Linguistics.

[121] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 448–462, London, July 1986. Springer Verlag LNCS 225.

[122] J. S. Moore. Computational logic: Structure sharing and proof of program properties, parts I and II. Technical Report DCL Memo 67, School of Artificial Intelligence, University of Edinburgh, Edinburgh, UK, 1974.

[123] Igor Mozetič and Christian Holzbaur. Integrating numerical and qualitative models within constraint logic programming. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 678–693, San Diego, CA, October 1991. MIT Press.

[124] K. Mukai. Anadic tuples in Prolog. Technical Report TR-239, ICOT, 1987.

[125] K. Mukai. A system of logic programming for linguistic analysis. Technical Report TR-540, ICOT, 1990.

[126] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Seattle, WA, August 1988. MIT Press.

[127] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 Conference on Lisp and Functional Programming*, pages 341–348, Nice, France, June 1990. ACM Press.

[128] Lee Naish. *Negation and Control in Prolog.* PhD thesis, Department of Computer Science, University of Melbourne, Parkville, Victoria, Australia, 1985. Appears as technical report 85/12, and as Springer LNCS 238.

[129] Greg Nelson. Juno, a constraint-based graphics system. *Computer Graphics*, 19(3):235–243, 1985.

[130] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, October 1979.

[131] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, 1990.

[132] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[133] W. Older and A. Vellino. Extending Prolog with constraint arithmetic on real intervals. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 14.1.1–14.1.4, 1990.

[134] A. V. Oppenheim, A. S. Willsky, and I. T. Young. *Signals and Systems.* Prentice-Hall, 1983.

[135] J. S. Ostroff. Constraint logic programming for reasoning about discrete event processes. *Journal of Logic Programming*, 11(3&4):243–270, 1991.

[136] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[137] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[138] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, CA, June 1989. IEEE.

[139] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[140] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[141] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988.

[142] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[143] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1989.

[144] David A. Plaisted. The occur-check problem in Prolog. In *Proceedings of the 1st IEEE Symposium on Logic Programming*, pages 272–280, Atlantic City, NJ, USA, 1984.

[145] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[146] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):32–41, 1965.

[147] P. Van Roy and A.M. Despain. The benefits of global dataflow analysis for an optimizing prolog compiler. In Saumya Debray and Manuel Hermenegildo, editors, *Proc. of the 1990 North American Conference on Logic Programming*, pages 501–515, Austin, TX, 1990. MIT Press.

[148] Peter Lodewijk Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming.* PhD thesis, University of California at Berkeley, 1990.

[149] K. Sakai and A. Aiba. CAL: A theoretical background of CLP and its applications. *Journal of Symbolic Computation*, 8(6):589–603, December 1989.

[150] K. Sakai and T. Matsuda. Several aspects of unification: Universal unification. Technical Report (Technical Memorandum) 0046, ICOT, 1984.

[151] K. Sakai and Y. Sato. Application of ideal theory to boolean constraint solving. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence*, 1990.

[152] Vijay Anand Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Computer Science Department, Carnegie Mellon University, January 1989. Available as technical report CMU-CS-89-108.

[153] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley and Sons, 1986.

[154] Adel S. Sedra and Kenneth C. Smith. *Microelectronic Circuits.* Holt-Saunders, 1982.

[155] Ehud Y. Shapiro. A subset of concurrent PROLOG and its interpreter. Technical Report TR-003, ICOT, Tokyo, 1983.

[156] Ehud Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[157] John C. Shepherdson. Negation as failure: A comparison of Clarke's completed data base and Reiter's closed world assumption. *Journal of Logic Programming*, 1(1):51–80, June 1984.

[158] John C. Shepherdson. Negation as failure II. *Journal of Logic Programming*, 2(3):185–202, October 1985.

[159] Stuart Shieber. An introduction to unification-based approaches to grammar. Technical Report CSLI Lecture Notes 4, Center for the Study of Language and Information, Stanford University, 1986.

[160] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[161] Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.

[162] H. Simonis. Formal verification of multipliers. In L.J.M. Claesen, editor, *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, November 1989. IFIP, North Holland, Elsevier Science Publishers.

[163] H. Simonis and M. Dincbas. Using logic programming for fault diagnosis in digital circuits. Technical Report TR-LP-18, E.C.R.C (European Computer-Industry Research Centre), December 1986.

[164] H. Simonis and M. Dincbas. Using an extended prolog for digital circuit design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, Munich, W.Germany, October 1987.

[165] H. Simonis and M. Dincbas. Using logic programming for fault diagnosis in digital circuits. In K. Morik, editor, *German Workshop on Artificial Intelligence (GWAI-87)*, pages 139–148, Geseke, W. Germany, September 1987. Springer-Verlag.

[166] H. Simonis, H. N. Nguyen, and M. Dincbas. Verification of digital circuits using chip. In G.J. Milne, editor, *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988. IFIP, North-Holland.

[167] H. Simonis and T. Le Provost. Circuit verification in chip: Benchmark results. In L.J.M. Claesen, editor, *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, pages 125–129, Leuven, Belgium, November 1989. IFIP, North Holland, Elsevier Science Publishers.

[168] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[169] G. L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., August 1980. Available as technical report MIT-AI TR 595.

[170] G. L. Steele and G. J. Sussman. Constraints. In *Proceedings of APL 79, in ACM SIGPLAN STAPL APL Quote Quad 9(4)*, pages 208–225, June 1979.

[171] L. Sterling. *The Practice of Prolog.* MIT Press, 1990.

[172] L. Sterling and E. Y. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[173] T. Sthanusubramonian. A transformational approach to configuration design. Master's thesis, Engineering Design Research Center, Carnegie Mellon University, 1991.

[174] Peter J. Stuckey. Incremental linear arithmetic constraint solving and detection of implicit equalities. *ORSA Journal of Computing*, 3(4):269–274, 1991.

[175] G. J. Sussman and R. M. Stallman. Heuristic techniques in computer-aided circuit analysis. *IEEE Transactions on Circuits and Systems*, 22(11), November 1975.

[176] G. J. Sussman and G. L. Steele. CONSTRAINTS — a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980.

[177] Ivan Sutherland. *A Man Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, January 1963.

[178] N. Suzuki and D. Jefferson. Verification decidability of Presburger array segments. In *Proc. Conf. on Theoretical Computer Science*, University of Waterloo, August 1977.

[179] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1948.

[180] A. Taylor. LIPS on a MIPS: Results from a Prolog compiler for a RISC. In David H. D. Warren and Peter Szeredi, editors, *Proc. of the 7th International Conference on Logic Programming*, pages 174–185, Jerusalem, Israel, 1990. MIT Press.

[181] Andrew Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, 1991.

[182] Joseph C. Tobias, II. Knowledge representation in the Harmony intelligent tutoring system. Master's thesis, Department of Computer Science, University of California at Los Angeles, 1988.

[183] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[184] C. J. van Wyk. *A Language for Typesetting Graphics.* PhD thesis, Department of Computer Science, Stanford University, June 1980.

[185] P. Voda. The constraint language Trilogy: Semantics and computations. Technical report, Complete Logic Systems, North Vancouver, B.C., Canada, 1988.

[186] P. Voda. Types of Trilogy. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 580–589, Seattle, WA, 1988. MIT Press.

[187] Clifford Walinsky. CLP(Σ^*): Constraint logic programming with regular sets. In David. H. D. Warren and Peter Szeredi, editors, *Logic Programming: Proceedings* 6^{th} *International Conference*, pages 181–196, Jerusalem, June 1989. MIT Press.

[188] D. H. D. Warren. Implementing Prolog — compiling logic programs 1 and 2. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1977.

[189] D. H. D. Warren. An abstract Prolog instruction set. Technical Report (Note) 309, SRI International, Menlo Park, California, October 1983.

[190] D. H. D. Warren, F. Pereira, and L. M. Pereira. User's guide to DECsystem–10 Prolog. Technical Report Occasional Paper 15, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1979.

[191] Roland Yap Hoc Chuan. CLP($\mathcal{R}$): Some aspects of the constraint paradigm and the implementation. Master's thesis, Department of Computer Science, Monash University, August 1988.

[192] Roland Yap Hoc Chuan. Restriction site mapping in CLP($\mathcal{R}$). In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 521–534, Paris, France, June 1991. MIT Press.

Index